SOFTWARE METHOD LEVEL SPECULATION FOR JAVA

by

Christopher John Francis Pickett

School of Computer Science

McGill University, Montréal, Québec, Canada

April 2012

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
DOCTOR OF PHILOSOPHY

Copyright © 2012 by Christopher John Francis Pickett

Abstract

Speculative multithreading (SpMT), also known as thread level speculation (TLS), is a dynamic parallelization technique that relies on out-of-order execution, dependence buffering, and misspeculation rollback to achieve speedup of sequential programs on multiprocessors. A large number of hardware studies have shown good results for irregular programs, as have a smaller number of software studies in the context of loop level speculation for unmanaged languages.

In this thesis we explore software method level speculation for Java. A software environment means that speculation will run on existing multiprocessors, at the cost of extra overhead. Method level speculation (MLS) is a kind of SpMT / TLS that creates threads on method invocation, executing the continuation speculatively. Although MLS can subsume loop level speculation, it is still a relatively unexplored paradigm. The Java programming language and virtual machine environment are rich and complex, posing many implementation challenges, but also supporting a compelling variety of object-oriented programs.

We first describe the design and implementation of a prototype system in a Java bytecode interpreter. This includes support for various MLS components, such as return value prediction and dependence buffering, as well as various interactions with features of the Java virtual machine, for example bytecode interpretation, exception handling, and the Java memory model. Experimentally we found that although high thread overheads preclude speedup, we could extract significant parallelism if overheads were excluded. Furthermore, profiling revealed three key areas for optimization.

The first key area for optimization was the return value prediction system. In our initial model, a variety of predictors were all executing naïvely on every method invocation, in order that a hybrid predictor might select the best performing ones. We developed an

adaptive system wherein hybrid predictors dynamically specialize on a per-callsite basis, thus dramatically reducing speed and memory costs whilst maintaining high accuracy.

The second area for optimization was the nesting model. Our initial system only allowed for out-of-order nesting, wherein a single parent thread creates multiple child threads. Enabling support for in-order nesting exposes significantly more parallelization opportunities, because now speculative child threads can create their own children that are even more speculative. This required developing a memory manager for child threads based on recycling aggregate data structures. We present an operational semantics for our nesting model derived from our implementation.

Finally, we use this semantics to address the third area for optimization, namely a need for better fork heuristics. Initial heuristics based on online profiling made it difficult to identify the best places to create threads due to complex feedback interactions between speculation decisions at independent speculation points. This problem grew exponentially worse with the support for in-order nesting. Instead, we chose to clarify the effect of program structure on runtime parallelism. We did this by systematically exploring the interaction between speculation and a variety of coding idioms. The patterns we identify are intended to guide both manual parallelization and static compilation efforts.

Résumé

L'exécution spéculative multifils (SpMT), aussi connue sous le nom de spéculation au niveau des fils d'exécution (TLS), est une technique de parallélisation dynamique qui se base sur l'exécution dans le désordre, la mise en mémoire tampon des dépendances spéculatives, et le refoulement des erreurs de spéculation pour atteindre l'accélération des programmes séquentiels sur les multiprocesseurs. D'extensives études architecturales ont révélé de bons résultats dans le cas des programmes irréguliers, tout comme plusieurs études logiciel dans la spéculation au niveau des boucles dans un langage non géré.

Dans ce mémoire, nous explorons la spéculation logiciel au niveau des méthodes pour Java. Un environnement logiciel signifie que la spéculation s'exécute sur les multiprocesseurs existants, au coût de charge additionnelle. La spéculation au niveau des méthodes (MLS) est une sorte de SpMT / TLS où des fils d'exécution sont créés à chaque invocation de méthode, exécutant les instructions qui suivent de manière spéculative. Malgré la possibilité de subsomption de la spéculation au niveau des boucles par la spéculation au niveau des méthodes, ce paradigme est relativement peu exploré. Le langage de programmation Java, ainsi que l'environnement de sa machine virtuelle, sont riches et complexes, ce qui pose plusieurs difficultés à l'implémentation, mais qui a l'avantage de supporter une grande variété de programmes orientés objet.

Nous décrivons d'abord la conception et l'implémentation d'un système prototype dans un interpréteur de code-octet Java. Cette implémentation supporte une variété de composantes de la spéculation au niveau des méthodes, telles la prédiction des valeurs de retour, la mise en mémoire tampon des dépendances spéculatives, ainsi qu'une variété d'interactions avec des caractéristiques de la machine virtuelle Java (JVM), par exemple, l'interprétation du code-octet, le gestion des exceptions et le modèle de la mémoire de Java.

Des expériences nous ont permis de constater d'encourageants résultats quant à la parallélisation des programmes, malgré une charge additionnelle importante dû à l'embranchement des fils d'exécution, ce qui empêche d'obtenir une accélération significative. De plus, le profilage effectué a révélé trois secteurs d'optimisation importants.

La première optimisation étudié est la prédiction des valeurs de retour. Notre modèle initial utilisait plusieurs outils de prédiction différents pour chaque invocation de méthode, afin qu'un outil de prédiction hybride puisse choisir les plus performants. Nous avons développé un système adaptatif où les outils de prédiction hybrides se spécialisent dynamiquement pour chaque site d'invocation, réduisant drastiquement la charge mémoire additionnelle et les ralentissements tout en préservant un haut degré de précision.

Le deuxième secteur d'optimisation étudié est celui des modèles d'emboîtement. Notre modèle initial permettait seulement l'emboîtement dans le désordre, où un seul fil d'exécution peut en créer plusieurs fils d'exécution spéculatifs. L'introduction du support de l'emboîtement en ordre expose un nombre conséquent d'opportunités de parallélisation, parce qu'un fil d'exécution spéculatif peut maintenant en créer un autre, encore plus spéculatif. Pour ce faire, nous avons développé un gestionnaire de mémoire pour les fils d'exécution spéculatifs basé sur le recyclage des structures de donnée agrégés. Nous présentons une sémantique des opérations de notre modèle d'emboîtement dérivée de notre implémentation.

Finalement, nous utilisons cette sémantique des opérations pour optimiser nos heuristiques d'embranchement. Initialement, l'utilisation d'heuristiques basées sur les données recueillies au temps d'exécution rendait difficile l'identification des meilleurs points pour créer de nouveaux fils d'exécution, dû à l'interaction entre les décisions spéculatives prises à différents points. Ce problème s'amplifie de façon exponentielle avec le support de l'emboîtement en ordre. Comme alternative, nous avons choisi de clarifier l'effet de la structure des programmes sur son parallélisme à l'exécution. Pour ce faire, nous avons exploré systématiquement l'interaction de la spéculation avec une variété d'idiomes de programmation. Les patrons identifiés sont utiles pour guider les efforts de parallélisation manuelle ainsi que de compilation statique.

Acknowledgments

There were various people in supervisory positions that helped me see this work to completion. First and foremost I would like to thank my supervisor Clark Verbrugge. This work would not have been possible with his guidance, enthusiasm, perseverance, and good humour. Allan Kielstra at IBM was a great source of technical expertise as well as moral support. Laurie Hendren was Clark's closest colleague and offered valuable arms-length opinions over the years. Etienne Gagnon created the SableVM Java virtual machine and helped me develop the extensions to it described in this thesis.

I worked as part of the the Sable Research Group in the School of Computer Science and met many interesting and helpful students there. The senior Ph.D. students I knew were Ondřej Lhoták and Feng Qian; the ones in my cohort were Dayong Gu, Grzegorz Prokopski, and Eric Bodden; and the new ones I've only known briefly are Nurudeen Lameed and Rahul Garg. Zhen Cao is apparently following in my footsteps... be careful, Zhen! There are of course a great number of M.Sc. students that I've known and valued the relationships with over the years as well. The ones I worked with closely on projects and would like to thank specifically are Félix Martineau, Haiying Xu, and Richard Halpert.

I have received truly generous amounts of funding during my degree. I was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) with two-year PGS A, two-year CGS D, and one-year CRD stipends. McGill awarded me a two-year Richard H. Tomlinson Master's in Science Fellowship. Le Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) provided travel support, as did SIGPLAN PAC and SIGSOFT CAPS. Finally, the IBM Toronto Centre for Advanced Studies (CAS) awarded me a four-year CAS Fellowship, as well as travel and living expenses while visiting Toronto.

I had the opportunity to teach a compiler design course twice, once in 2007 and again in 2009. Thanks to Laurie and Clark for arranging this, to Peng Zhang, Jesse Doherty, and Nurudeen Lameed for being great TA's, and to my students for getting really excited and supporting my first teaching experiences. This course was originally Laurie's creation, and was the one that motivated me to study compilers, virtual machines, and programming languages in the first place after I took it myself as a biochemistry undergraduate.

Thanks also to those people who offered their direct feedback on the written aspects of this work, whether in the form of informal exchanges or formal reviews. Your comments and suggestions have greatly shaped the final outcomes. In particular, my internal and external examiners Laurie Hendren and Greg Steffan provided detailed critiques of the thesis. Similarly, the feedback from the various presentations I've given over the years has been invaluable and a great source of new ideas. In the last stages, Annie Ying was instrumental in helping me prepare for my thesis defense, and Olivier Savary worked diligently to translate my thesis abstract into French.

My parents greatly encouraged my academic interests growing up. Roughly speaking, I'd attribute the science part to my mother and the technology part to my father. My dad's work with compilers and virtual machines undoubtedly contributed to my research choices here. In addition to my parents, many people played important roles in various supportive capacities. In particular, Jim Robinson has been consistently present during the good and bad moments of the last three years. I am also grateful to my friends, many of whom I made in Computer Science, for sticking it out with me through both fair and stormy weather.

Last but not least I would like to thank Emily Leong for her kindness, companionship, and love. It's been just great having you by my side throughout all of grad school. I've changed a lot since I started this degree and it's largely due to my relationship with you. Thank-you for sharing your own creative metamorphosis with me, it's deeply inspiring to watch you grow each day into an ever more wonderful woman.

To Emily

Table of Contents

Abstract **i**

Résumé **iii**

Acknowledgments **v**

Table of Contents **viii**

1 Introduction **1**

1.1 Speculative Multithreading 3
1.2 Software Method Level Speculation for Java 5
1.3 Contributions 11
1.3.1 Software Method Level Speculation 12
1.3.2 Java Language Support 13
1.3.3 Experimental Framework and Analysis 14
1.3.4 Return Value Prediction 16
1.3.5 Nested Speculation 19
1.3.6 Fork Heuristics 21
1.4 Roadmap 23

2 Software Method Level Speculation for Java **25**

2.1 Introduction 26
2.1.1 Contributions 28
2.2 Framework 29

2.2.1 Overview . . . 29
2.2.2 Multithreaded Execution Mode . . . 31
2.2.3 Single-Threaded Execution Mode . . . 34
2.2.4 System Configuration . . . 35
2.2.5 Logging and Trace Generation . . . 35
2.3 Speculative Method Preparation . . . 36
2.3.1 Static Analysis and Attribute Parsing . . . 36
2.3.2 Fork and Join Insertion . . . 37
2.3.3 Bytecode Instruction Modification . . . 38
2.3.4 Parallel Code Array Generation . . . 41
2.4 Speculative Runtime Support . . . 42
2.4.1 Thread Forking . . . 42
2.4.2 Priority Queueing . . . 43
2.4.3 Return Value Prediction . . . 46
2.4.4 Dependence Buffering . . . 47
2.4.5 Stack Buffering . . . 50
2.4.6 Thread Joining . . . 50
2.5 Java Language Considerations . . . 52
2.5.1 Class Loading . . . 52
2.5.2 Object Allocation . . . 52
2.5.3 Garbage Collection . . . 53
2.5.4 Native Methods . . . 53
2.5.5 Exceptions . . . 54
2.5.6 Synchronization . . . 54
2.5.7 The Java Memory Model . . . 54
2.6 Experimental Analysis . . . 55
2.6.1 Return Value Prediction . . . 56
2.6.2 Static Analysis Integration . . . 57
2.6.3 Speculation Overhead . . . 58
2.6.4 Speculative Parallelism . . . 61
2.6.5 Runtime Profiling . . . 62

2.6.6 Speculation Behaviour 63
2.6.7 Speedup 65
2.7 Conclusions and Future Work 68

3 Adaptive Software Return Value Prediction **71**
3.1 Introduction 72
3.1.1 Contributions 75
3.2 Predictor Unification Framework 76
3.2.1 History-Based Predictors 77
3.2.2 Argument-Based Predictors 81
3.2.3 Composite Predictors 82
3.3 Experimental Setup 91
3.3.1 Benchmarks 92
3.3.2 Communication Overhead 94
3.4 Initial Performance Evaluation 95
3.4.1 Accuracy 96
3.4.2 Speed 97
3.4.3 Memory Consumption 97
3.4.4 Sub-Predictor Comparisons 100
3.5 Hybrid Adaptivity 104
3.5.1 Offline Specialization 105
3.5.2 Online Specialization 106
3.5.3 Performance Comparisons 108
3.6 Conclusions 110
3.7 Future Work 110

4 Nested Method Level Speculation & Structural Fork Heuristics **113**
4.1 Introduction 114
4.1.1 Contributions 116
4.2 Child Thread Memory Allocation 117
4.2.1 Ownership Based Allocation 118

4.2.2 Multithreaded Freelist Allocation 119
4.2.3 Discussion 120
4.3 Nested MLS Stack Abstraction 123
4.4 Individual MLS Models 130
4.5 Speculation Patterns 136
4.5.1 Straight-Line 137
4.5.2 If-Then 138
4.5.3 Iteration 140
4.5.4 Tail Recursion 141
4.5.5 Head Recursion 143
4.5.6 Mixed Head and Tail Recursion 145
4.5.7 Olden Benchmark Suite 147
4.5.8 Tree Traversals 147
4.5.9 Divide and Conquer 150
4.5.10 Discussion 151
4.6 Conclusions and Future Work 152

5 Related Work 155
5.1 Hardware Architectures 157
5.2 Language Semantics 159
5.3 Software SpMT 160
5.4 Method Level Speculation 163
5.5 Dependence Buffering 165
5.6 Transactional Memory 166
5.7 Speculative Locking 167
5.8 Return Value Prediction 168
5.9 Memory Management 172
5.10 Nested Speculation 174
5.11 Irregular Parallelism 175
5.12 Fork Heuristics 177
5.13 Misspeculation Reduction 179

5.14 Non-Speculative Method Level Parallelism 180
5.15 Functional Language Speculation 183
5.16 Other Uses of Speculation 184

6 Conclusions & Future Work **185**

6.1 Conclusions 185
6.2 Future Work 188
6.2.1 Speculative Locking 188
6.2.2 Manual Parallelization 188
6.2.3 Static Analysis 189
6.2.4 Dynamic Purity Analysis for Speculation 190
6.2.5 JIT Compiler Speculation 190
6.2.6 New Systems 191

Bibliography **193**

Chapter 1
Introduction

Many computer programs are *serial* or *sequential*: there is only one thread of control and they are designed for, written for, compiled for, and executed on a single processor. These programs are straightforward to handle at all levels. The disadvantage is that the execution time of the program is bound by the speed of the processor, and cannot be helped by adding extra processors to the system. One important optimization is to use *parallel*, *concurrent*, or *multithreaded* programming: now the execution can be spread across multiple processors. This leads to program *speedup*, which for parallelization work is calculated simply as the sequential run time divided by the parallel run time.

The standard approach to creating parallel programs is to write them by hand, either by converting an existing sequential application or by targeting multiple processors in the initial design. This process is known as *manual parallelization*, and results in a program containing *explicit parallelism*. Synchronization primitives are required to manage conflicting data accesses: these can take the form of monitors, semaphores, critical sections, barriers, volatile variables, threads, processes, and atomic operations. The dominant paradigm in industry is *lock-based programming*. A program is split into multiple *threads*, each of which has its own flow of control and may execute concurrently on a separate processor. Threads communicate by first acquiring *locks*, which grant mutually exclusive access to *critical sections*, regions of code that modify shared data. Once finished with a critical section, a thread releases its lock and resumes non-shared parallel processing. There are many books on concurrent programming that explore these concepts in detail.

A well-written parallel program can perform very well, achieving great scalability to many processors, depending on the complexity of the application in question. However, lock-based programming is inherently difficult, with issues of deadlock, livelock, fairness, and safety being central concerns. It is widely considered a tedious and error-prone process. Recently *transactional memory* has been extensively explored as a viable replacement for lock-based programming [LR06]. In this programming model, threads acquire access to *atomic sections*, rolling back the operations inside if they fail to complete. On the surface, this model is simpler than lock-based programming, and it has the added benefit of allowing *out-of-order*, *speculative* execution of the code inside atomic sections. However, like any parallel programming, it still requires programmers to annotate code with synchronization primitives, and there are efficiency concerns when compared to a lock-based approach [CBM+08].

In contrast to these methods that require programmers to modify their software is *automatic parallelization* via compilers and runtime systems. There are two fundamental approaches. First, one can automatically convert a sequential program into a parallel program. Second, one can take a parallel program and identify sequential regions and parallelize them. In either case, the task is to expose *implicit parallelism* hidden in the program to obtain speedup. Automatic parallelization is perhaps even more difficult than manual parallelization due to the requirement for general applicability; the key motivation is that multithreaded programming is so difficult that it needs as much automation as possible.

Like any program optimization, the structure of the input sequential source code is a significant contributor to the success of an automatic tool. For instance, regular array-based computations are much more easily parallelized than irregular pointer-based ones. Beyond applicability and safety, a primary concern is the amount of runtime overhead incurred by parallelization, and whether this is offset by the gains in parallelism to yield a net speedup. Approaches may be fundamentally *static*, or *ahead-of-time*, decided based on pre-execution compiler analysis without knowledge of program inputs, or they may be *dynamic*, or *just-in-time*, decided based on runtime analysis of the inputs. Some approaches exploit profiling information collected at runtime; *offline profiling* is a static approach that uses this information to improve subsequent program runs, whereas *online profiling* is dynamic, applying the information before the program completes execution. Finally, automatic program trans-

formations, and in particular parallelizing ones, may be *conservative*, *pessimistic*, *in-order*, or *non-speculative*: they respect program structure and provide safety guarantees for all inputs; or they may be *liberal*, *optimistic*, *out-of-order*, or *speculative*: they allow for unsafe program behaviours at runtime, but provide a monitoring mechanism to catch them and roll back if necesary.

1.1 Speculative Multithreading

In this thesis we explore *speculative multithreading* (SpMT), also known as *thread level speculation* (TLS). It is a dynamic parallelization technique that relies on out-of-order speculative execution of sequential programs to achieve speedup on multiprocessors. In an ideal implementation it is fully automatic, but many proofs-of-concept rely on some form of manual guidance. The first step is to split or *decompose* a sequential region of code into one non-speculative *parent* thread and multiple speculative *child* threads, also referred to as child *tasks*. This process may be static or dynamic. At runtime, children are created or *forked*, and they begin speculative execution on separate processors, buffering main memory accesses and prohibiting I/O to allow rollback or abortion in the case of dependence violations, thus executing in a safe and isolated fashion. At some future point the children are joined, and in the absence of dependence violations their computations are merged back into the parent. The advantage of SpMT is its ability to parallelize applications that traditional static compilers cannot handle due to unknown runtime data dependences. It is typically considered at a hardware level, where proposed systems are capable of showing good speedups in simulation-based studies; prototype SpMT CPUs have also been manufactured. However, SpMT and related forms of optimistic execution have also shown viability in various software models, motivating the specific focus of this thesis, namely an investigation of software SpMT at the Java virtual machine level.

The salient runtime features of speculative multithreading systems include: 1) a means to create speculative threads that execute future code in an out-of-order fashion; 2) support for either memory access *dependence buffering* or undo logging; 3) some mechanism to detect violations and either undo or prevent unsafe operations; and 4) a means to either commit the speculative execution in a manner that preserves original program semantics,

or abort the execution safely. Our use of the term dependence buffering includes both dependence tracking to prevent read-after-write (RAW) or *true* dependence violations as well as buffering of speculative modifications to prevent write-after-read (WAR) or *anti* dependence violations and write-after-write (WAW) or *output* dependence violations. Undo logging differs from dependence buffering in that it provides a rollback mechanism that allows for speculative writes to go directly to main memory. Before execution, a basic parallelization or partitioning or thread decomposition strategy is required: speculation may occur at any or all of the basic block, loop, or method levels. A compiler and its accompanying dataflow analysis framework is necessary to partition programs into threads at the basic block and loop levels. Strictly speaking, static analysis is not necessary to speculate over method calls, as the runtime system or speculative hardware can detect and instrument these. Although there exists variance between the parallelization strategies, there is also considerable commonality. Two issues that must be addressed when preparing any sequential code for speculative execution are where precisely to fork new speculative threads, and where to verify the speculative execution and commit the results to memory if correct. Perhaps most importantly, the parallelization occurs at the thread level as opposed to the instruction level, and requires two or more CPUs, cores, or virtual cores for speedup.

There is extensive work on SpMT, much of it demonstrating compelling speedups, and a large number of novel SpMT hardware architectures and compilers that target them have been evaluated [KN07]. The standard claim is that novel hardware is absolutely necessary due to high speculation overheads, precluding an efficient software-only implementation. Most of this prior SpMT research also targets loop parallelization, because many important applications spend a majority of their computation time inside loops. Finally, the primary focus has been on imperative, regular, and often scientific programs written in C, C++, and Fortran, because these are seen as the applications that are most important to parallelize and also most scalable.

The problems with the general trends in SpMT research are correspondingly three-fold. First, new hardware is prohibitively expensive, and commercial uptake of research ideas can be slow and limited. Furthermore, hardware implementations are necessarily non-portable and platform-specific. Second, there is a large class of applications that are significantly more irregular and not loop-based that may well benefit from parallelization,

particularly with the prevalence of multi-core processors in modern machines. Third, the focus on C, C++, and Fortran means that existing studies tend to exclude richer languages such as Java and C#, with even less attention given to dynamic languages such as Python and MATLAB, thereby ignoring the impact of higher level language semantics and a virtual machine environment on speculation. The net result is that there are several interesting areas in SpMT research that remain relatively unexplored.

The prior work that does exist in these relatively unexplored areas shows significant potential. Specifically, speculative execution of Java programs can be effective, particularly at the method level [CO98, HBJ03, WK05], and there have been a number of advances in software SpMT research. In summary, overheads are high [PM01], coarser thread granularities help offset these overheads [DSK+07], manual source code changes are effective [KL00, KPW+07, OM08, WJH05], loop level speculation in software is viable [CL05], and functional programs are good candidates for automatic parallelization [HS07]. Related work is discussed in greater detail in Chapter 5.

1.2 Software Method Level Speculation for Java

Our overall approach is to expand the scope of applicability of SpMT by turning to these unexplored areas. Inspired by hardware designs, we provide a relatively complete and self-contained implementation of SpMT purely in software. Our focus for speculation is the method level, which allows us to capture the behaviour of irregular and recursive programs. In particular, object-oriented programs are often method-based rather than loop-based, and so any parallelization strategy should be adapted accordingly. Finally, we target Java at the virtual machine level, addressing many of the complexities contributed by this language and development platform. We take a broad perspective and use a methodical approach with the intent to uncover aspects of speculative execution that are at once peculiar, interesting, and foundational. This is the first software-only, method level, Java VM based implementation of SpMT. There is significant room for novel contribution here: to reiterate, most SpMT research to date has focused on loop level speculation for regular applications written in unmanaged languages such as C, C++, and Fortran that run on simulated hardware SpMT architectures.

There are two main arguments for software-only SpMT. First, it runs on existing multiprocessor machines without the need for specialized hardware. There are considerable time and money savings if hardware architecture changes can be avoided; not only is hardware expensive to produce, but at an experimental research level accurate hardware simulations are orders of magnitude slower than those in equivalent software systems. Second, a software system is not only more immediately useful, but also more easily modified. The malleability of software allows for clean design, refactoring, abstraction, extension, and portability. These qualities in turn are conducive to moving between a practical experimental system and a formal theoretical one. And, if new hardware support for SpMT emerges, software systems can be readily adapted to take advantage of it.

However, a software-only approach now introduces two particular challenges with respect to basic feasibility. First, as a hardware problem, the issues of ensuring correctness under speculative execution have been well defined; different rollback or synchronization approaches are sufficient to guarantee correct program behaviour at a low level. Further, hardware simulations permit simplifying abstractions by eliding portions of the program instruction stream under consideration, facilitating experimentation. However, software approaches cannot rely on the low level mechanisms of the underlying hardware to support speculation, and so must provide high level, intricate support to ensure correct language semantics. Further, for a non-simulated software system that does not elide instructions to work with arbitrary programs, it must account for the full source language and complete set of runtime behaviours. Second, software overheads are a much greater barrier to speedup than hardware overheads, often orders of magnitude higher, and may require significant high level optimization, whereas a specialized hardware system can complete many important operations in only a few cycles. Software versions of novel hardware circuits such as thread pools, dependence buffers, and value predictors may involve completely different approaches because of the inherent serialization. To summarize, any software system is bound to expose conflicts between the language and speculation, providing an excellent opportunity to resolve them; and addressing software overheads directly yields insight into where hardware support would help the most, if at all.

We refer to our choice of basic thread partitioning strategy as *method level speculation* (MLS). The literature may substitute ‘procedure’, ‘function’, ‘module’, or ‘subroutine’ in

place of ‘method’, and may also omit ‘level’; *speculative method level parallelism* (SMLP) is yet another term, alternatively found with corresponding substitutions for ‘method’. Under MLS, a *parent* thread forks a speculative *child* thread at a method call. The child executes the *continuation* past the return point, as if the call has already returned, while the parent executes the target method. Memory dependences in the child are either logged or buffered, such that any changes can be rolled back or discarded if necessary. When the parent returns from the call it joins its child, validates its state, and either commits the state to main memory or aborts the speculative execution accordingly. A successful commit allows the parent to jump ahead to the furthest point reached by the child. Given low enough overheads, the resultant parallelism is then a source of speedup on multiprocessor machines. Otherwise, the parent simply re-executes the child’s body.

Although the distinction between which thread is the child and which is the parent varies, MLS can be seen as the most optimistic and most automatic of a number of continuation-based parallelization schemes: futures, safe futures, parallel call, and implicit parallelization for functional languages. MLS suits our intended domain of irregular, non-numeric applications. Further, it is practical for implementation because all fork and join points already exist as method calls and returns, which means that in the strictest sense a static compiler analysis is not required to identify them. MLS is also able to subsume loop level speculation, which is often more suitable for regular applications, by outlining or extracting loop bodies into method calls. Extensions to the basic technique can accomodate unrolled and nested loops. This means that in theory, MLS can expose at least as much parallelism as loop level speculation; practical concerns such as the overhead of method invocation affect the eventual outcome. We examine this feature of MLS in Chapter 4. Finally, any implementation of MLS will need to consider processes for creating speculative code, forking and joining child threads, and handling method return values. We will explore these issues at length.

Figure 1.1 depicts the general MLS execution model. The left hand side shows the sequential execution of a method call in Java bytecode before parallelization. First the non-speculative parent thread T1 executes the pre-invocation instructions; next, it executes an invoke instruction and enters the target method, executing the instructions of its body; finally, T1 returns from the call and executes the post-invocation or continuation instructions

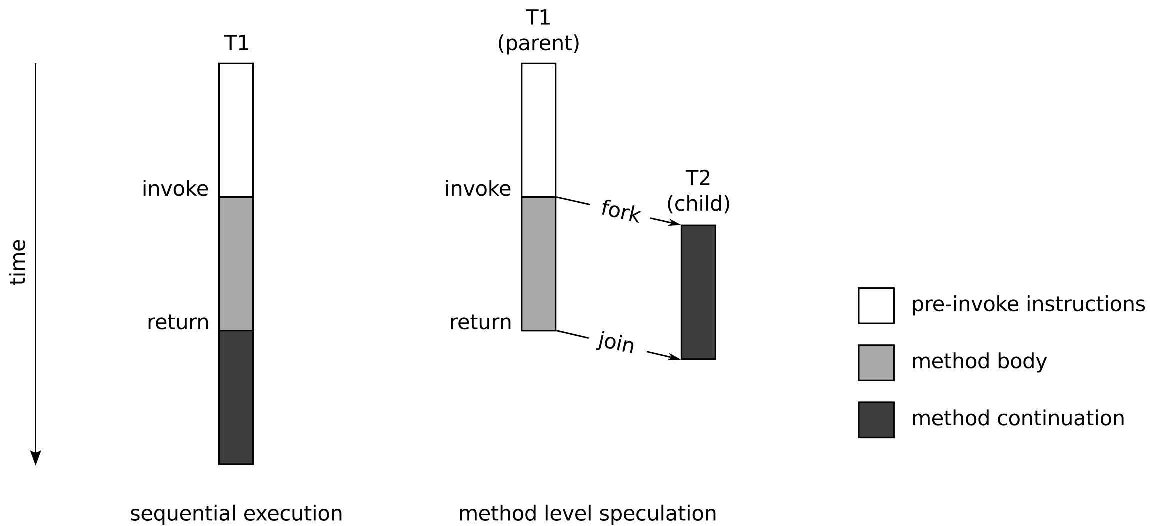


Figure 1.1: *Sequential execution vs. method level speculation.*

that follow the return point, located immediately after the invoke in the calling method. The right hand side shows the speculative execution of the same Java bytecode after parallelization via method level speculation. Upon reaching the invoke, or method callsite, the non-speculative parent thread T1 forks a speculative child thread T2. If the method is non-void, a predicted return value is pushed on T2's Java operand stack. T2 then continues past the return point, executing the continuation speculatively and in parallel with T1's non-speculative execution of the target method body. T2 provides strong isolation by buffering all reads from and writes to main memory and by stopping execution if any illegal instructions are encountered. When T1 returns from its call it joins T2, first signalling it and then waiting for it to stop, if it has not already stopped of its own accord. If the actual return value matches the predicted return value, and there are no dependence violations between T2's buffered reads and post-invoke values in main memory, T2's buffered writes are committed and non-speculative execution jumps ahead to where T2 left off. This yields parallelism, which in turn yields speedup if overheads are low. If there *are* dependence violations or the prediction is incorrect, T2 is simply aborted and the continuation is re-executed non-speculatively.

Note that in some implementations of method level parallelism, the distribution of work

in Figure 1.1 is inverted, such that the method body is executed in T2 and the continuation is executed in T1. This paradigm does offer advantages in terms of data locality if the memory accessed by the caller method pre-invoke instructions and continuation is relatively disjoint from that accessed by the callee method body. Also, if the execution model is non-speculative, reversing the roles of parent and child means the continuation stack frame can be used without copying it to a new thread. However, we did not consider this approach for three reasons. First, it adds significant complexity to our speculative stack model in terms of representation, clarity, and the mechanism for lazy stack frame buffering. This model is discussed completely in Chapter 4. Second, our design supports thread creation even when no free processors are available. Under this alternative paradigm, if no free processors were available, T1 would need to sleep so that T2 could execute on T1's processor. This would quite likely mean a higher fork overhead, even if thread pooling were used to alleviate the cost of a native thread context switch. Third, for multithreaded programs, if there are monitor exit or re-entry operations in the target method body, then using T2 to execute the body would require transferring lock ownership from T1 to T2.

In some cases, the parent method under MLS may be too long or too short with respect to the child continuation. There are several techniques to address this imbalance. One technique is to move the fork point either forwards into the method or backwards into the pre-invoke instructions. Another is to push the actual instructions surrounding the callsite into the target method, or to pull method body instructions out of the method and into the pre-invoke and/or continuation instructions. Finally, it is possible to filter out imbalanced fork points and promote balanced ones based on profiling, whether online or offline. The simple dynamic fork heuristics described in Chapter 2 do this implicitly. We use this last technique because it does not require static analysis and only requires wrapping callsites in the executing program. This methodology also supports one of our primary goals, which is to observe the "natural" properties of Java programs executing under MLS. Compiler techniques are certainly interesting, however, and we include them as part of our future work in Section 6.2.3.

Finally, our focus on Java at the virtual machine level is motivated by a number of considerations. Java is a widely used language, but the platform is not often considered for SpMT research or parallelization research in general, due to its complex runtime fea-

tures and the irregular object-oriented nature of programs written for it. Java's complex runtime features, which include garbage collection, dynamic class loading, bytecode interpretation, and exception handling, imply non-trivial interactions with SpMT and require any Java-based approach to software SpMT to fully account for them, which in turn implies significant research and development startup costs. On the other hand, the particular interactions with speculation are interesting in and of themselves, and any practical workarounds are most likely transferable to other complex languages. The complex runtime behaviour also makes a software approach particularly appropriate for Java SpMT, because there are high level language-specific optimizations available that do not necessarily translate well to generic hardware SpMT designs. The other main roadblock to parallelization, namely the irregular object-oriented programs that tend to be written in Java, arises from the structure of the core Java grammar and class library. These kinds of programs are traditionally the most difficult to parallelize, due to unknown boundary conditions within loops and a structure that is statically difficult to parallelize outside of loops. However, a part of good Java practice involves writing loosely coupled classes, which in turn suggests a rich source of speculative parallelism due to locally contained variable dependences. In particular, the MLS variant of SpMT accommodates Java's dense object-oriented method invocation structure, and has previously been demonstrated as a useful SpMT paradigm for the language. A further argument for Java MLS is that Java's structured call and return semantics imply well-behaved call stacks at runtime, which provide a similarly regular structure to Java-based MLS. Given the arguments in favour of software SpMT for Java, the Java virtual machine is a natural place to implement the required support, because both software SpMT and the JVM are a kind of virtual hardware. Our exclusive focus on the JVM leverages Java's write-once run-anywhere philosophy, and maximizes SpMT transparency, compatibility, and automation with respect to both existing Java programs and the underlying hardware. The combination of complex runtime features, object-orientation, language popularity, and relative lack of research attention when compared to languages such as C, C++, and Fortran make Java and the Java virtual machine a natural choice of implementation platform for any software MLS system.

Our methodology for constructing a software MLS system for Java was to start with an existing Java virtual machine and modify it to support MLS, and then use this as a base for

further development. We primarily used the SableVM Java bytecode interpreter [Gag02], an in-house project developed by our research group at McGill. We also used the Soot Java bytecode compiler framework [VR00], another in-house project, for some convenient code transformations and as a base for static analysis investigations. Our first goal was to build a complete prototype system. This would be a working implementation that was able to run the industry-standard SPEC JVM98 benchmarks under MLS that accounted for language and VM level safety requirements. It included software versions of several hardware components, including dependence buffers, return value predictors, and thread management structures, and various novel software components. It also incorporated various static analyses. The next step was to experiment with and understand this system by running benchmarks and profiling runtime behaviour. Based on these results, we identified where overhead and serialization issues were a significant concern, and determined some key SpMT-specific areas for optimization. At this point we refactored most of the SpMT logic into a separate software library, clarifying the design and providing a good context for developing optimizations. Then, guided by our profiling, we focused our remaining efforts on providing highly accurate software return value prediction that is both time and memory-efficient, supporting arbitrary nested method level speculation, and discovering a range of fork heuristics based on possible source level patterns. In general, we moved from concrete, practical concerns to more abstract, formal considerations once they became apparent in our work, with the intention to broaden applicability. We also expressly avoided manual intervention, wanting to create a fully automatic system that works with existing programs. We expand upon our primary contributions to the field in Section 1.3, and outline our research process and the overall structure of this thesis in Section 1.4.

1.3 Contributions

The broad contribution of this thesis is software MLS for Java. We began by designing and implementing a software MLS system, addressing Java-specific concerns, and developing an experimental framework. Building on this work, we explored several optimizations tailored for MLS in depth, namely return value prediction, arbitrary thread nesting, and method level structural fork heuristics. In each case we created an initial version and then

later added refinements guided by system profiling or development experience.

1.3.1 Software Method Level Speculation

We describe a software MLS design at the VM level. This design includes many distinct components, none of which are strongly tied to the Java language or dependent on our return value prediction, thread nesting, and fork heuristics optimizations. Indeed, for a basic MLS system, predictions can be made using any arbitrary value, thread nesting can be disabled, and threads can be forked with equal priority irrespective of program location or profiling data. The interaction with the Java language has many fundamental implications, but these can all be separated from Java-independent concerns.

Our design is inspired by hardware SpMT systems, and so includes software versions of hardware components. It also includes software-only components that would be difficult to implement in a hardware context. In general, there are many resources available in software, and we use these resources to facilitate ease and simplicity of implementation, rather than aim for maximum efficiency which is typically the focus of hardware designs. The resulting high level design allows for internal reuse throughout the system. As an example, we use the same hashtable design for dependence buffering, return value prediction, and callsite and method lookup, differentiated in behaviour by the software hash function employed.

We include a software VM as part of our design, which grants access to the entire program at runtime, in contrast with typical hardware SpMT systems that only have access to short instruction windows. This permits considerable flexibility in the overall design, and either reduces or eliminates the need for ahead-of-time compiler transformations. In practice we depend on an ahead-of-time compiler for some simple code insertions out of convenience, but our design would readily accomodate completely dynamic approaches because it works in the absence of static analysis.

The two common concerns of any SpMT system are memory access buffering and thread management. We describe a software dependence buffer that provides strong isolation of speculative reads and writes to heap and static memory. This is a simple form of software transactional memory based on layered hashtables. Our focus on high-level

design meant we were able to express our dependence buffer mechanism in very few lines, and the modularity of our system means it could be replaced by a wrapper around a highly refined software transactional memory library. A unique aspect of our software MLS system is that we buffer stack frames on entering and exiting methods speculatively. This stack frame buffering is independent of the heap and static dependence buffering described above. The result is that during exection local variables accesses do not incur hashtable lookup costs, and upon joining the speculative child local variables are not subject to redundant validation and can be committed immediately.

With respect to thread management, our design includes processes for forking and joining speculative threads, lightweight signalling between speculative and non-speculative threads, and hardware barriers and atomic instructions that ensure multiprocessor memory consistency. We include support for thread pooling and priority queueing based on speculative thread data structure reuse because OS-level threads and memory allocation are prohibitively expensive. Actual speculative code execution first requires a mechanism for transforming regular non-speculative code into speculative code that accesses the dependence buffer and other speculative runtime support components. Our design is based on creating duplicate speculative versions of method bodies and switching between non-speculative and speculative code at fork and join points. This kind of code manipulation is particularly suitable for software MLS, and is akin to the method body recompilations performed by an optimizing just-in-time compiler.

1.3.2 Java Language Support

The Java language and virtual machine have many complex behaviours, all of which must be considered when implementing a technique as pervasive as SpMT. Our MLS implementation is based on a Java 1.4 bytecode interpreter, as described in Chapter 2. This necessitated a thorough analysis of safety concerns with respect to VM and language features, including class loading, garbage collection, object allocation, synchronization, non-speculative multithreading, exception handling, native method execution, bytecode verifiability, and the Java memory model. It also includes a complete design for preparing Java bytecode for MLS and processes for forking and joining threads. Our treatment is

thorough enough to support speculation on every method invocation for SPECjvm98 and includes many non-obvious edge cases. Many of these solutions are interesting for their portability to other runtime environments. We finally present experimental data showing the impact of various Java language and VM features on whole system performance.

We use the significant amount of high level information available in the JVM in four different ways. First, we often extend speculation past the point where it would be forced to stop at the machine code level. For example, we can allocate objects speculatively in non-speculative heap space, acquiring a global VM lock in the process, because Java guarantees that they are visible only to threads that have references to them. Second, we prevent speculation from causing fatal errors. For example, by limiting speculative object references to the start of objects in VM heap space, the VM will never dereference an invalid pointer. Third, we optimize our implementation of software speculation support. For example, by exploiting knowledge of primitive type widths, we can reduce storage and computation costs in our return value prediction framework. Fourth, we draw on the wealth of VM information available for profiling and analysis purposes, including symbol names, types, threads, instructions, callsites, methods, and classes. This supports both debugging and optimizing the speclative system, as well as analysing the runtime behaviour of individual benchmarks under speculation.

1.3.3 Experimental Framework and Analysis

To advance the state of the art in SpMT research, we needed not only to design and implement novel optimizations, but to evaluate them with an experimental framework. However, prior to this work, there was no existing software MLS system for Java, and accordingly no experimental framework. Thus in parallel with the development of our optimizations, which are motivated by the need for performance, we also developed a complete experimental framework, which is motivated by the need for analysis. We use our framework to profile, analyse, and understand the behaviour of our system as a whole, individual components of this system, and a suite of standard benchmark programs that run on top of it. The experimental results from our initial implementation in Chapter 2 formed the basis for the optimizations presented in Chapters 3 and 4, as described in Section 1.4.

Our software MLS for Java experimental framework is named SableSpMT. It is an extension of the SableVM Java virtual machine [Gag02]. SableSpMT provides a convenient hardware abstraction layer by operating at the bytecode instruction level, takes the full Java language and VM specification into account, supports static analysis through the Soot bytecode compiler framework [VR00] and parsing of Java classfile attributes [PQVR+01], and runs on existing multiprocessor systems. SableSpMT provides a full set of MLS support features, including generic method level speculation and return value prediction. Our work is designed to facilitate SpMT research, and includes a unique debugging mode, significant instrumentation and runtime logging, online and offline profiling, a range of built-in performance metrics, and portability amongst the features that make it appropriate for experimentation and prototyping new designs.

After our initial implementation, we separated out the majority of the speculation support features into a connected library, which we named libspmt. This library is intended to be VM and language-agnostic, and forms a base for future work on integration with new systems, including different virtual machines, interpreters, ahead-of-time compilers, just-in-time compilers, and non-Java source languages. The combination of SableSpMT plus libspmt is quite flexible, in that the various components can be swapped out or used in isolation. For example, in related work we used our framework for a non-speculative dynamic purity analysis [XPV07]. The library support for return value prediction that we describe in Chapter 3 is generic and could similarly be used easily in a non-speculative context. A final example is the dependence buffering component that we use; it would again be quite practical to experiment with various transactional memory libraries in place of it. A full description of libspmt can be found in [PVK07]; with respect to this thesis it served mostly as an intermediate engineering step between the initial SableSpMT implementation and later optimizations.

A large part of our contribution with SableSpMT and libspmt is providing a common open source platform for future experimentation with software MLS and SpMT in general. SpMT has been investigated through many hardware proposals and simulations and a smaller but not insignificant number of software implementations. Each of these offers its own analysis of various implementation and optimization techniques. However, it can be difficult to evaluate these proposals with respect to and in combination with each other,

as there are multiple source languages, thread partitioning schemes, SpMT compilers, and hardware simulators being used. Even if these variables remain fixed, it is highly unlikely that an identical software architecture or set of simulation parameters will be used. Our focus on flexibility and analysis is intended to facilitate these kinds of comparisons: there are many parameters for runtime configuration in our system, it is straightforward to add new ones, and the same data collection framework provides results, whether the system changes are as small as adjusting a single integer parameter for fine-grained control or as large as changing the client of the library from SableSpMT to another system.

1.3.4 Return Value Prediction

When speculating on a non-void method it is necessary to predict the return value if it will later be consumed speculatively. Mispredictions of consumed return values generate dependence violations, and so lead to failed speculation. Thus, in a system that allows speculating on non-void methods, good support for *return value prediction* (RVP) increases speculation success rates. In general, RVP is best described as a runtime technique for predicting the results of non-void function, method, or procedure calls. In the case of speculative optimizations, it is useful to know the predicted value before the call returns, but there are other applications of RVP such as program understanding where execution time is not a factor. RVP is a specific kind of value prediction, its unique features with respect to predicting the results of arbitrary value-generating instructions being that methods may take arguments, that methods provide the core building block of modularity and thus exhibit an extremely broad range of behaviour, and that method calls occur relatively infrequently and so resources available for RVP are less constrained. Generalized value prediction is of course beneficial to all speculative systems, but in this work we focus on RVP as an MLS-specific optimization, noting that our design is certainly adaptable to speculative load prediction. The impact of RVP on MLS is demonstrated in Chapter 2, followed by a detailed account of our RVP design, implementation, and analysis in Chapter 3.

Consider the example with many complex uses of return values in Figure 1.2. Here the method `foo` returns a result into `r` with type `MyObject`. Following some delay, `r` is stored into another variable; tested for nullness to influence control flow and then if non-

```
MyObject r; // variable r with reference type
r = foo (a, b, c); // predict return value if speculating on foo
... // delay before uses of r
s = r; // stores
if (r != null) { // control flow
  r.f = r.g; // field access
  r.send (m); // virtual dispatch
}
use (r); // arguments
return r; // return values
```

Figure 1.2: *Motivating example for return value prediction.*

null, dereferenced to read from and write to its fields and be used as the receiver object for a virtual call; passed as a parameter; and finally returned from the method. Any misuse of r will cause speculation to fail. Of course there are other ways to use values, with the example serving primarily to illustrate the variety; for non-reference types arithmetic computations are an important class of use. Any time such a use is desired before the call returns, predicting the return value is important. And even without such a desired use, prediction data can provide valuable profiling information.

In our basic exploration of RVP we investigate a variety of predictors, both fixed-size and table-based. We experiment with every kind of value predictor that appeared suitable for return value prediction. We investigate a new memoization predictor that hashes together function arguments to retrieve a prediction from a hashtable. This use of memoization differs in that the result can be incorrect, widening its applicability to all methods instead of only pure ones. We include all of these predictors in a hybrid predictor that uses many different subpredictors and selects the best performing one to make a prediction. A unique aspect of our software design is that predictors are associated with individual callsites. This means that RVP costs scale with program size and that callsites do not interfere with each other. Overall we find high prediction accuracy and that memoization is a natural and useful technique for RVP.

After MLS system profiling, we found that the RVP execution overhead was so high as to preclude speculative performance. On this basis we designed a hybrid predictor that dynamically adapts by freeing the memory and computation resources associated with unused sub-predictors after an initial warmup period. This predictor maintains high accuracy while reducing time and memory costs. As part of our refactoring work to create this new hybrid, we wrote simplified and independent software versions of hardware circuits to replace our initial highly optimized and enmeshed predictors. This reworking included basic predictor logic as well as hash functions and hashtables. Our design led to an abstract unification framework for classifying and relating return value predictors. This framework facilitates understanding and can also be used to combine individual features of existing predictors to synthesize new ones. Our framework is easily extended to support new predictors, and can also be used as an independent RVP library without SpMT.

Our software RVP design differs from hardware designs in several important ways. First, we exploit the extra memory resources available in software to ensure a higher prediction accuracy. Second, we use a wide range of predictors, since new predictor logic is essentially free. However, the cost of executing many predictors in succession is expensive; whereas in hardware predictors can be effectively parallelized, in software they are serialized. Third, the dynamic reconfigurability displayed by our final adaptive hybrid, which in essence relies on a strategy design pattern, is particular to our software context; in existing hardware designs, the memory and hardware circuit costs are essentially fixed. This reconfigurability of the software context is also what permits instantiating new predictors for each callsite in the program and specializing callsites independently. Finally, we use high level VM information to facilitate understanding, analysis, and implementation. In particular, we show how the best predictor for a given callsite often reveals some interesting aspect of localized behaviour, and how whole-system predictor performance often correlates with overall program behaviour. We also explore the predictability differences between Java's primitive and reference types, and further exploit type information to reduce memory usage.

1.3.5 Nested Speculation

The speculation model presented in Figure 1.1 only depicts the execution of a single child. However, this view of MLS ignores two important questions. First, what happens when a parent thread creates a child, executes the target method, and encounters another method invocation? If it can create a second child, before joining the first, then this is *out-of-order nesting*, because the second child is first in sequential program order, and the first child is second. Second, what happens when a child thread encounters a method invocation? If it can create a child of its own, then this is *in-order nesting*. Complete support for these two kinds of nesting is necessary to expose all of the method level parallelism available in programs.

```
void a() {
  b(); // can parent create child 2 here?
  X; // child 2 might begin execution here
}

void main() {
  a(); // parent creates child 1 here
  Y; // child 1 begins execution here
  c(); // can child 1 create child 3 here?
  Z; // child 3 might begin execution here
}
```

Figure 1.3: *Motivating example for nested speculation.*

Consider the example of nested MLS in Figure 1.3, in which we assume threads are created as soon as possible. If the speculation model prohibits nesting and only allows one child per parent thread at a time, then the parent executes `b();X;` while child 1 executes `Y;c();Z;`. If the model allows out-of-order nesting, under which a parent can have more than one child at once, then the parent executes `b();`, child 2 executes `X;`, and child 1 executes `Y;c();Z;`. If instead in-order nesting is allowed, under which children can

create children of their own, then the parent executes `b();X;`, child 1 executes `Y;c();`, and child 3 executes `Z;`. If both in-order and out-of-order nesting are permitted, then `b();`, `X;`, `Y;c();`, and `Z` can all execute in parallel. The precise nature of the resulting parallel behaviour at runtime is not intuitively obvious, and depends on the interaction between source code, MLS system design, and underlying thread scheduling.

In our initial system we allowed for unlimited out-of-order nesting with many children per parent thread, but no in-order nesting. Results obtained using this nesting model are presented in Chapter 2. Profiling revealed that processors were largely idle, and so we developed support for in-order nesting as well to expose more parallelism. This led to a speculative stack data type and accompanying algorithms for manipulating it and forking and joining threads. In Chapter 4 we present an abstract stack-based model of nested speculation drawn directly from our practical implementation experience. We consider all possible runtime stack configurations at fork points and group them into nesting models with varying degrees of flexibility. This is the first comprehensive semantics for nested MLS. We also address the issue of how to depict speculative threads under MLS by presenting a stack-based form that reads straightforwardly, mirrors the machine state at runtime, and scales linearly on a 2D page as more threads are added.

There are also two specific memory management problems that arise for nested speculation. First, speculative threads require runtime data structures for an execution context, and these must be allocated quickly to ensure efficient speculative thread creation. A simple solution is to allocate a fixed number of threads per processor, but a more general solution must allow for an arbitrary number of threads on a finite number of processors. We solve this problem by recycling the entire aggregate thread data structure at once. Second, under in-order nesting memory can be allocated in one thread but freed in another. The simplest example is when child thread C1 allocates its own child C2, but then later C1's parent P joins both C1 and C2, which results in P freeing C2's memory which it did not allocate. We solve this problem by using per-processor freelists and migrating blocks of threads at once. Thus we contribute a custom, lightweight, single-purpose memory manager that efficiently recycles entire aggregate data structures at once on multiprocessor machines, also described in Chapter 4. This memory manager in turn supports arbitrarily nested MLS.

1.3.6 Fork Heuristics

Given the ability to fork a thread at any callsite and efficiently predict return values, the final question is where to actually fork threads. Since the answer can only be approximated, we refer to the broad class of thread decomposition or program partitioning techniques as *fork heuristics*. At a coarse granularity, the use of MLS is a heuristic choice itself, based on the assumption that method boundaries are appropriate delimiters of speculative parallelism; then within MLS, there are medium-grained decisions to be made, such as which callsites to speculate at; and then given a set of fork points, there are fine-grained decisions, such as how often to speculate or under what performance conditions.

```
void a() {
  X;
}

void b() {
  Y;
}

main () {
  a(); // speculation success rate = M
  b(); // speculation success rate = N
  Z;
}
```

Figure 1.4: *Motivating example for fork heuristics.*

Two primary issues in constructing fork heuristics are thread lengths and speculation success rates. In an ideal situation, threads would only be forked if there was a high probability of success, if the parent and child threads executed long enough to outweigh overhead costs, and if the load was fairly balanced between child and parent. Consider the example in Figure 1.4. Here, a thread could be forked at `a()`, `b()`, or both. In the absence of spec-

ulation, and ignoring function call costs, the parent has length $X + Y + Z$. If a child is forked at `a()` but not `b()` then the parent has length X whereas the child's length is anywhere between 0 and $Y + Z$, depending on how long X takes. If a child is forked at `b()` but not `a()` then the parent has length $X + Y$ and the child has length between 0 and Z, depending on how long $X + Y$ takes. If a child forked at `a()` creates another child at `b()`, then the parent has length X, child 1 has length between 0 and Y, and child 2 has length between 0 and Z. With respect to success rates, if children are forked by a parent thread at `a()` or `b()`, then they have probability M and N of succeeding respectively. However, if a child forked at `a()` forks another child at `b()`, then the probability of success for the child forked at `b()` is $M \times N$ because a speculative dependence is created, and the first thread must succeed in order for the second to.

Tracking all of this data quickly becomes complicated. Not only is thread length and success rate contingent upon which other threads have been created at fork time, but the decision *not* to fork a thread can affect the lengths and success rates of other speculative threads. In an ideal model, thread lengths and success rates would be tracked according to precise context. However, for our first approach, we simply consider every callsite to be a possible fork point and use online profiling and feedback-driven heuristics to filter out the unprofitable ones, as described in Chapter 2. This is a completely dynamic model. Broad criteria include expected probability of speculation success, return value predictability, the number of previous speculation attempts, current speculation nesting depth and height, and expected lengths of parent and child threads.

We found based on profiling that this approach yields thread lengths that are on average too short, that there are a large number of factors that influence speculation, and that we need more insight into runtime behaviour to support the design of better fork heuristics. We decided on this basis to study the structural features of input programs that determine their suitability for speculation, as described in Chapter 4. We considered a variety of common programming idioms in abstract forms, and exhaustively explored the relationship between speculation decisions and parallel behaviour. The result is a pattern language for method level speculation that draws on our nesting models as well as existing examples of method level parallelism found in manually parallelized benchmarks. Many of our conclusions apply equally well to non-speculative method level parallelization techniques.

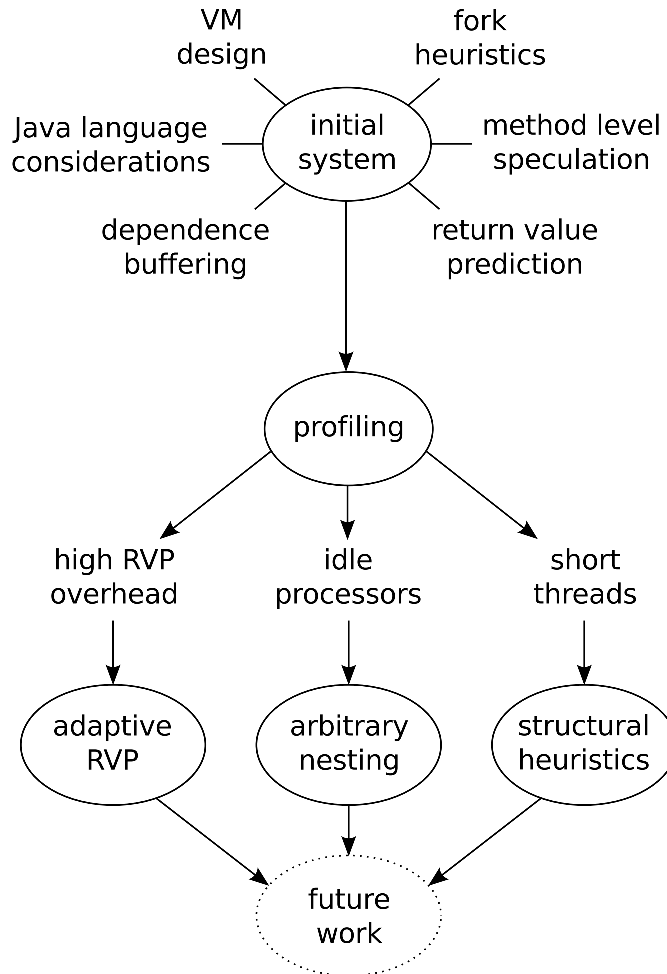


Figure 1.5: *Research process overview.*

1.4 Roadmap

At a broad level, this thesis is structured around our research process, as outlined in Figure 1.5. First, we built an initial SpMT prototype system, SableSpMT. This required several components: 1) basic support for method level speculation, which creates child threads at callsites and joins them when the parent invocation returns; 2) return value prediction, which allows children forked at non-void callsites to proceed past consumption of the return value, reducing misspeculations; 3) dependence buffering, which allows for specula-

tive children to read from and write to the Java heap in a strongly isolated fashion; 4) fork heuristics, for deciding where and with what priority to create child threads; 5) a VM-based design that allows for dynamic switching between non-speculative and speculative code; and 6) support for many Java language safety considerations, including garbage collection, object allocation, native methods, exception handling, bytecode interpretation, dynamic class loading, and the Java memory model. Although this initial prototype system demonstrated speedup in a relative sense, it slowed down in an absolute sense due to overhead costs. We then profiled the system to identify performance bottlenecks. This profiling revealed three things: 1) naïve software return value prediction has high overhead costs; 2) that spare processors in the system were mostly idle; 3) that committed threads were usually quite short. We then identified corresponding optimizations to address these bottlenecks: 1) online adaptive return value prediction that dynamically specializes on a per-callsite basis, reducing unnecessary computation; 2) arbitrarily nested method level speculation, which allows for flexible thread creation by both speculative and non-speculative threads, in turn providing processors with extra work; and 3) improved fork heuristics based on a structural approach to eliminating the creation of short threads. These optimizations in turn lay the foundation for future work on making method level speculation feasible.

The remainder of this thesis is organized as follows. In Chapter 2 we present the design, implementation, and experimental evaluation of a software method level speculation system for Java. In Chapter 3 we explore software return value prediction in detail, optimizing for speed, memory consumption, and predictor accuracy. In Chapter 4 we describe a stack model for nested method level speculation, and then use it to analyse the speculative runtime behaviour of a range of common programming idioms and derive a set of structural fork heuristics. Chapter 5 we survey related work on speculative parallelization. Finally, in Chapter 6 we discuss conclusions and future work.

Chapter 2
Software Method Level Speculation for Java

Speculative multithreading has shown great promise as a strategy for fine to medium grain automatic parallelization. In a hardware context, techniques to ensure correct SpMT behaviour and significant performance gains are now well established. However, hardware is expensive to produce, and software alternatives are desirable. Further, data acquisition from and analysis of such systems is difficult and complex, typically being limited to a specific hardware design and simulation environment. For their part, software and virtual machine SpMT designs require adherence to high level language semantics and their performance is limited by increased overhead. These factors can impose many additional constraints on SpMT behaviour, as well as open up new opportunities to exploit both language-specific information and software plasticity.

In this chapter we describe SableSpMT, our research SpMT framework based on method level speculation. We present a detailed design for this Java-specific, software MLS system that operates at the bytecode level and fully addresses the problems and requirements imposed by the Java language and VM environment. We demonstrate its use as a research framework by including extensive analysis information, including data gathered from the return value prediction component, results from the integration of static analyses, an analysis of speculation overhead, parallelism analysis, dynamic analysis based on runtime profiling, runtime speculation behaviour analysis, and speedup analysis.

Our results provide a comprehensive survey of the corresponding costs and benefits of software MLS for Java. We find that exceptions, GC, and dynamic class loading have

only a small impact, but that concurrency, native methods, and memory model concerns do play an important role, as does an appropriate, language-specific runtime SpMT support system. Profiling results further indicate that return value prediction performance, support for in-order nested speculation, and improved fork heuristics are areas for future work. Our experience indicates that full consideration of language and execution semantics is critical to correct and efficient execution of high level SpMT designs; our work here provides a baseline for future software implementations with features as complex as those found in Java and the Java virtual machine.

2.1 Introduction

SpMT and MLS have been investigated through many hardware proposals and simulations, and a smaller but not insignificant number of software designs, each offering its own analysis of various implementation and optimization techniques. However, it is difficult to evaluate these proposals with respect to and in combination with each other, as there are multiple source languages, thread partitioning schemes, SpMT compilers, and hardware simulators being used. Even if these variables remain fixed, it is highly unlikely that an identical software architecture and/or set of simulation parameters will be used. Furthermore, as a hardware problem, the issues of ensuring correctness under speculative execution have been well defined, such that different rollback or synchronization approaches are sufficient to guarantee overall correct program behaviour. Software approaches to SpMT, however, need to take into account the full source language semantics and behaviour to ensure correct and efficient execution, and in general this is not trivially ensured by low level hardware mechanisms.

We present SableSpMT as a common framework and solution to these problems, as an extension of the SableVM Java virtual machine [Gag02]. SableSpMT provides a convenient hardware abstraction layer by operating at the bytecode instruction level, takes the full Java language and VM specification into account, including all bytecode instructions, object allocation, garbage collection, synchronization, exceptions, native methods, dynamic class loading, and the Java memory model, supports static analysis through the Soot bytecode compiler framework [VR00] and parsing of Java classfile attributes [PQVR+01], and

works on existing multiprocessor systems. SableSpMT further provides a full set of SpMT support features, including thread forking and joining based on method level speculation, dependence buffering, stack buffering, priority queueing, and return value prediction. Our framework is designed to facilitate SpMT research, and includes a unique debugging mode, logging, and portability amongst the features that make it appropriate for experimentation and new designs.

We report on both Java benchmark and framework behaviour to illustrate the forms of experimental and design analysis we support, and to understand the behaviour of our system. Through dynamic measurements we show that while speculative coverage, or the percentage of sequential program execution that occurs successfully in parallel, can be quite high in Java programs, the overhead costs are significant enough in our initial implementation to preclude actual speedup. However, we are able to perform experiments to determine upper bounds on speedup in the absence of all overhead. Furthermore, our execution times are still faster than those offered by hardware simulators providing similar functionality [KT98].

At a finer level of detail, we also break down the SpMT overhead costs to determine performance bottlenecks and set optimization goals. In our case overhead is dominated by verification of speculative threads and the concomitant interprocessor memory traffic, lock and barrier synchronization, and update costs for return value prediction. We also find further opportunities suggested by short thread lengths and a lack of available threads to execute under our precise speculation model, out-of-order MLS nesting. With regards to RVP, results gathered within our framework extend previous studies to include more realistic benchmark runs, offer further data on the relative benefits, requirements and costs of various prediction strategies, and expose the potential benefits of exploiting both static and runtime feedback optimization information. Finally, language and VM level speculation also produce design constraints due to efficiency concerns; for instance, Java programs tend to have frequent heap accesses, object allocations, and method calls. Our runtime SpMT support system accomodates this behaviour, and we evaluate the relative importance of dependence buffering, stack buffering, return value prediction, speculative allocation, and priority queueing.

Hardware simulations have already demonstrated the great potential in speculative mul-

tithreading. We contend that the same techniques, however, can be investigated more generally and efficiently at the virtual machine level using commodity multiprocessor hardware, given an appropriate analysis framework. Virtual machines allow for exploration of complex design changes, facilitate detailed instrumentation, provide high level information that is not generally available to hardware approaches, and are able to interact directly with the underlying architecture. Our work is intended to enable SpMT investigations by providing an execution and analysis environment as well as real data from a working implementation. In addition to using SableSpMT to characterize both thread parallelism and overhead under software speculation, our work here aims to provide a thorough Java SpMT design and implementation suitable for future work and an understanding of the requirements and relative impact of high level language semantics.

2.1.1 Contributions

We make the following specific contributions:

- We describe SableSpMT, a complete implementation of MLS-based SpMT for Java that runs on real multiprocessor hardware, and present its suitability as an analysis framework. This is the first complete such work within a virtual machine. We include descriptions of all major VM changes necessary, including bytecode modifications, novel SpMT runtime support components, and the handling of Java language features.

- We simplify the implementation and analysis of new SpMT designs by providing a deterministic, single-threaded uniprocessor mode as well as logging facilities and statistics gathering.

- We demonstrate that high level analysis information can be easily exploited by our framework. Ahead-of-time results computed by Soot as well as runtime profiling-based feedback can passed to our execution engine to improve performance, and we illustrate the technique using our work on return value prediction.

- We provide detailed data on the speculative execution of SPEC JVM98 at size 100,

a suite of non-trivial benchmark programs. These data include a breakdown of overhead costs, the impact of highly accurate RVP, measurements of dynamic parallelism, the impact of Java language features and MLS support components, and overall running times.

In Section 2.2 we give an overview of how our framework is constructed and its main features. This includes an exposition of the components required for software MLS for Java, our multithreaded execution and single-threaded debugging modes, system configuration options, and data logging and trace generation features. We then turn to the details of our Java MLS design. In Section 2.3 we describe how to prepare method bodies for speculative execution, in Section 2.4 we survey our speculative runtime support components, and in Section 2.5 we discuss the intricacies of the Java language and their interaction with speculation. In Section 2.6 we analyse actual data, demonstrating the flexibility of our system in terms of data gathering and providing a wide variety of observations. Finally, we conclude and discuss future work in Section 2.7. Related work specific to software MLS for Java as well as alternative approaches is discussed in Chapter 5.

2.2 Framework

We begin with an overview of our framework, followed by a brief exposition of our multithreaded speculative execution model. Then we present some of the features of our framework that help with the implementation, debugging, and analysis of such a complex undertaking, namely a single-threaded execution mode, system configuration options, and support for logging and trace generation.

2.2.1 Overview

An overview of the SableSpMT analysis framework and Java SpMT execution environment is shown in Figure 2.1. SableSpMT is an extension of the switch-threaded bytecode interpreter in SableVM [Gag02], an open source software Java virtual machine. SableVM adheres to the JVM Specification [LY99], and is capable of running Eclipse and other

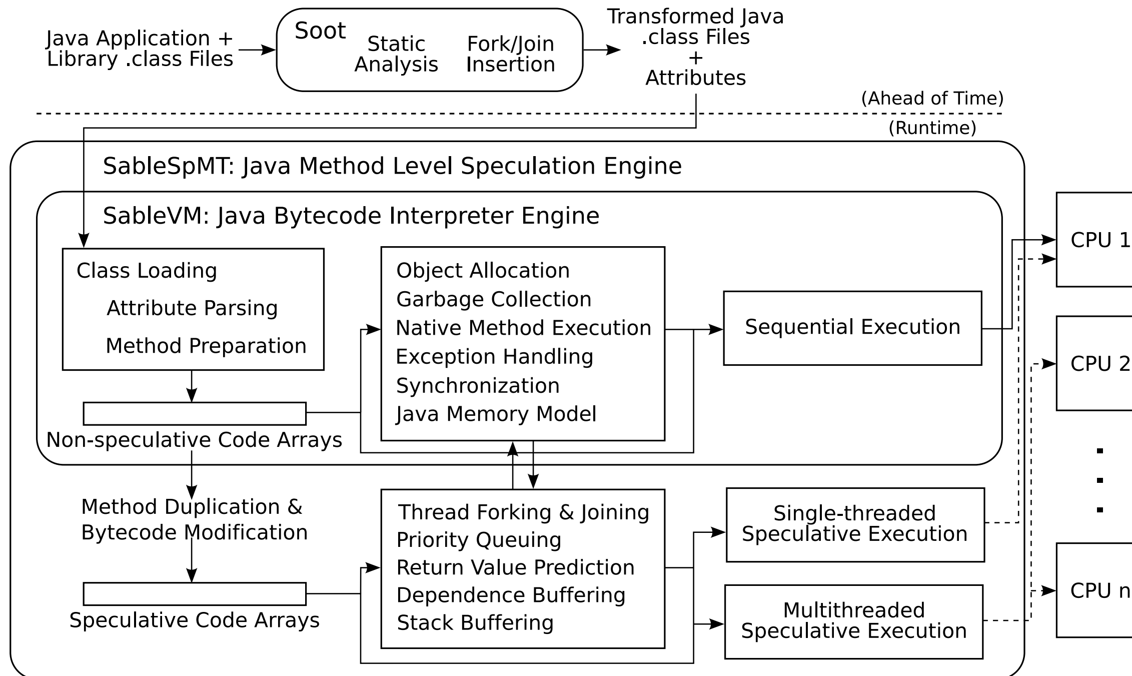


Figure 2.1: *The SableSpMT method level speculation execution environment.*

large, complex programs. The implementation is completely POSIX-compliant and written in ANSI C. SableVM ports exist for 13 different architectures, such that porting the SableSpMT engine to new architectures should relatively straightforward; currently it runs on multicore or multiprocessor `x86_64` machines. Most of the porting complexity derives from defining the right atomic operations in assembly language.

Soot [VR00] is used to transform, analyse, and attach attributes to Java `.class` files in an ahead-of-time step [PQVR+01], although this could also occur at runtime. SableSpMT reads in these classes during class loading, parsing attributes and preparing method bodies. These method bodies are implemented internally as *code arrays*, contiguous sequences of word-sized instructions and instruction operands derived from Java bytecode. SableVM already creates normal non-speculative code arrays at runtime. Rather than include conditional checks in many non-speculative instructions, we chose to have SableSpMT duplicate and modify the entire non-speculative code array at method preparation time to create an exclusively speculative one. Sequential execution depends only the non-speculative code

arrays, and interacts with normal JVM support components. Speculative execution causes SableSpMT to fork and join child threads at runtime, and these depend on the speculative code arrays for safe out-of-order execution. The matching `pc` offsets of instructions in these code arrays allows for straightforward switches between non-speculative and speculative execution. Two execution modes are provided, a single-threaded "simulation" mode and a true multithreaded mode. The single-threaded mode alternates between non-speculative and speculative execution in a single thread, whereas the multithreaded mode splits single Java threads across multiple cores or processors.

Various SpMT runtime support facilities are needed, including priority queueing, return value prediction, dependence buffering, and stack buffering. SableSpMT also interacts with SableVM's own runtime support components, including a semi-space copying garbage collector, object allocation, native method execution, exception handling, synchronization, and the Java memory model. Outside of thread forking and joining, speculation has negligible impact on and is largely invisible to normal multithreaded VM execution. Specifically, it uses what we define as *zero-sum speculative threading*, such that $s = \max(n - p, 0)$ speculative threads run only on free processors, where n is the number of processors and p is the number of non-sleeping non-speculative parent Java threads.

2.2.2 Multithreaded Execution Mode

Many components are needed for MLS to work properly in a JVM, the full details of which are given in Sections 2.3, 2.4, and 2.5. A high level view of the multithreaded mode involving multiple threads and method calls that brings together all of these components is shown in Figure 2.2; the finer details of the execution of a speculative child can be found in the depiction of our single-threaded simulation mode in Figure 2.3. A *dependence buffer* protects main memory from out-of-order and possibly invalid speculative operations, and some form of *stack buffering* is necessary to give child threads a protected execution context. New `SPMT_FORK` and `SPMT_JOIN` instructions surround every callsite; the fork instruction enqueues child threads onto an $O(1)$ priority queue, which are dequeued and executed on separate processors by MLS *helper threads*, and the join instruction stops and validates children, either committing or aborting them. In the figure, children C1, C2,

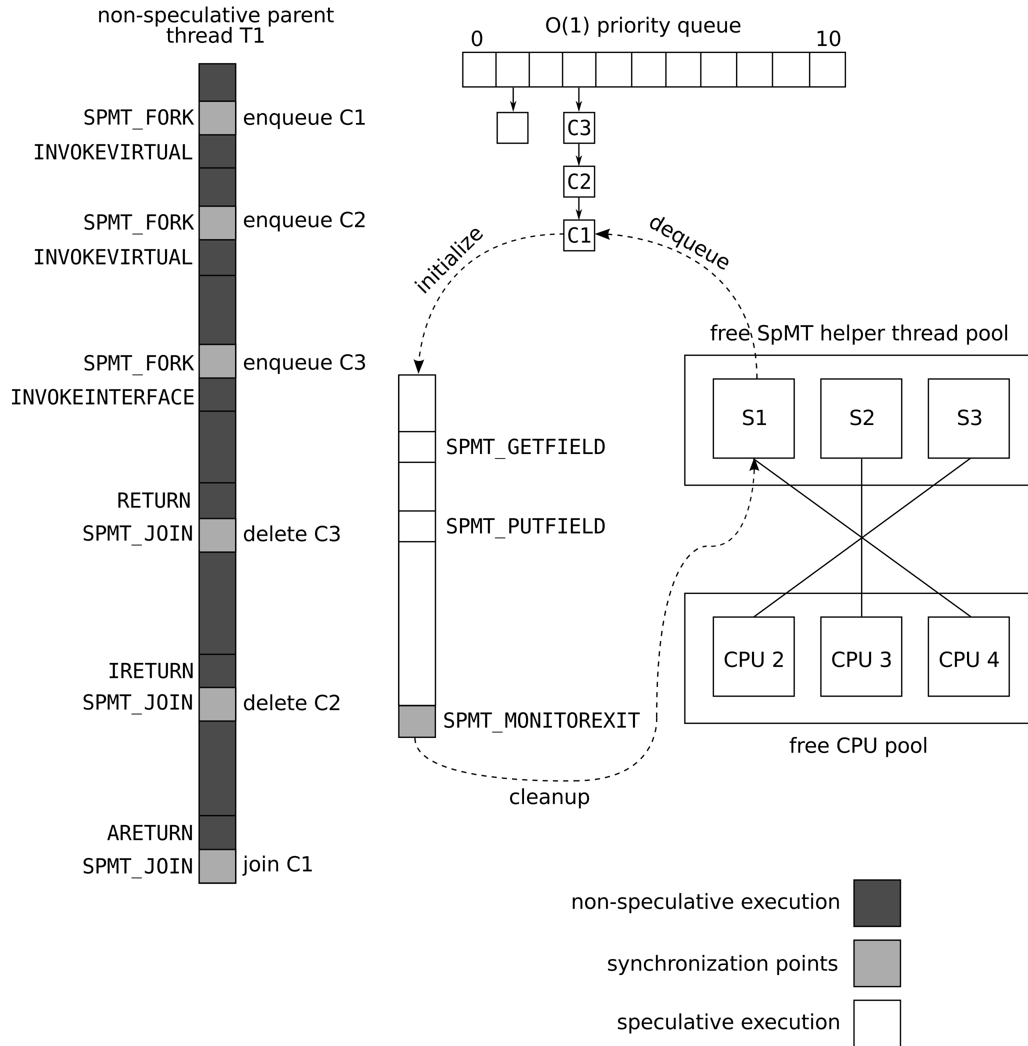


Figure 2.2: *Multithreaded execution mode.*

and C3 are enqueued, but only C1 is executed and joined, meaning that C2 and C3 are deleted from the queue. ARETURN returns a reference and IRETURN returns an integer, which means that C1 and C2 need some kind of return value prediction to execute safely. On the other hand, RETURN is used for void methods, and so C3 does not need a predicted value. While executing speculative code, we need *modified bytecode instructions* to protect against unsafe control flow; for example, GETFIELD is modified to read from a dependence buffer, and MONITOREXIT causes speculation to come to an abrupt halt, although it does

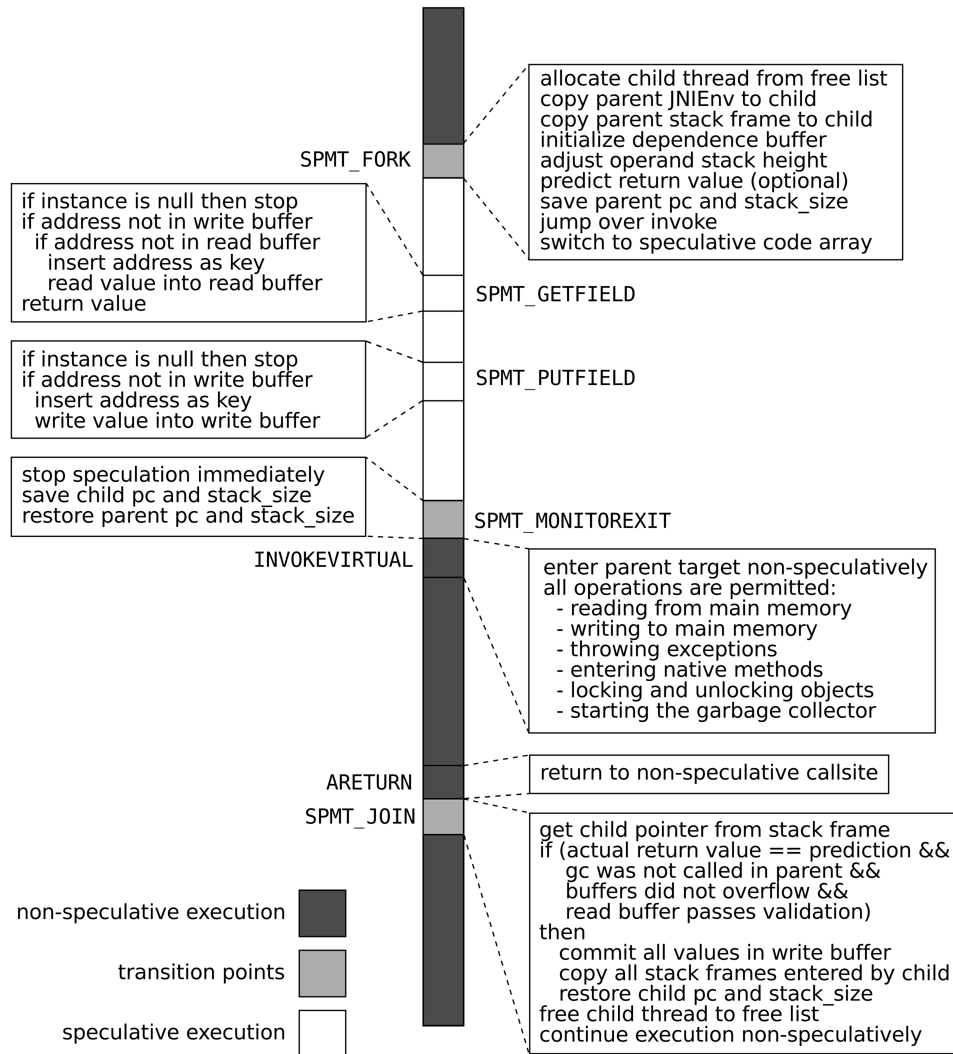


Figure 2.3: *Single-threaded execution mode.*

not automatically force abortion. Finally, we need to make sure speculation interacts safely with exception handling, object allocation, garbage collection, native method execution, synchronization, class loading, and the Java memory model.

We make several different optimizations to these components in SableSpMT, some of the more notable ones being aggressive return value prediction [PV04a, PV04b], improvements to the dependence buffer, allowing for speculative threads to enter and exit methods, better enqueueing algorithms, speculative object allocation, and reduction of interproces-

sor memory traffic. Most of the techniques we have encountered in the literature can be implemented within our framework; in Section 2.6 we illustrate typical data gathering and analysis using our work on return value prediction and the speculative engine itself as examples.

2.2.3 Single-Threaded Execution Mode

One of the unique features of our design is a single-threaded simulation mode that mimics the process of speculative execution in a single thread. Early on in the development of SableSpMT, we found ourselves wanting some way to test the components we had written in the context of an executing JVM, without introducing the complexity of actual concurrency into our debugging process. The resulting deterministic design is shown in Figure 2.3. In this mode a single thread of Java execution follows the complete speculative control flow. Upon reaching a fork point, the method call is skipped, and the ensuing continuation code is executed speculatively; when a terminating condition is reached, the same thread jumps back to the non-speculative execution of the method call, and upon returning from the call, it attempts to join with its own speculative result.

There are three primary advantages to having this single-threaded simulation mode. First, it allows for testing of SpMT components in an incomplete system, most importantly one without multiprocessor support. It does so by providing state saving and restoral, and interleaving the execution of speculative and non-speculative code. Second, by not running multiple threads it prevents race conditions, deadlocks, and memory traffic from interfering with development, helping to minimize the search space when faced with debugging. We were able to alternate coding with designing support for MLS according to the full JVM Specification, and only after we had completed a requirements analysis in this manner did we develop the multithreaded execution mode. Third, it means we have the foundations for Java checkpointing and rollback within a virtual machine. This has utility for Java outside of SpMT, for example in traditional debugging [Coo02], database transactions such as `java.sql.Connection`, formal verification [Eug03], fault-tolerance [FK03], and software transactional memory [LR06].

2.2.4 System Configuration

In our framework, system properties specified on the command line are used to select different SpMT algorithms and data structures, which facilitates experimental analysis by eliminating the need for multiple VM builds. As changes to SableSpMT are introduced, rather than outright replace old control flow or adjust constants to optimal values, system properties are used wherever possible, and thus it is straightforward to make controlled comparisons with old configurations and revert if necessary. In finalized builds, these properties can be automatically converted to constants via preprocessor directives and a single Autoconf `configure` option, so that the added runtime overhead of conditionals testing them will be optimized away. There are over 50 such properties in SableSpMT, controlling everything from maximum RVP hashtable sizes to the number of executing MLS helper threads, and it is easy to introduce new ones. The only other significant compile-time `configure` options in SableSpMT allow the user to 1) enable MLS in the first place, 2) enable debugging and assertions, and 3) enable statistics gathering for post-execution analysis.

2.2.5 Logging and Trace Generation

Finally, SableSpMT provides a comprehensive logging and trace generation system that can present Java SpMT events by themselves, or interleave them with existing execution traces of class loading, method invocation, garbage collection, synchronization, and bytecode execution. An example trace with interleaved method invocation, bytecode, and SpMT events is shown in Figure 2.4. Here a speculative child executes three instructions of the continuation past a non-speculative call to `Object.<init>` before being successfully committed. These traces are primarily useful for debugging purposes when implementing new techniques. SableVM supports only the JVMDI and JDWP for integration with debuggers at this time, and although we do not provide trace compression or an implementation and extension of the related JVMPI or JVMTI profiling interfaces, these facilities could be incorporated to permit detailed analysis of SpMT execution traces, using a dynamic metrics tool such as *J [Duf04].

Thread	pthread	type	code address	bytecode instruction or internal event
T1	P16384	N	@0x2a976bad68	ALOAD_0
T1	P16384	N	@0x2a976bad70	SPMT_FORK
T1	P16384	N	<internal>	enqueue spmt child @0x5ed850
T1	P16384	N	@0x2a976bad88	INVOKESPECIAL
T1	P49156	S	<internal>	dequeue spmt child @0x5ed850
T1	P49156	S	<internal>	start spmt
T1	P16384	N	<internal>	entering java/lang/Object.<init>()V
T1	P49156	S	@0x2a976baf38	ALOAD_0
T1	P16384	N	@0x2a976baf90	RETURN
T1	P49156	S	@0x2a976baf40	ALOAD_1
T1	P16384	N	<internal>	exiting java/lang/Object.<init>()V
T1	P16384	N	@0x2a976badb0	SPMT_JOIN
T1	P49156	S	@0x2a976baf48	SPMT_PUTFIELD
T1	P16384	N	<internal>	signalling spmt thread halt @0x5ed850
T1	P49156	S	<internal>	stop spmt - signalled by parent
T1	P16384	N	<internal>	spmt passed @0x5ed850

Figure 2.4: *SpMT execution trace.* Type N means non-speculative and type S means speculative.

2.3 Speculative Method Preparation

Before speculative execution can begin, method bodies must be *prepared* for MLS. This process entails parsing classfile attributes for static analysis info, inserting fork and join points, and modifying bytecode instructions. The actual generation of a parallel speculative code array occurs when a given method is invoked for the first time. Once primed for speculation, a child thread can be forked at any callsite within the method body. Furthermore, speculation can continue across method boundaries as long as the methods being invoked or returned to have been similarly prepared.

2.3.1 Static Analysis and Attribute Parsing

An advantage to language level SpMT is the ability to use high level program information. In our case, we use the Soot bytecode compiler framework [VR00], a convenient tool

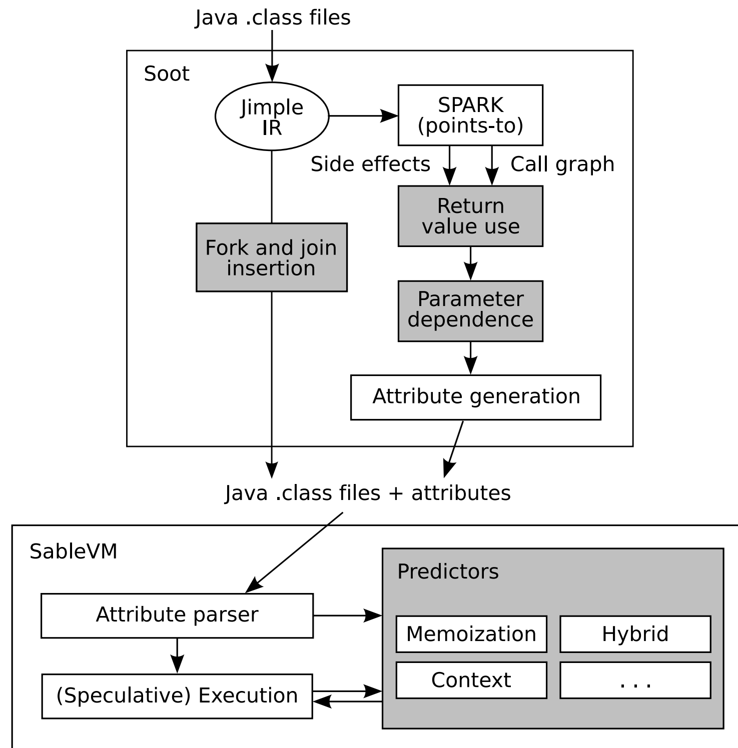


Figure 2.5: *Static analysis integration.*

for ahead-of-time static analysis and transformation in the absence of the runtime static analysis support typically found in JIT compilers. In Figure 2.5 we show the use of Soot to transform the base input Java classfiles in order to insert `SPMT_FORK` and `SPMT_JOIN` instructions. The same process can also be used to append static analysis information as classfile attributes [PQVR+01], which are then interpreted by the SpMT engine during class loading. We use attributes to encode the results of two analyses for improved RVP using Soot [PV04a]. During method preparation, the analysis data are associated with callsites for use by the RVP component; a summary of results is given in Section 2.6.2.

2.3.2 Fork and Join Insertion

The SableSpMT engine needs the ability to fork and join child threads. The `SPMT_FORK` and `SPMT_JOIN` instructions provide this functionality. Under MLS threads are forked

and joined immediately before and after method invocations, and so these instructions are inserted around every INVOKE<X> instruction. This design is callsite-oriented; a target-oriented design would insert the fork and join instructions at method entry and exit instead.

The actual insertion by Soot involves placing calls to dummy static void Spmt.fork and Spmt.join methods around every callsite, and then during runtime method preparation replacing these with the appropriate SPMT_FORK and SPMT_JOIN instructions. This approach has several advantages: first, transformed classfiles will run in the absence of SpMT support, the dummy methods being trivially inlined; second, integration with a static analysis to determine good fork points is facilitated; and third, bytecode offsets are automatically adjusted.

2.3.3 Bytecode Instruction Modification

The majority of Java's 201 bytecode instructions can be used verbatim for speculative execution; however, roughly 25% need modification to protect against potentially dangerous behaviours, as shown in Table 2.1. If these instructions were modified in place, the overhead of extra runtime conditionals would impact on the speed of non-speculative execution. Instead, modification takes place in a duplicate copy of the code array created especially for speculative execution. Indeed, the only significant change to non-speculative bytecode is the insertion of fork and join points. Problematic operations include:

- *Global memory access*. Reads from and writes to main memory require buffering, and so the <X>A(LOAD|STORE) and (GET|PUT)(FIELD|STATIC) instructions are modified to read and write their data using a dependence buffer, as described in Section 2.4. If final or volatile field access flags are set, these instructions may also require a memory barrier to correctly order memory accesses, as described in Section 2.5, in which case speculation must also stop.

- *Exceptions*. In unsafe situations, many instructions must throw exceptions to ensure the safety of bytecode execution, including (I|L)(DIV|REM) that throw an ArithmeticException upon division by zero, and others that may throw a NullPointerException, ArrayIndexOutOfBoundsException, or ClassCastExcep-

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads class(es)	orders memory	forces stop
GETFIELD	yes					maybe		once	maybe	maybe
GETSTATIC	yes							once	maybe	maybe
<X>ALOAD	yes					maybe				maybe
PUTFIELD		yes				maybe		once	maybe	maybe
PUTSTATIC		yes						once	maybe	maybe
<X>ASTORE		yes				maybe				maybe
(I|L)(DIV|REM)						maybe				maybe
ARRAYLENGTH						maybe				maybe
CHECKCAST						maybe		once		maybe
ATHROW						yes				yes
INSTANCEOF								once		maybe
RET										maybe
MONITORENTER	yes	yes	yes			maybe			yes	yes
MONITOREXIT	yes	yes		yes		maybe			yes	yes
INVOKE<X>	maybe	maybe	maybe			maybe	maybe	once	maybe	maybe
<X>RETURN	maybe	maybe		maybe		maybe	maybe	once	maybe	maybe
NEW		yes			yes	maybe		once		maybe
NEWARRAY		yes			yes	maybe				maybe
ANEWARRAY		yes			yes	maybe		once		maybe
MULTIANEWARRAY		yes			yes	maybe		once		maybe
LDC_STRING					once					once

Table 2.1: *Java bytecode instructions modified to support speculation.* Each instruction is marked according to its behaviours that require special attention during speculative execution. These behaviours are marked 'once', 'maybe', or 'yes' according to their probabilities of occurring within the instruction. 'Forces stop' indicates whether the instruction may force termination of a speculative child thread, but does not necessarily imply abortion and failure. Not shown are branch instructions; these are trivially fixed to support jumping to the right `pc`.

tion. Application or library code may also throw explicit exceptions using ATHROW. In both implicit and explicit cases, speculation rolls back to the beginning of the instruction and stops immediately; however, the decision to abort or commit is deferred until the parent joins the child. Exceptions must also be handled safely if thrown by non-speculative parent threads with speculative children, as discussed in Section 2.5.

- *Detecting object references.* The INSTANCEOF instruction computes type assignability between a pre-specified class and an object reference on the stack. Normally, bytecode verification promises that the stack value is always a valid reference to the start of an object instance on the heap, but speculative execution cannot depend on this guarantee. Accordingly, speculation must stop if the reference does not lie within heap bounds, or if it does not point to an object header. Currently we insert a magic word into all object headers, although a bitmap of heap words to object headers would be more accurate and space-efficient.

- *Subroutines.* JSR (jump to subroutine) is always safe to execute because the target address is hardcoded into the code array. However, the return address used by its partner RET is read from a local variable, and must point to a valid instruction. Furthermore, for a given subroutine, if the JSR occurs speculatively and the RET non-speculatively, or vice versa, the return address must be adjusted to use the right code array. Thus a modified *non-speculative* RET is also needed.

- *Synchronization.* The INVOKE<X> and <X>RETURN instructions may lock and unlock object monitors, and MONITOR(ENTER|EXIT) will always lock or unlock object monitors; they furthermore require memory barriers and are strongly ordering. These instructions are also marked as reading from and writing to global variables, as lockwords are stored in object headers. In our design, non-speculative threads that encounter a locked object monitor must block and cannot become speculative instead. Similarly, speculative threads that encounter lock or unlock operations are always forced to stop. We discuss related work on speculative locking in Section 5.7 and future work on speculative locking in Section 6.2.1.

- *Method entry*. Speculatively, INVOKE<X> are prevented from entering unprepared methods and triggering class loading and method preparation. Furthermore, at non-static callsites, the receiver is checked to be a valid object instance, the target is checked to have the right stack effect, and the type of the target's class is checked for assignability to the receiver's type. Invokes are also prevented from entering native code or attempting to execute abstract methods.

- *Method exit*. After the synchronization check, the <X>RETURN instructions require three additional safety operations: 1) potential buffering of the non-speculative stack frame from the parent thread, as described in Section 2.4; 2) verifying that the caller is not executing a *preparation sequence*, a special group of instructions used in SableVM to replace slow instructions with faster versions; and 3) ensuring that speculation does not leave bytecode execution entirely, which would mean Java thread death, VM death, or a return to native code.

- *Object allocation*. Barring an exception being thrown or GC being triggered, the NEW and ((MULTI|)A|)NEWARRAY instructions are safe to execute. The LDC_STRING specialization of LDC allocates a constant String object upon its first execution, the address of which is patched into both non-speculative and speculative code arrays, and forces speculation to stop only once. Allocation and GC are discussed in greater detail in Section 2.5.

2.3.4 Parallel Code Array Generation

The goal of this extensive bytecode modification is to prepare parallel code arrays for speculative execution, as shown in Figure 2.6. The non-speculative array is duplicated, branch targets are adjusted, and modified instructions replace ordinary non-speculative versions where necessary. Additionally, SPMT_FORK and SPMT_JOIN surround every INVOKE<X> in both code arrays, enabling both non-speculative and speculative threads to create and join children. Transitions between non-speculative and speculative execution are facilitated by identical instruction offsets in each array.

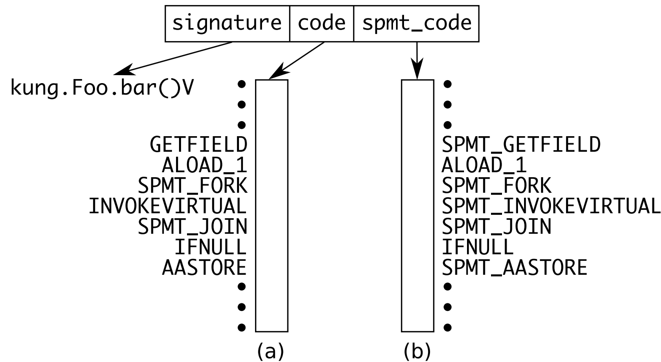


Figure 2.6: *Parallel code arrays.* (a) non-speculative code array prepared for method `bar`; (b) speculative version of the same code array with modified instructions.

2.4 Speculative Runtime Support

Following the preparation of method bodies for speculative execution, the speculation engine makes use of various runtime support components that interact with bytecode and allow for child thread forking, queueing, execution, and joining to take place while ensuring correct and efficient execution through appropriate return value prediction, dependence buffering, and stack buffering.

2.4.1 Thread Forking

Speculative child threads are forked by non-speculative parents and also by speculative children at `SPMT_FORK` instructions. Speculating at every fork point is not necessarily optimal, and in the context of MLS various heuristics for optimizing fork decisions have been investigated [WK05]. SableSpMT permits relatively arbitrary fork heuristics based on runtime profiling information; however, we limit ourselves to a simple "always fork" strategy in this chapter as a more generally useful baseline measurement. In Chapter 4 we consider structural fork heuristics for MLS.

Having made the decision to fork a child, several steps are required. First, those variables of the parent thread environment that can be accessed speculatively are copied to a child thread environment. The parent environment is a `JNIEnv` struct, and so each child

thread needs a `JNIEnv` struct of its own. In this fashion, the child assumes the identity of its parent. Second, a child stack buffer is initialized and the parent stack frame is copied to the child, giving it an execution context. Third, a dependence buffer is initialized; this protects main memory from speculative execution, and allows for child validation upon joining. Fourth, the operand stack height of the child is adjusted to account for the stack effect of the invoke following the fork point, and the `pc` of the child is set to the first instruction past the invoke. Fifth, a return value is predicted for non-void methods; technically, any arbitrary value can be used as a prediction, although the chance of speculation success is greatly reduced by doing so. Speculation in the child then begins, continuing until some stopping condition is reached: either unsafe control flow, a predefined sequence length limit, or the parent signalling the child from a join point. The complete join process is discussed in Section 2.4.6.

In the above steps, memory reuse is critical in reducing the overhead of thread environment, dependence buffer, and stack buffer allocation. We describe our child thread memory allocator fully in Section 4.2. Further, to reduce the forking overhead on non-speculative parent threads, the child is enqueued on the priority queue after the first step and the remaining steps occur in a separate helper thread after the child is removed from the queue for execution.

2.4.2 Priority Queueing

In the default multithreaded speculative execution mode, children are enqueued at fork points on a global $O(1)$ concurrent priority queue. As discussed, a minimal amount of initialization is done prior to enqueueing to limit the impact of fork overhead on non-speculative threads. Priorities 0–10 are computed as $\min(l \times r/1000, 10)$, where l is the average bytecode sequence length and r is the success rate; higher priority threads are those that are expected to do more useful work. l is computed as i_s/f, where f is the total number of speculative threads forked at the current callsite and i_s is the total number of speculative instructions executed by these threads. Similarly, r is computed as c/f, where c is the total number of successful commits. i_s itself is computed as $i_c + i_a$, where i_c and i_a are the number of instructions executed by committed and aborted threads respectively.

Internally, i_c, i_a, c, and f are runtime statistics gathered at each callsite. Further, we strength reduce $l \times r = (i_c + i_a/f) \times (c/f)$ to $((i_c + i_a) \times c)/(f \times f)$, replacing one out of two divide operations with a multiply. This formula implies that we consider long threads with low success rates and short threads with high success rates to be as good as each other. A more sophisticated priority computation might include the various sources of overhead identified in Section 2.6.3. Although the "always fork" heuristic used in our experiments forks threads independent of priority, it is straightforward to use a heuristic that forks only above a certain priority.

The queue consists of an array of doubly-linked lists, one for each priority, and supports `enqueue`, `dequeue`, and `delete` operations. `enqueue` inserts a thread into the beginning of a list with a specified priority, `dequeue` removes a thread from the end of the highest priority non-empty list, and `delete` unlinks the specified thread. Helper OS threads compete to dequeue and run children on separate processors; our zero-sum speculative threading model ensures that one helper thread is active per free processor in the system. If a parent thread joins a child that it previously enqueued, and that child did not get dequeued by a helper OS thread, the child is deleted by simply unlinking it from the list for that priority, and its memory is recycled. Otherwise, the child has started speculative execution, and so the parent signals it to stop before beginning validation. The queue is globally synchronized using spinlocks, which works well for a small number of priorities and processors [SZ99].

Although commits always occur in correct program order, the priorities we use do imply an ordering to the *beginning* of speculative thread execution. This ordering can invert or maintain the dependence ordering between threads. For two threads with different priorities both waiting on the priority queue, the higher priority thread H will begin execution before the lower priority thread L, even if in sequential program order L comes before H. The only exception is if H is deleted from the queue by its parent. For two threads of the same priority A and B attached to a parent thread P, where A occurs earlier in sequential program order than B, either in-order or out-of-order nesting is possible, as shown in Figure 2.7, each of which has different implications for execution ordering.

Under in-order nesting, where P forks A and then A forks B, there is only one possible ordering of queue operations: A is enqueued, A is dequeued, B is enqueued, and finally B is either dequeued or deleted. This is true even when A and B have different priorities.

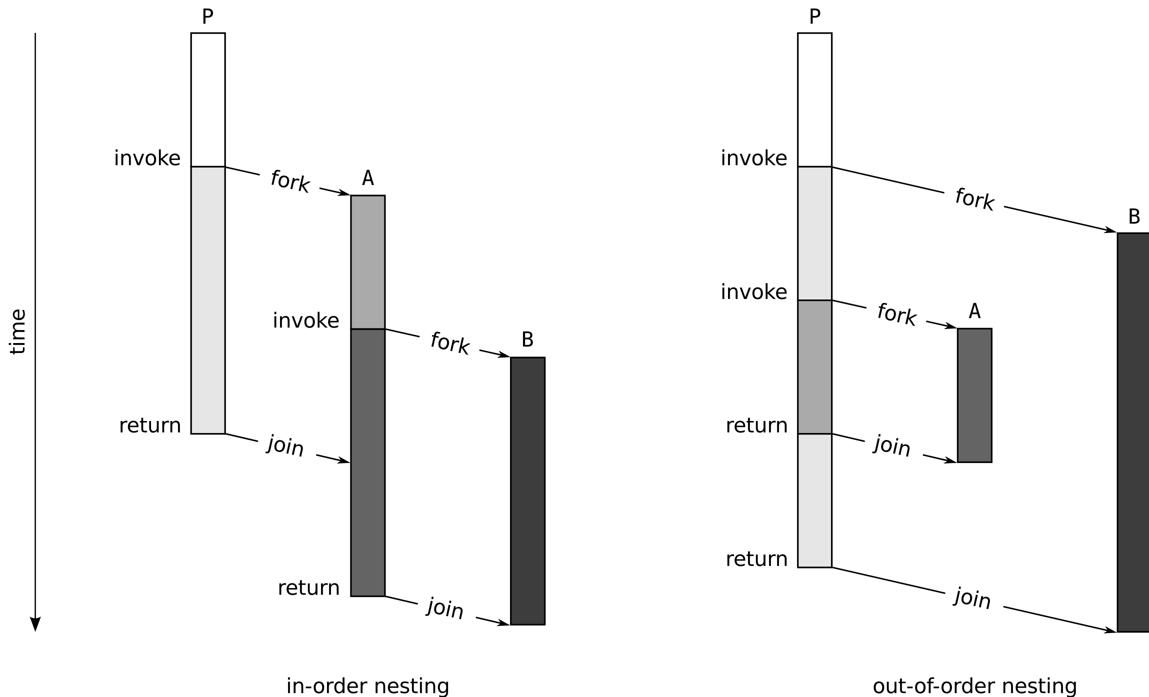


Figure 2.7: *In-order nesting vs. out-of-order nesting.*

Under out-of-order nesting, where the same parent thread P forks A in a higher stack frame than B (assuming call stacks grow upwards), B will be enqueued before A. In our design, dequeueing threads removes them from the end of the priority list, such that B will also begin execution before A, maintaining the out-of-order relationship between the threads.

It would be straightforward to conduct alternative experiments in which dequeueing threads removed them from the beginning of the priority list, correcting the execution ordering between A and B to be in-order in the case of out-of-order nesting. We chose to prioritize B over A under out-of-order nesting to allow for a longer parent execution in the event that P returns to A's fork point before it has been dequeued, thereby deleting it from the queue instead of joining it. Although our final implementation of SableSpMT supports both in-order and out-of-order nesting, the experiments in this chapter are based on an initial version that only supports out-of-order nesting; it was the profiling work here that motivated the later in-order nesting support. Both kinds of thread nesting are explored in detail in Chapter 4.

2.4.3 Return Value Prediction

Speculative children forked at non-void callsites need their operand stack height adjusted to account for the return value, and must be aborted if an incorrect value is used. Accurate return value prediction can significantly improve the performance of Java MLS, particularly because return values are on average consumed within 10 instructions after a method call [HBJ03]. We previously reported on our initial return value prediction implementation in SableSpMT [PV04b] and the use of two static compiler analyses [PV04a]. The attributes generated by the RVP compiler analyses are parsed during method preparation, and can be used to relax predictor correctness requirements and reduce memory consumption. We discuss these analyses in Section 2.6.2.

We depend on a variety of well-known predictors in the initial implementation of SableSpMT used for the experiments in this chapter. Fixed-space designs include last value and stride predictors (Table 3.1), a two-delta stride predictor (Table 3.2), and a parameter stride predictor (Table 3.6). The table-based designs we use are shown in Figure 2.8. They include a finite context method predictor that hashes together a history of the last five return values (Table 3.4), and a new memoization predictor that hashes together method arguments (Table 3.5). These six predictors are unified by a hybrid predictor that executes and updates them all on every method invocation, selecting the best performing one to make a prediction (Table 3.7). Hybrid predictors are associated with individual callsites, along with other dynamic per-callsite information. In Chapter 3 we explore the RVP subsystem, a wider variety of sub-predictors, and adaptive optimizations to the hybrid predictor in detail.

The hybrid predictor we use maintains a dynamic measure of predictor accuracy per callsite, as discussed in Chapter 3. The accuracy of a given hybrid predictor instance can be used as a measure of *predictor confidence* for that callsite. We did not include predictor confidence as a separate input to the priority computation discussed in Section 2.4.2, for the success rate measure used already includes return value predictor confidence implicitly. This is because return value prediction failure induces speculation failure. Nevertheless, it would be straightforward to modify SableSpMT to include return value predictor confidence explicitly in the priority computation and thus dynamic fork heuristics.

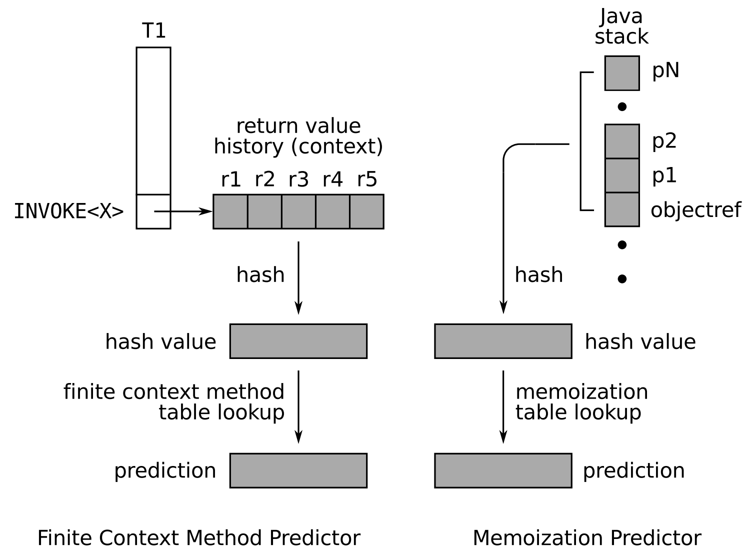


Figure 2.8: *Table-based return value prediction.*

2.4.4 Dependence Buffering

Many SpMT designs propose a mechanism to track speculative reads from main memory and buffer speculative writes to main memory to protect against dependence violations; if not, they propose an alternative *undo logging* mechanism to reverse speculative writes. We focus on tracking reads and buffering writes, which we refer to collectively as *dependence buffering*. In hardware, dependence buffers can be built as table based structures similar to caches [SCZM05]. We propose a similar design for software SpMT, as shown in Figure 2.9. In Java, main memory consists of object instances and arrays on the garbage-collected heap, and static fields in class loader memory. As discussed in Section 2.3.3, one set of bytecodes writes to class static, object field, and array element locations, and a matching set reads from these locations. We modified the speculative versions of these bytecodes to access the dependence buffer instead of main memory.

At a high level, the dependence buffer maps addresses to values using a write hashtable layered on top of a read hashtable, which in turn is layered on top of main memory. A speculative write to the buffer goes directly to the write hashtable, overwriting any previous value for that address, such that the buffer always contains the latest speculative writes.

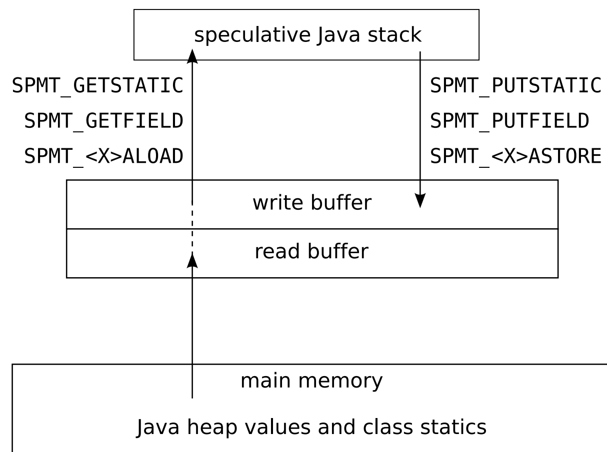


Figure 2.9: *Dependence buffering.* When a speculative global read instruction is executed, first the write buffer is searched, and if it does not contain the address of the desired value then the read buffer is searched. If the value address is still not found, the value at that address is loaded from main memory. When a speculative global write instruction is executed, the write buffer is searched, and if no entry is found a new mapping is created.

A speculative read from the buffer first searches the write hashtable. If the value is not found, it searches the read hashtable. If the value is still not found, it is retrieved from main memory and stored in the read hashtable, such that the buffer always contains the earliest speculative reads. Thus the read hashtable tracks RAW dependences and the write hashtable buffers WAR and WAW dependences. Note that our design does not support forwarding values between buffers in different speculative threads. This is an optimization that can reduce misspeculations, and we include it as part of our future work in Section 6.2.

At a low level, buffers are attached to speculative thread objects and implemented as pairs of hashtables. For a given read or write hashtable, the values are stored and retrieved using the value address as a key. We use open addressing hashtables with double hashing for fast lookup [CLRS01]. The algorithm to find the index of a given key (address) is provided by the function `search_table_for_key` shown in Figure 2.10. It requires that the table size be a power of two. For a particular key and hashtable, this algorithm will search until one of the following three cases is true: either the key is found, the key is not found and there is an empty slot for a new key, or the key is not found and there are no empty

```
word_t
search_table_for_key (table_t *table, word_t key)
{
  boolean_t new_key = FALSE;
  boolean_t found = FALSE;
  word_t index = 0;
  word_t hash_1 = key & (table->size - 1);
  word_t hash_2 = hash_1 | 1;
  for (word_t i = 0; i < table->size && !new_key && !found; i++)
    {
      index = (hash_1 + i * hash_2) & (table->size - 1);
      if (table->keys[index] == 0)
        new_key = TRUE;
      if (table->keys[index] == key)
        found = TRUE;
    }
  if (new_key)
    {
      table->keys[index] = key;
      table->entries[table->load++] = index;
    }
  else if (!found)
    table->overflow = TRUE;
  return index;
}
```

Figure 2.10: *Source code for hashtable key index lookup based on double hashing.*

slots, which means that the table is overflowing. In the event of a new key being added, its index is appended to an array-based list of entries. This list is later used for fast iteration over hashtable elements during validation, committal, and reset operations. The key index retrieved by this function is used as an index into an array of values to implement buffer read and write operations. Note that we also use open addressing hashtables with double

hashing to implement the table-based return value predictors described in Section 2.4.3 and Chapter 3, except that there the input keys are computed as message digests.

For the experiments in this chapter, each child thread has nine pairs of read and write hashtables associated with it: one pair for each of the eight Java primitive types and another pair for reference types. These tables each have a fixed capacity of 128 entries. When we later refactored SableSpMT into the implementation in libspmt [PVK07], we used a single pair of read and write hashtables for the entire thread and tagged values according to their widths to differentiate between types. Further, although these new tables are also created with an initial 128 entry capacity, they can expand dynamically.

2.4.5 Stack Buffering

In addition to heap and static data, speculative threads may also access local variables and data stored on the Java operand stack. It follows that stack accesses must be buffered to protect the parent stack in the event of failure, as shown in Figure 2.11. The simplest mechanism for doing so is to copy stack frames from parent threads to separate child stacks both on forking children and on exiting methods speculatively. Additionally, children must create new stack frames for any methods they enter.

Pointers to child threads are stored one per stack frame. This allows for convenient out-of-order thread nesting [RTL+05], such that each parent can have multiple immediate children. This in turn exposes significant additional parallelism. When in-order speculation is combined with out-of-order nesting it can lead to a tree of children for a single fork point. In this chapter we consider only out-of-order nesting; full support for in-order nesting is described in Chapter 4.

2.4.6 Thread Joining

Upon reaching some termination condition, a speculative child will stop execution and leave its entire state ready for joining by its parent. The child may stop of its own accord if it attempts some illegal behaviour as summarized in Table 2.1, if it reaches an *elder sibling*, that is, a speculative child forked earlier on by the same parent at a lower stack frame, or if it reaches a pre-defined speculative sequence length limit. The parent may also signal the

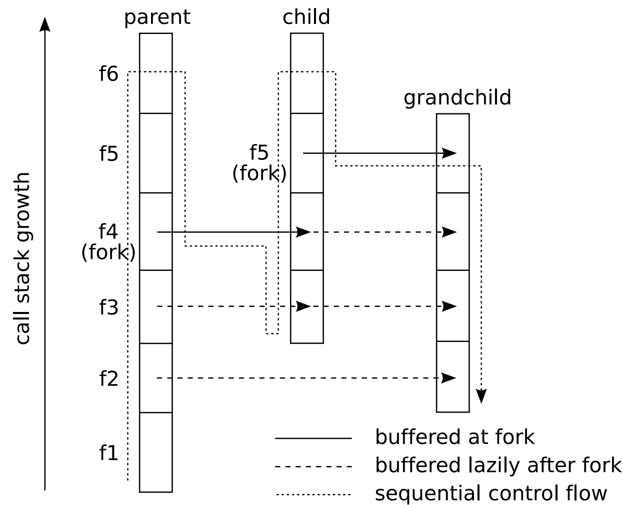


Figure 2.11: *Stack buffering. f1* through *f6* are stack frames corresponding to Java methods. A speculative child is forked at *f4* in the parent, and in turn a second-generation grandchild thread is forked at *f5* in the child. Note that this constitutes in-order nesting. Stack frames are buffered on forking, and additionally when children return from methods; *f2* in the grandchild is buffered from the non-speculative parent, as its immediate ancestor never descended below *f3*.

child to stop if it reaches the join point associated with the child's fork point, in which case it will attempt to join the child, or if it reaches the child's forking frame at the top of the VM exception handler loop, in which case it will unconditionally abort it.

The join process involves verifying the safety of child execution and committing results. First, a full memory barrier is issued, and the child is then validated according to four tests: 1) the predicted return value is checked against the actual return value for non-void methods, according to the safety constraints of static analyses [PV04a]; 2) the parent is checked for not having had its root set garbage-collected since forking the child; 3) the dependence buffers are checked for overflow or corruption; and 4) values in the read dependence buffer are checked against main memory for violations.

If the child passes all four tests, then the speculation is safe; all values in the write buffer are flushed to main memory, buffered stack frames entered by the child are copied to the parent, and non-speculative execution resumes with the `pc` and operand stack size set as the child left them. Otherwise, execution continues non-speculatively at the first instruction past the `SPMT_JOIN`. Regardless of success or failure, the child's memory is recycled for

use at future fork points, as described in Section 4.2. Note that buffer commits may result in a reordering of the speculative thread's write operations, which must in turn respect the requirements imposed by the Java memory model, as discussed in Section 2.5.

2.5 Java Language Considerations

Several traps await the unsuspecting implementor that tries to enhance a JVM to support method level speculation. These traps are actually core features of the Java language — class loading, object allocation, garbage collection, native method execution, exception handling, synchronization, and the Java memory model — and a Java SpMT implementation must handle them all safely in order to be considered fully general. The impact of these features is measured in Section 2.6.6.

2.5.1 Class Loading

All methods in Java belong to some class, such that each containing class must be linked, loaded, and initialized before its methods can execute. Some classes are loaded as part of a VM bootstrap process, whereas others are loaded when the first reference to them is encountered. In our design speculative class loading is unsafe and simply forces speculation to stop. The cost is small for most programs since classes are only loaded once.

2.5.2 Object Allocation

Object allocation occurs frequently in many Java programs, such that permitting speculative allocation significantly increases maximum child thread lengths. A benefit of speculative allocation is that it becomes unnecessary to buffer accesses to objects allocated speculatively. Speculative threads can either allocate without synchronization from a thread-local heap, or compete with non-speculative threads to acquire a global heap mutex. Speculation must stop if the object to be allocated has a non-trivial finalizer, *i.e.* not `Object.-finalize`, for it would be incorrect to finalize objects allocated by aborted children. Allocation also forces speculation to stop if either GC or an `OutOfMemoryError` would be triggered as a result. Object references only become visible to non-speculative Java threads

upon successful thread validation and committal; aborted children will have their allocated objects reclaimed in the next collection. Although this does increase collector pressure, we did not observe any difference in GC counts at the default heap size when speculative allocation was enabled.

2.5.3 Garbage Collection

All objects in Java are allocated on the garbage-collected Java heap. This is one of the main attractions of the language, and as such, any serious proposal to extend it must consider this feature; indeed, many Java programs will simply run out of memory without GC. SableVM uses a stop-the-world semi-space copying collector by default, meaning that every object reference changes upon every collection; thus, any speculative thread started before GC must be invalidated after GC. Speculative threads are invisible to the rest of the VM and are not stopped or traced during collection. However, because heap accesses are buffered, speculation can safely continue during GC, even if ultimately the computation is wasteful. The mechanism for invalidation is simple: threads are aborted if the collection count of the parent thread increases between the fork and join points. The default collector in SableVM is invoked relatively infrequently, and we find that GC is responsible for a negligible amount of speculative invalidations. Other GC algorithms are trickier to negotiate with, and may require either pinning of speculatively accessed objects or updating of dependence buffer entries.

2.5.4 Native Methods

Java provides access to native code through the Java Native Interface (JNI) [Lia99]. Native methods are used in class libraries, application code, and the VM itself for low-level operations such as thread management, timing, and I/O. Although timing-dependent execution cannot always be sped up, speculation can still be useful; for instance, consider higher quality processing of video playback buffers by speculative threads. Speculation must stop upon encountering native methods, as these cannot be executed in a buffered environment without significant further analysis. However, non-speculative threads can safely execute native code while their speculative children execute pure bytecode continuations.

2.5.5 Exceptions

Implicit or explicit exceptions simply force speculation to stop. Speculative exception handling is not supported in SableSpMT for three reasons: 1) exceptions are rarely encountered, even for "exception-heavy" applications such as `jack` (refer to Table 3.8); 2) writing a speculative exception handler is somewhat complicated; and 3) exceptions in speculative threads are often the result of incorrect computation, and thus further progress is likely to be wasted effort.

Non-speculatively, if exceptions are thrown out of a method in search of an appropriate exception handler, any speculative children encountered as stack frames are popped must be aborted. In order to guarantee a maximum of one child per stack frame, children must be aborted at the top of the VM exception handler loop, before jumping to the handler `pc`. This prevents speculative children from being forked inside either `catch` or `finally` blocks while another speculative child is executing in the same stack frame.

2.5.6 Synchronization

Object access is synchronized either explicitly by the `MONITORENTER` and `MONITOREXIT` instructions, or implicitly via synchronized method entry and exit. Speculative synchronization is unsafe without explicit support [MT02], and must force children to stop; somewhat surprisingly, synchronization has been unsafely ignored by past Java SpMT studies [CO03a, HBJ03]. Non-speculatively, synchronization always remains safe, and it is even possible to fork and join speculative threads inside critical sections.

2.5.7 The Java Memory Model

The Java memory model [MPA05] imposes constraints on multithreaded execution; these constraints can be satisfied by inserting appropriate memory barriers [Lea05b]. Speculative execution can only continue past a memory barrier if the dependence buffer records an exact interleaving of memory accesses and the relevant barrier operations; that we reuse entries for value addresses already in the buffer and do not record memory barriers precludes doing so in our current implementation.

The orderings required for various API calls, including non-speculative thread creation and joining, are provided by our design due to their implementations as native methods, which already force speculation to stop. For object synchronization several rules apply; most critically, a memory barrier is required before unlock operations to guarantee that writes in the critical section are visible to future threads entering the same monitor. By disabling speculative locking entirely we provide a much stronger guarantee than required; future work on speculative locking will need a finer grained approach.

Loads and stores of volatile fields also require memory barriers, to ensure interprocessor visibility between operations. Java also provides a `final` keyword which can be used to annotate fields. A non-static final field can only be written to once in the constructor method of the class that defines it. Loads and stores of final fields require barriers, except that on `x86` and `x86_64` these are no-ops [Lea05b]. However, speculatively, we must stop on final field stores, which appear only in constructors, to ensure that a final field is not used before the object reference has been made visible, a situation that is made possible by reordering writes during commit operations. Our conservative solution is to stop speculation on all volatile loads and stores and also all final stores.

2.6 Experimental Analysis

In this section we present various kinds of analysis available using SableSpMT, which themselves form an analysis of the MLS engine itself. These analyses include analysis of our return value prediction framework, static analyses for improved return value prediction, speculation overhead analysis, parallelism analysis, online profiling, Java language feature and MLS support component analysis, and speedup analysis. We provide experimental results that demonstrate how these analyses give insight into the properties of individual benchmarks, components of the framework, and the framework as a whole. We also show how the results suggest interesting areas for future investigation and optimization research.

Our codebase consisted of SableSpMT revision 4320, an extension of the SableVM 1.1.9 switch interpreter. SableSpMT also includes various static analyses that depend on slight modifications to Soot found in Soot revision 1704. For the JDK class libraries, SableVM Classpath revision 3311 was required, itself a derivative of GNU Classpath 0.13.

For benchmarks we used the SPEC JVM98 benchmark suite at size 100 (S100) [Sta98]. Although `raytrace` is technically not part of SPEC JVM98 and therefore excluded from geometric means, we include results for purposes of comparison; it is the single-threaded equivalent of `mtrt`. All runtime results were performed on a 1.8 GHz 4-way SMP AMD Opteron machine running Linux 2.6.7 using native 64-bit binaries. Children were forked at every callsite reached non-speculatively, which meant that in-order nested speculation was disabled but out-of-order nesting was enabled. These two forms of nesting are discussed in Section 2.4.2 and illustrated in Figure 2.7. All free processors were occupied by speculative helper threads, and an optimally accurate return value prediction configuration was used, unless otherwise stated.

2.6.1 Return Value Prediction

We first performed an initial RVP study without speculation in which we instrumented the RVP component of our system to obtain a wealth of profiling information [PV04b]. When a variety of existing predictors from the literature were combined in a hybrid we achieved an average return value prediction accuracy of 72% over SPEC JVM98. The inclusion of our new memoization predictor increased this average to 81%. Exploiting VM level knowledge about the width of primitive types then allowed us to reduce hashtable memory by 35%. One of the more interesting results obtained was how the finite context method and memoization predictors exhibited dramatically different accuracy depending on benchmark. Another was how we were able to identify a small percentage of callsites as being responsible for either the production or consumption of highly variable data, according to final finite context method or memoization predictor sizes respectively. We build on this study and explore RVP exhaustively in Chapter 3.

Given our success with predicting return values, the RVP system became a key part of SableSpMT. The predictors used in our initial study are also used in every experiment in this chapter, except where explicitly disabled. Specific details on the individual predictors are given in Section 2.4.3. Note that one important difference that ends up affecting speed, accuracy, and memory consumption is that our initial results were based on a 32-bit architecture, whereas the version of SableSpMT we analyse here and in Chapter 3 is 64-bit.

The RVP system is easily extendable to support new types of return value predictors, and could even be used for general purpose load value prediction by speculative threads. In general, as far as using SableSpMT as an experimental framework is concerned, any MLS support component could be instrumented for similar analysis purposes, for example the dependence buffer or priority queue.

2.6.2 Static Analysis Integration

Following our initial RVP study, we next used side effect and callgraph information derived from Soot's points-to analysis in two compiler analyses for improved RVP [PV04a]. We could then study the effect on runtime predictor behaviour using SableSpMT. The first analysis is a *return value use* analysis that determines how return values are used after returning from a method call. We found statically that an average 10% of non-void callsites generate *unconsumed* return values that are never used, and 21% of callsites generate *inaccurate* return values, which we define as those that are used only inside boolean or branch expressions. Unconsumed return values do not need any prediction, whereas inaccurate return values have relaxed predictor accuracy constraints that must nevertheless be checked at validation time. Actual runtime measurements show less improvement: only 3% of dynamic method invocations return unconsumed values, whereas 14% return inaccurate values. This analysis does reduce hashtable collisions, saving 3% of predictor memory and increasing accuracy by up to 7%.

The second analysis computes *parameter dependence*, a form of slicing that determines which parameters affect the return value. Statically, we observed that 25% of consumed callsites with one or more parameters have zero parameter dependences, and 23% have partial dependences, such that the return value does not depend on one or more parameters. At runtime, however, we found that 7% of dynamic method invocations have zero dependences and only 3% have partial dependences. The results of this analysis are exploited to eliminate inputs to the memoization predictor, and the accuracy of memoization alone for `jack`, `javac`, and `jess` increases by up to 13%, with overall memory requirements being reduced by a further 2%. Although these analyses yield only incremental improvements, at least in their current form, they do demonstrate how new static analyses can be easily

incorporated into SableSpMT and both validated and employed at runtime. Although their contribution is minor, we nevertheless make use of these analyses for the subsequent results in this chapter.

2.6.3 Speculation Overhead

The overhead of thread operations in any SpMT system is a major concern [WS01], and this is especially true in a pure software environment. As shown in Figure 2.12, parent threads suffer overhead when forking, enqueueing, joining, and validating child threads, and child threads suffer on startup and when they reach some stopping condition. We introduced profiling support based on hardware timestamp counters into our framework in order to provide a complete breakdown of SpMT overhead incurred by both non-speculative parent and speculative helper threads; the results are shown in Tables 2.2 and 2.3 respectively.

The striking result in Table 2.2 is that the parent spends so much of its time forking and joining speculative threads that its opportunities for making progress through normal Java bytecode and native code execution are reduced by up to 5-fold. This overhead on the non-speculative thread is in the critical path of the program, which means that any optimizations here will improve performance. We see that joining threads is significantly more expensive than forking threads, and that within the join process, predictor updates and waiting for the speculative child to halt execution are the most costly sub-categories. We choose to focus on return value prediction overhead for its unique relevance to method level speculation, and describe optimizations to predictor updates and validation in Chapter 3. Of course, the other overhead sub-categories are not insignificant, and we have several suggestions for future work in this area. The cost of buffer validation and child committal would perhaps be best addressed by using one of several highly optimized software transactional memory packages, as discussed in Section 5.6. The cost of deleting forked but unstarted children could be minimized by a less aggressive or adaptive forking strategy, perhaps following the guidelines in Chapter 4. Finally, the cost of signalling and waiting could be addressed by an AOT or JIT compiler that modified the parent method to signal the child some number of instructions before returning from the call. The cost of profiling itself is high, but disappears in a final production build.

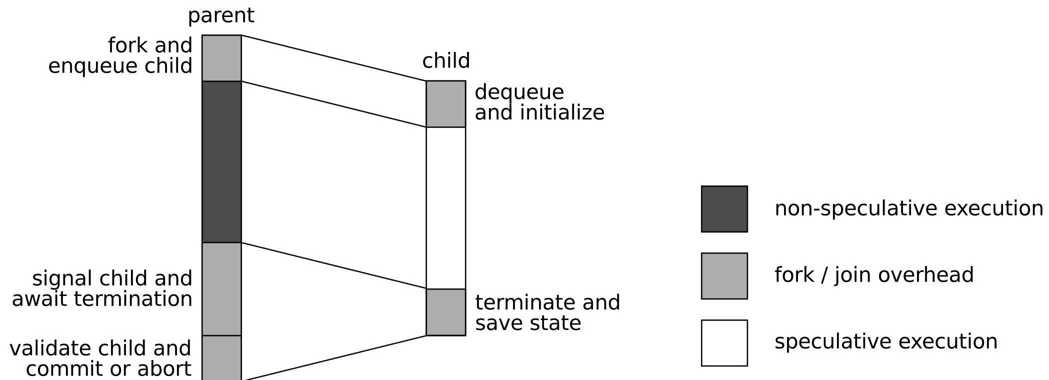


Figure 2.12: *Speculation overhead.* Both non-speculative parent and speculative child threads suffer wasted cycles due to overhead at fork at join points.

parent execution	comp	db	jack	javac	jess	mpeg	mtrt	rt
USEFUL WORK	39%	24%	29%	30%	21%	59%	49%	58%
initialize child	2%	5%	3%	4%	4%	2%	1%	2%
enqueue child	4%	10%	10%	9%	7%	3%	2%	2%
TOTAL FORK	6%	15%	13%	13%	11%	5%	3%	4%
update predictor	7%	13%	12%	11%	12%	6%	7%	7%
delete child	5%	5%	5%	4%	5%	2%	2%	2%
signal and wait	15%	14%	11%	11%	19%	8%	26%	11%
validate prediction	4%	4%	4%	5%	7%	3%	2%	3%
validate buffer	4%	6%	6%	5%	5%	3%	1%	2%
commit child	5%	5%	7%	6%	6%	3%	2%	3%
abort child	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
clean up child	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
profiling	11%	10%	10%	12%	11%	7%	5%	6%
TOTAL JOIN	53%	59%	57%	56%	67%	34%	47%	36%
PROFILING	2%	2%	1%	1%	1%	2%	1%	2%

Table 2.2: *Non-speculative thread overhead breakdown.* Parent execution consists of useful work, fork overhead, and join overhead, and also the profiling overhead inherent in delineating these three broad tasks. Profiling in the join process includes the cost of gathering overhead info for the other eight sub-tasks, and of updating various SpMT statistics.

helper execution	`comp`	`db`	`jack`	`javac`	`jess`	`mpeg`	`mtrt`	`rt`
IDLE	86%	82%	78%	78%	78%	55%	53%	71%
INITIALIZE CHILD	3%	4%	4%	4%	4%	2%	5%	4%
startup	<1%	<1%	<1%	<1%	<1%	<1%	1%	<1%
query predictor	3%	5%	4%	4%	6%	5%	15%	8%
useful work	5%	6%	10%	10%	10%	34%	20%	13%
shutdown	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
profiling	<1%	<1%	<1%	<1%	<1%	1%	2%	1%
EXECUTE CHILD	9%	12%	16%	16%	17%	41%	40%	24%
CLEAN UP CHILD	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
PROFILING	1%	1%	1%	1%	<1%	1%	1%	<1%

Table 2.3: *Speculative thread overhead breakdown.* Helper SpMT threads execute in a loop, idling for an opportunity to dequeue children from the priority queue, and then initialize them, execute them, and clean them up. The child execution process itself consists of startup, querying the return value predictor, useful work (*i.e.* bytecode execution), and shutdown, induced by reaching some termination condition. There is profiling overhead both when executing speculative code, and when switching between tasks in the helper loop.

In Table 2.3, we can make several observations about the execution of speculative children. First, the SpMT helper threads spend the majority of their time being idle, waiting to dequeue tasks from the priority queue, the implication being that the queue is often empty. In these experiments, we allow for out-of-order nesting [RTL+05], in which multiple *immediate* children are attached to a non-speculative parent, one per Java stack frame. However, we do *not* allow for in-order nesting, wherein speculative children can fork speculative children of their own, which greatly limits the available parallelism. We extend our system to support in-order nesting in Chapter 4.

When the helper threads *are* running speculative children, they spend a majority of their time doing useful work, which is all bytecode execution for speculative threads. In fact, if idle times are ignored, the ratio of useful work to overhead is higher speculatively than non-speculatively, due to higher non-speculative overheads. Outside of bytecode execution, we see that predictor lookup is quite expensive, due mostly to our naïve hybrid design.

2.6.4 Speculative Parallelism

SableSpMT allows for investigation into runtime speculative parallelism at a fairly fine granularity. Speculative thread lengths are recorded on a per-callsite basis and can be analysed in both the single-threaded simulation and multithreaded modes. Thread length information, particularly when associated with specific callsites, can be quite instructive as to the effect of SpMT optimizations on the system. In the single-threaded mode, children run until either an unsafe operation occurs or an arbitrary limit on sequence length is reached. Using a sequence length limit of 1000 instructions, we found that over all speculative children, 30% are successful and in the 0–10 bytecode instructions range, with very few failures, and 15% are successful and run for 90+ instructions. On the other hand, 25% of all threads are accounted for by failures at 90+, which derives from the correspondence between thread length and risk of dependence violation or unsafe execution.

In the multithreaded mode, child threads are additionally stopped when parents return to fork points or pop frames in the exception handler. Here 80% of speculative threads are accounted for by success in the 0–10 instruction range, with only 1–2% found in subsequent 10 instruction buckets. As we reduce overhead costs, we expect children to run longer, and for parallelism to increase. An interesting point to note is that in hardware simulations, thread lengths of 40 *machine* instructions are considered impressive [JEV04], and although uncommon, some children in our multithreaded mode can run for hundreds of *bytecode* instructions. Our fork heuristic that speculates on every non-speculative method call is a large contributor to short thread lengths. We explore improved fork heuristics based on program structure in Chapter 4.

In Figure 2.13, we examine *speculative coverage*, the percentage of sequential program execution that occurs successfully in parallel. We compute this as $i_c/(i_c + i_n)$, where i_c is the number of instructions executed by committed speculative threads, and i_n is the number of instructions executed by non-speculative threads. Adding processors to the system has an effect on all benchmarks, and with just 4 processors and no support for in-order nesting, the amount of parallel execution is quite high, an average of 33%. Disabling the RVP component by always predicting zero brings the average speculative coverage on four processors from 33% down to 19%. Thus we can confirm the result previously obtained

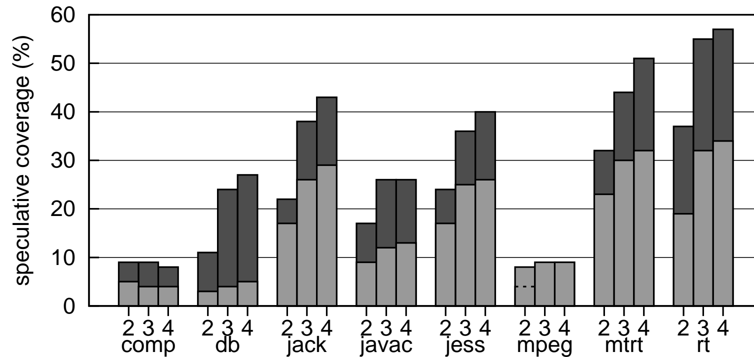


Figure 2.13: *Speculative coverage with and without RVP.* The SPEC JVM98 benchmarks are shown running with 2, 3, and 4 processors, and the dark regions indicate the improvement as return value prediction is enabled.

by Hu *et al.*, namely that RVP plays an important role in MLS [HBJ03]. Note that to truly disable return value prediction, we would need to either disable forking at non-void callsites or force all speculations at non-void callsites to fail, because predicting zero is still correct in some instances; for details, refer to the accuracy of the null predictor (N) in Figure 3.5. This means that the value of RVP is actually somewhat higher than indicated by these results.

2.6.5 Runtime Profiling

SableSpMT provides a facility for runtime profiling and feedback-directed optimization. This is often crucial to performance in runtime systems; for example, JIT compilers typically depend on interpreter profiling to determine hot execution paths [AFG+05]. We make various measurements of the dynamic performance of our system available to optimizations by associating data with the fork and join points surrounding invokes. Currently our optimizations are written to exploit per-callsite information and thus calling context-sensitivity, although the data does generally remain available on a per-target basis.

In the context of return value prediction, our hybrid predictor selects the best subpredictor over the last 32 return values, predictor hashtables expand according to load factors and performance, and future work in Chapter 3 addresses disabling sub-optimal predictors after a warmup period. In the context of choosing optimal fork points, we can

assign child thread priorities or disable speculation completely according to various dynamic profiling data, including transitive target method size, speculation success and failure and the corresponding reasons, speculative sequence lengths, predictor confidence, and premature child termination due to various stopping reasons.

Despite the extent of online profiling information, we have not discovered optimal fork heuristics, although others have had success with offline profiling for Java programs [WK05]. As we disable speculation at undesirable fork points, our system does exhibit speedup, but its source is both better opportunity for speculation as well as reduced overhead. Given that significant reductions in overhead are likely possible without reducing the number of dynamic forks, we defer investigations based on dynamically restricting fork points until no further speedup can be made on that front. However, we do explore the relationship between program structure, choice of fork point, and resulting parallelism in Chapter 4.

2.6.6 Speculation Behaviour

We now employ the SableSpMT framework to analyse the impact of both speculation support components and Java language features on MLS runtime behaviour. In Table 2.4, total counts are given for all child thread termination reasons. In all cases, the majority of children are signalled by their parent thread to stop speculation. Significant numbers of child threads are deleted from the queue, and elder siblings are frequently reached. We examined the average thread lengths for speculative children and found them to be quite short, typically in the 0–10 instruction range. These data all indicate that threads are being forked too frequently, and are consistent with the general understanding of Java application behaviour: there are many short leaf method calls and the call graph is very dense [DDHV03]. Further experiments with dynamic fork heuristics were hampered by a lack of insight into whole-system behaviour. We chose instead to explore the impact of coding idioms and fork point choice on speculation behaviour to create a set of structural fork heuristics, as detailed in Chapter 4. Method inlining is another vector for optimization. Inlining changes the call graph structure, often by eliminating calls to short leaf methods, which are naturally undesirable fork points. Hu *et al.* previously argued that the coarser method granularity of inlined execution in a JIT compiler benefits Java MLS in particular [HBJ03]. Introducing

termination reason	comp	db	jack	javac	jess	mpeg	mtrt	rt
class resolution and loading	2.14K	1.76K	94.8K	487K	3.80K	14.7K	4.79K	5.64K
failed object allocation	1	3	23	17	39	0	28	40
invalid object reference	563	553K	342K	280K	431K	485	407K	278K
finals and volatiles	842	1.45M	2.17M	1.11M	1.95M	888	115K	68.8K
synchronization	4.30K	26.8M	6.95M	17.0M	4.89M	10.4K	658K	351K
unsafe method entry or exit	2.66K	1.55K	16.0K	622K	2.62K	1.65K	3.60K	3.00K
implicit non-ATHROW exception	989K	828K	9.57K	572K	78.6K	2.00K	31.2K	20.8K
explicit ATHROW exception	0	0	187K	82	0	0	0	0
native code entry	332	28.2K	1.02M	1.02M	2.63M	527K	259K	260K
elder sibling reached	1.24M	3.81M	5.06M	16.1M	5.62M	14.1M	4.03M	4.23M
deleted from queue	348K	686	559K	3.13M	2.55M	4.48M	34.2M	1.57M
signalled by parent	202M	92.6M	20.1M	42.1M	56.3M	80.8M	122M	124M
TOTAL CHILD COUNT	204M	127M	36.5M	82.4M	74.5M	99.9M	162M	131M

Table 2.4: *Child thread termination.*

inlining into our system is part of our future work, as discussed in Section 6.2.5.

Outside of these categories, it is clear that synchronization and the memory barrier requirements for finals and volatiles are important; enabling speculative locking and recording barrier operations would allow threads to progress further. Native methods can also be important, but are much harder to treat speculatively. The other safety considerations of the Java language do not impact significantly on speculative execution; even speculative exceptions are responsible for a minority of thread terminations.

Data on the number of speculative thread successes and failures, as well as a breakdown of failure reasons, are given in Table 2.5. Failures due to GC, buffer overflows and exceptions are quite rare, and the majority of failures typically come from incorrect return value prediction. This again emphasizes the importance of accurate RVP in Java MLS, and the weak impact of exceptions and GC. Dependence violation counts are not insignificant, and reusing predictors from the RVP framework for generalized load value prediction should help to lower them. In general, failures are much less common than successes, the geometric mean failure rate being 12% of all speculations. While this is encouraging, many

join status	`comp`	`db`	`jack`	`javac`	`jess`	`mpeg`	`mtrt`	`rt`
exception in parent	0	0	386K	23.4K	0	0	0	0
incorrect prediction	18.0M	22.7M	2.80M	11.3M	5.80M	7.73M	4.85M	3.72M
garbage collection	4	20	119	206	470	0	90	68
buffer overflow	0	0	0	10	0	0	0	0
dependence violation	1.60M	1.44K	160K	1.53M	342K	14.7M	4.14M	4.00M
TOTAL FAILED	19.6M	22.7M	3.34M	12.9M	6.14M	22.4M	9.00M	7.72M
TOTAL PASSED	184M	103M	32.6M	66.4M	65.8M	73.0M	119M	122M

Table 2.5: *Child thread success and failure.*

threads are quite short due to an abundance of method calls and therefore forked children, and the high overheads imposed by thread startup. Thus it is likely the case that had they progressed a lot further, more violations would have occurred.

2.6.7 Speedup

The ultimate goal of any speculative system is measurable program speedup. Speedup can be calculated simply as the sequential run time divided by the parallel run time. Parallelism contributes positively to speedup whereas overhead contributes negatively; thus, there may be positive parallelism effects that are masked by excessive overheads, leading to system slowdown. Slowdown itself is simply the inverse of speedup. In order to factor out the effects of speculation overhead, we compute a *relative speedup*. Here the sequential run time is replaced by a new baseline execution time from experiments in which speculation occurs as normal but failure is automatically induced at every join point. This provides an upper bound on performance.

We provide overall performance data in Table 2.6. Currently, thread overheads preclude actual speedup, and run times with speculation enabled are within one order of magnitude of sequential execution. Over the entire SPEC JVM98 suite, which does not include `raytrace`, there is a geometric mean slowdown of 4.49x. This is competitive with hardware simulations providing full architectural and program execution detail, which can be slower by up to three orders of magnitude [KT98]. Slowdowns increase as parallelism in-

experiment	comp	db	jack	javac	jess	mpeg	mtrt	rt	mean
vanilla SableVM	368s	144s	43s	108s	77s	347s	55s	67s	120s
MLS must fail	1297s	931s	293s	641s	665s	669s	1017s	1530s	722s
MLS may pass	1224s	733s	211s	468s	405s	662s	559s	736s	539s
slowdown	3.33x	5.09x	4.91x	4.33x	5.26x	1.91x	10.16x	10.99x	4.49x
relative speedup	1.06x	1.27x	1.39x	1.37x	1.64x	1.01x	1.82x	2.08x	1.34x

Table 2.6: *Execution times, slowdown, and relative speedup.* The first row is the sequential vanilla SableVM run time. The second row is an experiment in which MLS occurs but failure is forced at every join point, thereby incurring speculation overhead but eliminating parallelism. The third row is the parallel run time of regular MLS execution. The fourth row is the system slowdown, computed as row 3 / row 1. The fifth row is relative speedup, computed as row 2 / row 3.

creases, due to the concomitant increase in overhead, consistent with our observation that eliminating even good speculation opportunities can lead to speedup. From an analysis perspective, experiments run in acceptable times, well suitable for normal, interactive usage. This demonstrates the utility of SableSpMT as a research and analysis framework.

When overhead is factored out, the geometric mean relative speedup over SPEC JVM98 is 1.34x, again excluding `raytrace`. This means that at least for these benchmarks, out-of-order nesting alone provides only limited speedup. Although this amount of speedup would be useful on a 2-way machine, these results are for a 4-way machine, which translates to at best 36% processor utilization for `raytrace`. This relatively low upper bound motivates any work on increasing it, such as the complementary support for in-order nesting described in Chapter 4, under which speculative threads become able to fork their own children. Although our approach here is not perfectly accurate for obvious reasons, our results do lean towards being somewhat pessimistic in terms of calculated speedup: Table 2.2 shows that MLS failure is slightly less expensive than success, and the fact that we compute a relative speedup of only 1.01x for `mpegaudio` despite a speculative coverage of 9% derives from this. In general, loop-based applications that produce or consume random compressed data exhibit the least speedup (`compress` and `mpegaudio`), numeric and embarrassingly parallel applications exhibit the greatest (`mtrt` and `raytrace`), and object-oriented applications fall somewhere in-between (`db`, `jack`, `javac`, and `jess`).

experiment	comp	db	jack	javac	jess	mpeg	mtrt	rt	mean
no method entry and exit	0.94x	1.02x	0.97x	0.98x	1.02x	0.95x	0.79x	0.91x	0.95x
no dependence buffering	1.04x	1.22x	1.12x	1.05x	1.16x	1.02x	0.95x	0.97x	1.08x
no object allocation	0.95x	1.30x	1.39x	1.26x	1.55x	0.98x	1.13x	1.23x	1.21x
no return value prediction	1.03x	1.17x	1.28x	1.24x	1.44x	1.03x	1.72x	1.70x	1.25x
no priority queueing	0.94x	1.22x	1.35x	1.32x	1.58x	0.97x	1.68x	2.05x	1.27x
full runtime MLS support	1.06x	1.27x	1.39x	1.37x	1.64x	1.01x	1.82x	2.08x	1.34x

Table 2.7: *Impact of MLS support components on application speedup.* The priority queue was disabled by only enqueueing threads if a processor was free, return value prediction was disabled by always predicting zero, and the remaining components were disabled by forcing premature thread termination upon attempting to use them. As discussed in Section 2.6.4, truly disabling return value prediction would require either disabling forking at non-void callsites or forcing all speculations at non-void callsites to fail. This means that these results actually understate the value of RVP somewhat.

Clearly there are significant improvements required in order to achieve actual speedup with our MLS design. Our overhead analysis suggests a number of potential optimizations to reduce overhead and increase the relative amounts of speculative execution. Designing and implementing these further improvements is part of our future work. In Chapter 3 we describe optimizations to the RVP system, eliminating almost all of the overhead incurred by it. In Chapter 4 we describe support for in-order nesting, which significantly increases the amount of available parallelism, and we also describe fork heuristics that operate at the level of program structure to provide longer thread lengths.

We can use a similar kind of speedup analysis to examine the importance of various contributors to parallelism in the system. Table 2.7 shows the impact of individual support components on Java MLS speedups. Speedups are given relative to the baseline experiment in Table 2.6 where speculation occurs as normal but failure is forced at every join point, thus factoring out overhead costs. The performance of the system with all components enabled and also with individual components disabled is shown to provide an understanding of their relative importance.

We note first of all that `compress` and `mpegaudio` are resilient to parallelization, likely

due to a combination of our current, naïve thread forking strategies and their use of highly variable compressed data. In some cases, disabling components can even lead to slight speedup. This phenomenon occurs if overhead costs outweigh component benefits; for example, disabling return value prediction can mitigate the cost of committing many short threads. In general, we can order the support components by importance: the priority queue is least important; method entry and exit, or stack buffering, and dependence buffering are most important; return value prediction and speculative object allocation lie somewhere in-between. However, it is important to remember that these conclusions pertain not only to the benchmarks in question but also the overall system configuration.

2.7 Conclusions and Future Work

Investigation of any sophisticated, general optimization strategy requires significant design, implementation and experimental flexibility, as well as a common ground for investigation. Further, analysis of speculative execution in Java has mostly been confined to data gathered from hardware simulation studies. Such work validates specific hardware designs, but is not typically targetted at general analysis of SpMT and associated program behaviour.

Our design focuses on defining correct Java semantics in the presence of software MLS and demonstrating the associated cost. Our main goal here is to provide a complete, correct system and basic analysis useful to further Java MLS or SpMT investigations. Our system provides a robust framework for Java MLS exploration that simplifies the implementation effort and allows for easy data gathering and analysis. That SableSpMT does not depend on a hardware simulator means that empirical measurements of MLS behaviour are made using a cycle and timing accurate implementation, that experimentation with different, existing multiprocessors is possible through porting to new architectures, and that a variety of MLS or even more general SpMT designs are available due to the general plasiticity of software components.

We include detailed collection of dynamic data and also allow for application of internal feedback at runtime. To evaluate high level program information we include an interface to Soot-generated Java attributes, and can thus incorporate static information as well. We have demonstrated the use of all these features through realistic optimization and perfor-

mance analyses. Measurements of speculative sequence length, speculative coverage, and relative speedup all indicate that significant parallelism does exist in the sequential threads of Java programs, and our analysis of speculative overhead indicates where to focus optimization efforts. We are relatively optimistic as to improving the efficiency of our initial MLS implementation.

Language and software based thread level speculation requires non-trivial consideration of the language semantics, and Java in particular imposes some strong design constraints. Here we have also defined a complete system for Java MLS in particular, taking into account various aspects of high level language and virtual machine behavioural requirements. Our implementation work and experimental analysis of Java-specific behaviour show that while most of these concerns do not result in a significant impact on performance, conservatively correct treatment of certain aspects can reduce potential speedup, most notably synchronization. Part of our future work is thus to investigate different forms of speculative locking [RG01, MT02, RS03] within a Java-specific context.

As with any speculative system, performance and SpMT overhead are major concerns, and efforts to improve speedup in many fashions are worthwhile, as suggested by our profiling results. We address the problem of RVP overhead in Chapter 3 with support for adaptive hybrid prediction, and the problems of idle processors and short threads in Chapter 4 with support for in-order nesting and structural fork heuristics respectively. In fact, when we developed the support for in-order nesting as described in Chapter 4, we found that when combined with out-of-order nesting it exposed so *much* parallelism that a straightforward experimental comparison with the results in this chapter was impractical.

We are confident that other sources of overhead can be greatly reduced in our prototype implementation, through optimization of individual components, greater use of high level program information, and employment of general and Java-specific heuristics for making forking decisions and assigning thread priorities. Further speedup can also be provided by reusing parts of the RVP subsystem for generalized load value prediction. Significant further work is required, but providing JIT compiler support for software MLS is another major challenge. The presence of a JIT offers both positive and negative opportunities for MLS analysis and execution, and will certainly be interesting to examine. There are two key techniques directly related to MLS that are enabled by either AOT or JIT compiler support.

First, method inlining should reduce the overhead due to forking threads on very short leaf methods that almost immediately return and join their children. Second, method *outlining* or extraction of key loop bodies into their own methods could expose more parallelism in certain loop-centric applications. Disabling fork points inside library code could also have a strong positive effect. Soot would be a useful experimental framework for prototyping these transformations. We describe opportunities for future work in greater detail in Section 6.2.

Continued improvements to our framework will also provide for new research opportunities. We have implemented method level speculation, but other researchers have also had success with loop level [RS01], lock level [MT02,RG01], and arbitrary speculation [BF04] strategies. These approaches have largely common internal requirements, and side-by-side implementations within our framework will make direct and meaningful comparisons of the various techniques feasible, and furthermore enable their composition.

Chapter 3
Adaptive Software Return Value Prediction

In the preceding chapter, return value prediction (RVP) was used by speculative children to predict the return values of method calls before they actually completed, exposing additional parallelism under method level speculation by allowing speculation to proceed past the consumption of return values in non-void method continuations. In addition to improving MLS performance, RVP can also enable a number of other program optimizations and analyses. However, despite the apparent usefulness, RVP and value prediction in general have seen limited uptake in practice. Hardware proposals have been successful in terms of speed and prediction accuracy, but the cost of dedicated circuitry is high, the available memory for prediction is low, and the flexibility is negligible. Software solutions are inherently much more flexible, but a naïve approach can only achieve high accuracies in exchange for significantly reduced speed and increased memory consumption. In this chapter we first express many different existing prediction strategies in a unification framework, using it as the basis for a software implementation. We then explore an adaptive software RVP design that relies on simple object-orientation in a hybrid predictor. It allocates predictors on a per-callsite basis instead of globally, and frees the resources associated with unused hybrid sub-predictors after an initial warmup period. We find that these techniques dramatically improve speed and reduce memory consumption while maintaining high prediction accuracy. The framework we present here is modular enough for general purpose reuse in a variety of applications, which we discuss.

3.1 Introduction

Return value prediction (RVP) is a runtime technique for guessing the result of a function, method, or procedure call. It is a specific case of value prediction in general, differentiated by the fact that functions may take arguments, and also by the fact that as the core building block of modularity, functions provide an extremely broad range of behaviour. Value prediction enables a variety of speculative optimizations, with their success and practical value depending on the accuracy and relative overhead of the prediction system.

Following our initial results in Chapter 2, we knew that RVP was beneficial to MLS execution, per Figure 2.13 and Table 2.7. However, it was also a source of significant performance overhead: on average, predictor execution accounted for 14% of non-speculative execution and 44% of speculative execution, per Tables 2.2 and 2.3 respectively. To address this, we first refactored our JVM-based implementation of RVP into libspmt, a software library with much cleaner, object-oriented code, as described in Chapter 2. This included removing the overlap between sub-predictor state, which in turn enabled an optimization based on adaptively specializing hybrid predictors. We describe these new hybrids in this chapter and demonstrate their comparable accuracy and improved performance. We also increase the number of predictors under consideration from six to twelve, making this study much broader than our initial efforts.

Value prediction is typically investigated in a hardware context, where the focus is on providing high accuracy with minimal circuit and cycle costs on a novel architecture. Software designs are much less common, but can be supported on existing and off-the-shelf machines. The primary implementation advantages of software prediction are relatively unbounded memory resources, cheap development costs, and high level runtime information. In terms of applications, software prediction allows greater and more portable use of value prediction data in optimization and analysis, but it also requires careful optimization and understanding of predictor performance in order to ensure practical efficiency. Previous work in software value prediction has concentrated on mimicking hardware designs in software. We believe that software value prediction can be more generally useful and is worth exploring in its own right, its relationship to hardware value prediction being analogous to that between software transactional memory and hardware transactional memory.

In this chapter we seek to establish a software state of the art in value prediction by providing a fast, accurate, and memory efficient design and implementation for return value prediction. Note that although in this chapter we focus on making accurate predictions, accuracy requirements can be relaxed somewhat by a *return value use* analysis, as discussed in Section 2.6.2.

Our approach was to implement several known predictors, including both simple fixed-space designs that need only limited resources and more complex table-based predictors that have significant resource requirements. A significant problem we encountered in reviewing the hardware literature was understanding exactly how the existing predictors worked, and how they were related to each other. To this end we developed a unification framework for organizing the various predictors and created straightforward software implementations of them. We included both space-efficient computational predictors and space-inefficient table-based predictors in our design. The higher level of abstraction provided by a software approach simplified the design of easily composable, modular predictors, and this was in fact essential to designing an effective software hybrid predictor, as well as exposing the potential for several new sub-predictors. We applied our predictors to the standard SPEC JVM98 Java benchmarks to measure their return value predictability, as well as the relative accuracy, speed, and memory consumption of individual predictor types.

The core of our design is a hybrid predictor that brings together all of the predictors in our framework. Hybrids work to select the best-performing sub-predictor for a given prediction, based on either offline or online profiling. Figure 3.1 shows what a typical implementation of hybrid RVP in hardware might look like. First to make a prediction, a callsite address is hashed to an entry in a primary hashtable. This entry contains the hybrid predictor state, which includes prediction accuracies for individual sub-predictors as well as stateful information they might need, such as a history of return values. The hybrid then selects the best performing sub-predictor to create a prediction. In-place sub-predictors compute a value based directly on the state, whereas table-based sub-predictors hash components of the state to a predicted value in a secondary hashtable. On each prediction, even though only one will be selected, all sub-predictors execute, which in hardware is easily parallelized. When the function returns from the call, sub-predictor correctness be-

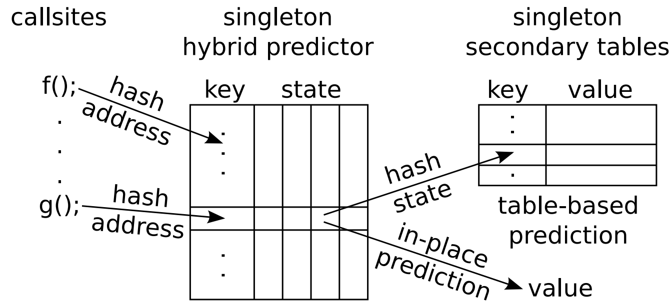


Figure 3.1: *Hybrid prediction in hardware (conventional).*

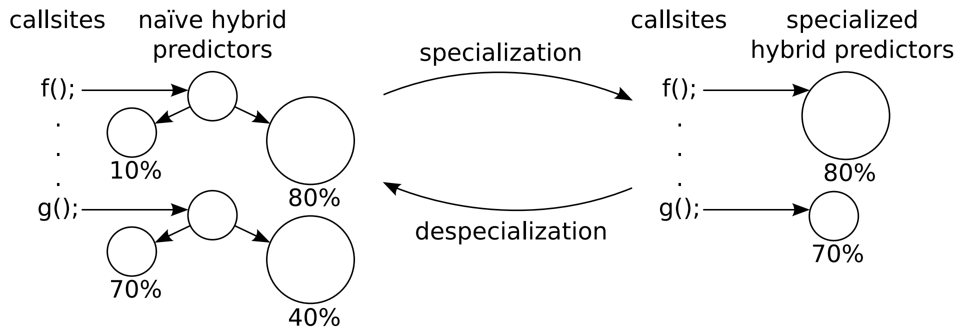


Figure 3.2: *Hybrid prediction in software (novel).*

comes known, and the hybrid state and all corresponding table-based predictor entries get updated. The most notable feature for our purposes is that due to hardware constraints, all data structures are fixed-size global singletons. Given that hybrids necessarily track predictor accuracy through some kind of confidence measure, this information could also be used as an input to dynamic fork heuristics, as discussed in Section 2.4.3.

Although hardware hybrid predictors are well-studied [BZ02], software hybrids are not. Our hybrid design exploits its software context to provide adaptivity, as shown in Figure 3.2. The first major kind of adaptivity is that a single hybrid predictor instance is associated with each callsite, which allows for scaling according to program size and client usage. Each hybrid has some private state, and each sub-predictor has its own state as well. Importantly, there is no state sharing between sub-predictors. On prediction and update, the hybrids execute and update every sub-predictor. This design can be extended

through sub-classing, avoids conflicts, achieves high accuracy, and allows for tables to grow as necessary. The primary disadvantages are that serialized sub-predictor execution leads to high overhead costs and that the memory consumption can be excessive. The second major kind of adaptivity is an attempt to optimize away these costs. After a warmup period, if the accuracy of an individual sub-predictor meets a certain threshold, the hybrid specializes. This frees all other predictor resources, such that prediction and update only access the individual sub-predictor. If accuracy ever drops below a certain threshold, the hybrid despecializes. Thus we can maintain accuracy while reducing speed and memory overhead.

3.1.1 Contributions

We make the following specific contributions:

- A unification framework for specifying and relating predictors to each other based on the patterns they capture, their mathematical expression as functions of inputs and value sequences, and their software implementations. This work clarifies the extant literature, exposes the potential for new predictors, and demonstrates how object-oriented predictor composition can simplify understanding and implementation.

- Several new sub-predictors, including a 2-delta last value predictor, a table-based memoization predictor that hashes together function arguments, and memoization stride and memoization finite context method predictors derived from it. These argument-based predictors capture repetition in function inputs that existing value history-based predictors do not. Further, the table-based predictors in our software design use dynamically expanding hashtables to conserve memory where possible without affecting accuracy.

- An adaptive software hybrid predictor composed of many sub-predictors that dynamically specializes to whichever sub-predictor performs best. Its object-oriented design and implementation enables two unique optimizations. First, it allocates one hybrid predictor instance per prediction point to eliminate conflicts and improve accuracy. Second, it identifies ideal sub-predictors at runtime and specializes at a pre-

diction point granularity, bypassing the execution of unused sub-predictors and actually freeing their associated data structures. The end result is dramatic speed and memory consumption improvements that do not sacrifice high prediction accuracy.

- A software library implementation of return value prediction. This library is open source, portable, modular, and supported by unit tests. We use this library and its built-in profiling to obtain a comprehensive set of speed, memory, and accuracy prediction data for both our hybrid and its component sub-predictors, gathered at every method invocation over SPEC JVM98 at size 100, a significant improvement to existing data [HBJ03].

In the next section, we present our predictor unification framework. Section 3.3 describes our experimental setup, and Section 3.4 provides an initial performance evaluation. We then develop and apply our adaptive hybrid design in Section 3.5 to optimize these results. Finally, we present our conclusions and future work. Related work specific to RVP is described in Chapter 5.

3.2 Predictor Unification Framework

A wide variety of value predictors have been proposed, making a basic organization and evaluation essential to our study. Many predictors described in the literature are presented as hardware implementations, often using circuit diagrams. This approach clearly expresses the design in terms of feasibility, power and space efficiency, but many of these details can also obscure the intended algorithmic behaviour of the predictor. In designing a software solution, we abstracted the simplest implementation approach for each predictor, and so discovered many commonalities between predictors that are not immediately apparent in hardware designs. Based on this exploration, we developed a unification framework for value predictors to clarify their intended behaviour and implementation and relate them to each other. This framework also suggested several new predictors.

Tables 3.1–3.7 give a structured presentation of a variety of common predictors. These tables contain typical history-based predictor designs, extended predictors that also consume argument state, and composite predictors that contain sub-predictors. In each case we

provide an idealized mathematical expression, an example if appropriate, and the stateful data and pseudo-code used to implement the actual predictor. The mathematical expressions illustrate predictor behaviour by showing how the current prediction (v_n) is derived from a history of actual return values ($v_{n-1}, v_{n-2}, \ldots$), as well as current and past function arguments ($args(n), args(n-1), \ldots$). Implementation details include fields for actual state and pseudo-code inside `predict` and `update` functions that provide a common predictor interface. `predict` optionally takes function arguments and returns a new predicted value, while `update` takes the actual return value and updates internal predictor state. For brevity we use several non-standard but self-explanatory functions in these descriptions. Our unification framework does not include predictors that are unsuitable for return value prediction, nor those that are substantially equivalent to the ones presented here. However, extensions are straightforward, and our experience suggests that all predictors benefit from expression in this form.

In the following subsections we describe our logic in constructing Tables 3.1–3.7, and give further detail on the individual predictors. We follow this in Section 3.4 with an experimental analysis using our software RVP framework, exploring the relative accuracy of different predictor designs as well as their memory and time costs.

3.2.1 History-Based Predictors

Tables 3.1–3.4 contain predictors based only on the history of return values for the associated function. We used predictor names as reported in the literature, except for last N stride, which is a local version of the global gDiff predictor [ZFC03]. At the top of each table are predictors that derive their prediction from the value history directly, whereas at the bottom are predictors that use the differences or *strides* between values in the history. It is useful to think of the stride predictors as derivatives of the value based predictors; the word ‘differential’ chosen by the creators of the differential finite context method predictor in Table 3.4 is expressing this relationship [GVdB01]. This organized division between primary and derivative forms suggests a new 2-delta last value predictor, shown in Table 3.2. We used a standard value of $N = 4$ in our experimental analysis of the last N value and last N stride predictors in Table 3.3, where N is the value or stride history length. We used a similarly

standard value of $C = 5$ in our analysis of the finite context method, differential finite context method, and memoization finite context method predictors in Tables 3.4 and 3.6, where C is again the value or stride history length.

Last Value (LV)

The last value predictor is perhaps the simplest useful predictor. It merely predicts that the return value v_n will be the same as the last value v_{n-1} returned by the function. It has a single field `last` that gets returned when making a prediction and assigned to when the actual return value is known. In the example, after seeing the sequence $1, 2, 3$, a last value predictor would simply predict 3 as the next output.

Stride (S)

A stride predictor can be seen as a derivative of the last value predictor, computing a prediction based on the sum of the last stride between values and the last value. For instance, upon seeing $1, 2, 3$, it would predict 4 from the last value 3 plus the stride of 1 between 2 and 3. While not completely comparable, this captures most of the same patterns as the last value predictor as well as new ones. In particular, many loop indices and other incrementing or decrementing sequences are easily recognized. Disadvantages are that it takes an extra prediction to warm up, the update and predict operations are somewhat slower, and there is an extra field of storage.

2-delta Stride (2DS)

The 2-delta stride predictor is similar to the stride predictor, imposing the extra constraint that the stride must be the same twice in a row before the predictor updates the stride used to make the prediction. In the example, the stride of 1 detected early in the history is still used to predict 4 even after seeing 3 twice, whereas a simple stride predictor would predict 3 based on the last stride. This design reduces mispredictions by being able to ignore single abberations in a sequence, as can occur in the context of nested loop iterations. In the hardware literature the 2-delta stride predictor has an extra “hysteresis” bit to detect repeats in the stride.

2-Delta Last Value (2DLV)

The 2-delta last value predictor is a new predictor that was suggested by the lack of a corresponding, non-derivative form of the 2-delta stride predictor. A last value approach is used, but the value stored is only updated if the last value is the same twice in a row. For instance, given a sequence such as $1, 1, 2, 3$, the stored last value is not updated during periods of change, and the predicted value will be 1 until the return value again repeats.

In a general sense, the 2-delta pattern can be generalized to arbitrary C-delta predictors, for arbitrary predictors and constant or bound C. Increasing C improves robustness, at a cost of increased warm-up time and larger state.

Last N Value (LNV)

The last N value predictor maintains an N-length history of return values, and uses that list to search for matches to the most recent return value. A match results in a prediction of the next value in the history. This allows the last N value predictor to identify short repeating sequences, capturing simple alternations such as $0, 1, 0, 1, \ldots$, or more complex patterns such as $1, 2, 3, 1, 2, 3, \ldots$, neither of which are ideally predicted by the last value or stride predictors. Our example illustrates the latter case, where assuming $N \geq 3$, a value of 1 is predicted based on the most recent return value of 3 and a history containing a 3 followed by a 1.

Last N value is a generalization of the last value predictor, which may also be expressed as a last 1 value predictor. In their analyses Burtscher and Zorn found that $N = 4$ was a reasonable tradeoff of accuracy against predictor complexity [BZ99a], and so we use this configuration in our experiments.

Last N Stride (LNS)

The last N stride predictor is the corresponding stride version of the last N value predictor, recording a history of strides rather than a value history. It generalizes and subsumes the stride predictor, which can also be considered a last 1 stride predictor.

The example shows a sequence with strides repeating in the pattern $1, 2, 3$. Given the last value of 13, the last stride was 3, which historically was followed by a stride of 1.

Adding 1 to the last value gives a prediction of 14. This example is contrived for purposes of illustration, but repeating stride patterns can occur naturally in several ways, for example by accessing field addresses that have identical offsets in multiple objects.

Finite Context Method (FCM)

To capture more complex historical patterns, the finite context method predictor hashes together a *context*, or recent history of return values of length C. The hashed value is used as hashtable key to index the corresponding return value. This allows for multiple, different patterns to coexist, trading hashing and storage costs for improved accuracy; in the example the pattern $2, 3$ is detected as recurrent and used for the next prediction, despite the existence of other output behaviour before and after. In our suggested implementation the key is stored as a predictor field so that later updates do not have to recompute the hash value, improving performance, although also potentially reducing accuracy.

Hashtable management is a non-trivial concern here: in addition to a good hashing function, table size and growth must be controlled. We use Jenkins' fast hash [Jen97] to compute hashtable keys and power-of-2 sized open addressing tables that use double hashing for lookup in our implementation. At runtime we allow hashtables to dynamically expand up to a maximum table size by doubling in size when 75% full. We experiment with maximum table size in Section 3.4 to assess how accuracy and memory requirements interact, but otherwise use a maximum size of 2^{25} bits, one power-of-2 larger than necessary for all benchmarks. Finally, we use a context length of $C = 5$ in our experiments, which Sazeides and Smith also favoured in their study of finite context method predictors [SS97a].

Differential Finite Context Method (DFCM)

Analogous to the finite context method, the differential finite context method predictor hashes together a recent history of strides rather than values. This is used to look up a stride in a hashtable for adding to the last value in order to make a prediction. The example shows a sequence containing the stride pattern $3, 2$, which is recognized and used to predict the next value of 21. DFCM has the potential to be more space efficient and faster to warm up than the finite context method predictor.

3.2.2 Argument-Based Predictors

Return value prediction accuracy can be improved by taking into account function inputs instead of or as well as function outputs when making a prediction. Tables 3.5 and 3.6 contain the predictors that exploit this information, again separated in terms of normal and derivative forms. In each of these cases the `predict` function now receives the current function arguments as input. In our implementation we disable these predictors for methods that do not take any arguments.

Memoization (M)

The memoization predictor is a new predictor that behaves like the finite context method predictor but hashes together method arguments instead of a recent history of return values. The predictor name comes from the traditional functional programming technique known as memoization, alternatively function caching, that “skips” pure function execution when the arguments match previously recorded $<$arguments, return value$>$ table entries. In our example, the argument pattern of $1, 2, 3$ is hashed together and the key found existing in the hashtable, resulting in a prediction of 4 for the third invocation of f. A key difference from traditional approaches is that memoization based predictions can be incorrect. This means that only the lookup key needs to be stored in the table as opposed to the entire set of arguments. It also makes memoization applicable to all functions that take arguments instead of only the typically much smaller subset of pure, side-effect free functions found in an object-oriented program. The MS predictor is a simple stride derivative, whereas MFCM incorporates value history.

Memoization Stride (MS)

A similar memoization approach can be applied to stride values. Memoization stride stores a stride between return values in its hashtable instead of an actual value, much like the differential finite context method predictor, and adds this value to the last value to make a prediction. The example shows a stride of 3 associated with arguments $1, 2, 3$, resulting in a new prediction of 7 based on the previous value of 4 and the stride found for that argu-

ment pattern. Unlike the differential finite context method predictor, it is not necessarily more space efficient than its non-derivative form, since the set of values used to compute a hashtable key remains the same.

Memoization Finite Context Method (MFCM)

The memoization finite context method predictor is a direct combination of the memoization and finite context method predictors. It concatenates the recent history of return values with the function arguments and uses the result to compute a hash value for table lookup. This is significantly more expensive than either memoization or finite context method predictors, but has the potential to capture complicated patterns that depend on both historical output and current input. The example shows a context of length 2, recognizing the output sequence $5, 6$ followed by an argument of 3, which leads to predicting the previously seen value of 7. In comparison, a pure memoization predictor would predict 9 here from the prior $<$argument, return value$>$ pair given by $f(3) = 8$, and a pure FCM predictor would return 8 due to the preceding output sequence of $5, 6, 8$. Note that a differential version of the memoization finite context method predictor would naturally follow from our framework; instead we investigated the parameter stride predictor, as shown in Table 3.6.

Parameter Stride (PS)

The parameter stride predictor identifies a constant difference between the return value and one parameter, and uses this to compute future predictions. A simple example of a function it captures is one that converts lowercase ASCII character codes to alphabet positions. Although the parameter stride predictor is in general subsumed by the memoization predictor, parameter stride is simpler in implementation, warms up very quickly, and requires only constant storage.

3.2.3 Composite Predictors

Table 3.7 contains predictors that are composites of one or more sub-predictors. The hybrid predictor uses the other predictors directly, returning a prediction by the best performing

sub-predictor, whereas composite stride is in fact a generalized pattern for creating other predictors.

Hybrid (H)

The hybrid predictor is composed of one of each kind of sub-predictor. To make a prediction, it first obtains a prediction from each sub-predictor and records this value. It then returns the prediction of the predictor with the highest accuracy, favouring the earliest sub-predictor in the event of a tie. In our implementation we keep track of accuracy over the last n values, where n is the number of bits in a word; $n = 64$ on our `x64_64` machines. This allows sub-predictors with locally good but globally poor accuracies to be chosen by the hybrid. To update the hybrid, for each such sub-predictor `update` is called, the actual return value is compared against the predicted return value, and the accuracy histories are updated accordingly. The accuracy of the hybrid itself can be used as an explicit *predictor confidence* input to priority computation and thus dynamic fork heuristics, as discussed in Section 2.4.3.

Composite Stride

The composite stride predictor is not an individual predictor but rather a generalized implementation pattern for constructing stride predictors. A composite stride simply contains another predictor that it will use to predict a stride value, and adds that to the previous return value. Each predictor at the bottom of Tables 3.1–3.4 as well as the memoization stride predictor in Table 3.5 can be alternatively constructed as a composite stride predictor containing the corresponding predictor at the top. In our implementation we applied this pattern to implement all stride predictors except the parameter stride predictor, which does not follow this pattern because it predicts a constant difference between the return value and one parameter. This object-oriented simplification was only realized once we expressed the predictors using this framework.

Last Value [Gab96] – LV

$$v_n = v_{n-1}$$

Predicts using the last value.

example: $1, 2, 3 \rightarrow 3$

fields: `last`

predict`()`:

```
  return last;
```

update`(value_t rv)`:

```
  last = rv;
```

Stride [Gab96] – S

$$v_n = v_{n-1} + (v_{n-1} - v_{n-2})$$

Predicts using the difference between the last two values.

example: $1, 2, 3 \rightarrow 4$

fields: `last, stride`

predict`()`:

```
  return last + stride;
```

update`(value_t rv)`:

```
  stride = rv - last;
  last = rv;
```

Table 3.1: *History-based predictors I.*

2-Delta Last Value (new) – 2DLV

$v_n = v_{n-i},$ where i is the min i s.t.
$v_{n-i} = v_{n-i-1}$
or v_{n-1} if no such i exists

LV that only updates if the last value is the same twice in a row.

example: $1, 1, 2, 3 \rightarrow 1$

fields: `last1, last2`

predict`()`:

```
return last2;
```

update`(value_t rv)`:

```
if (rv != last1) last1 = rv;
else last2 = rv;
```

2-Delta Stride [SS97b] – 2DS

$v_n = v_{n-1} + v_{n-i} - v_{n-i-1},$ where i is the min i s.t.
$v_{n-i} - v_{n-i-1} = v_{n-i-1} - v_{n-i-2}$
or v_{n-1} if no such i exists

S that only updates if the stride is the same twice in a row.

example: $1, 2, 3, 3 \rightarrow 4$

fields: `last, stride1, stride2`

predict`()`:

```
return last + stride2;
```

update`(value_t rv)`:

```
if (rv - last != stride1) stride1 = rv - last;
else stride2 = rv - last;
last = rv;
```

Table 3.2: *History-based predictors II.*

Last N Value [LS96, BZ99a] – LNV

$$v_n = v_{n-i},\ \text{where } i \leq N \text{ is the min } i \text{ s.t. } v_{n-1} = v_{n-i-1} \text{ or } v_{n-1} \text{ if no such } i \text{ exists}$$

Predicts using the value at some position in the last N values.

example: $1, 2, 3, 1, 2, 3 \rightarrow 1$

fields: `values[N], last_correct_pos`

predict`():`

```
return values[last_correct_pos];
```

update`(value_t rv):`

```
last_correct_pos = contains (values, rv) ?
  index_of (rv, values) :  1;
shift_into (values, rv);
```

Last N Stride [ZFC03] – LNS

$$v_n = v_{n-1} + (v_{n-i} - v_{n-i-1}),\ \text{where } i \leq N \text{ is the min } i \text{ s.t. } v_{n-1} - v_{n-2} = v_{n-i-1} - v_{n-i-2} \text{ or } v_{n-1} - v_{n-2} \text{ if no such } i \text{ exists}$$

Predicts using the stride at some position in the last N strides.

example: $1, 2, 4, 7, 8, 10, 13 \rightarrow 14$

fields: `last, strides[N], last_correct_pos`

predict`():`

```
return last + strides[last_correct_pos];
```

update`(value_t rv):`

```
last_correct_pos = contains (strides, rv - last) ?
  index_of (rv - last, strides) :  1;
shift_into (values, rv - last);
```

Table 3.3: *History-based predictors III.* `contains (a[], v)` returns true if array `a[]` contains value `v`, whereas `index_of (v, a[])` returns the position of `v` in `a[]`. `shift_into (a[], v)` adds `v` to `a[]`, shifting all other elements down and removing the oldest element.

Finite Context Method [SS97b, SS97a] – FCM

$$v_n = v_{n-i},\ \text{where } i \text{ is the min } i \text{ s.t. } v_{n-c} = v_{n-i-c},\ \text{for all } c \leq C \text{ or } 0 \text{ if no such } i \text{ exists}$$

Captures value history patterns of length $C + 1$.

example: $1, 7, 2, 3, 8, 4, 7, 2 \rightarrow 3$ for $C = 2$

fields: `key, context[C]`

predict`()`:

```
key = hash (context);
return lookup (key);
```

update`(value_t rv)`:

```
store (key, rv);
shift_into (context, rv);
```

Differential Finite Context Method [GVdB01] – DFCM

$$v_n = v_{n-1} + (v_{n-i} - v_{n-i-1}),\ \text{where } i \text{ is the min } i \text{ s.t. } v_{n-c} - v_{n-c-1} = v_{n-i-c} - v_{n-i-c-1},\ \text{for all } c \leq C \text{ or } 0 \text{ if no such } i \text{ exists}$$

Captures stride history patterns of length $C + 1$.

example: $1, 6, 9, 11, 16, 19 \rightarrow 21$ for $C = 2$

fields: `last, key, context[C]`

predict`()`:

```
key = hash (context);
return last + lookup (key);
```

update`(value_t rv)`:

```
store (key, rv);
shift_into (context, rv - last);
```

Table 3.4: *History-based predictors IV.* `hash (a[])` produces a hashtable key from the values in `a[]`; `lookup (key)` returns the hashtable value at index `key`; and `store (key, v)` stores value `v` at index `key`.

Memoization (new) – M

$v_n = v_{n-i},$ where i is the min i s.t.
$\mathit{args}(n) = \mathit{args}(n-i)$, or 0 if no such i exists

Maps function arguments to return values.

example: $f(1,2,3) = 4,\ f(4,5,6) = 7,\ f(1,2,3) \rightarrow 4$

fields: `key`

predict`(value_t args[]):`

```
key = hash (args);
return lookup (key);
```

update`(value_t rv):`

```
store (key, rv);
```

Memoization Stride (new) – MS

$v_n = v_{n-1} + (v_{n-i} - v_{n-i-1}),$ where i is the min i s.t.
$\mathit{args}(n) = \mathit{args}(n-i)$, or 0 if no such i exists

Maps function arguments to return value strides.

example: $f(1,2,3) = 4,\ f(1,2,3) = 7,\ f(1,2,3) \rightarrow 10$

fields: `key, last`

predict`(value_t args[]):`

```
key = hash (args);
return last + lookup (key);
```

update`(value_t rv):`

```
store (key, rv);
last = rv;
```

Table 3.5: *Argument-based predictors I.*

Memoization Finite Context Method (new) – MFCM

$v_n = v_{n-i}$, where i is the min i s.t.
$v_{n-C} = v_{n-i-C}$, for all $c \leq C$, and
$\mathit{args}(n) = \mathit{args}(n-i)$, or 0 if no such i exists

Maps function arguments × value history to return values.

example: $f(1)=5,\ f(2)=6,\ f(3)=7, f(3)=9,\ f(1)=5,$
$f(5)=6,\ f(5)=8,\ f(1)=5,\ f(2)=6,\ f(3) \rightarrow 7$ for $C=2$

fields: `key, context[C]`

predict`(value_t args[]):`
```
key = hash (concat (args, context));
return lookup (key);
```
update`(value_t rv):`
```
store (key, rv);
shift_into (context, rv);
```

Parameter Stride [HBJ03] – PS

$v_n = \mathit{args}(n)[a] + (v_{n-i} - \mathit{args}(n-i)[a])$, where i is the min i s.t.
$v_{n-i} - \mathit{args}(n-i)[a] = v_{n-i-1} - \mathit{args}(n-i-1)[a]$
for some argument index a, or 0 if no such i exists

Identifies a constant offset between one parameter and the return value.

example: $f(\text{‘}r\text{’}) = 17,\ f(\text{‘}v\text{’}) = 21,\ f(\text{‘}p\text{’}) \rightarrow 15$

fields: `a = A, old_args[A], strides[A]`

predict`(value_t args[]):`
```
copy_into (old_args, args);
return a < A ? args[a] + strides[a] :  0;
```
update`(value_t rv):`
```
for (i = A - 1; i >= 0; i--)
  if (rv - old_args[i] == strides[i]) a = i;
  strides[i] = rv - old_args[i];
```

Table 3.6: *Argument-based predictors II.* `concat (a[], b[])` returns arrays `a[]` and `b[]` concatenated into a single array. `copy_into (a[], b[])` copies the contents of `b[]` into `a[]`.

Hybrid [BZ02] (new design) – H

$$v_n = f(v_1, \ldots, v_{n-1}, \mathit{args}(n)),$$
where f is the best performing sub-predictor

Combines many different sub-predictors and identifies the best one.

fields: `predictors[], accuracies[], predictions[]`

predict`(value_t args[]):`
```
  for (p = 0; p < P; p++)
    predictions[p] = predictors[p].predict (args);
  return predictions[max_index (accuracies)];
```
update`(value_t rv):`
```
  for (p = 0; p < P; p++)
    predictors[p].update (rv);
    accuracies[p] = (rv == predictions[p]) ?
      min (accuracies[p] + 1, 64) :
      max (accuracies[p] - 1, 0);
```

Composite Stride (new) – CS

$$s_{n-i} = v_{n-i} - v_{n-i-1},\ \forall\, 2 \leq i < n$$
$$s_{n-1} = f(s_1, \ldots, s_{n-2}, \mathit{args}(n-1)),$$
where f is any sub-predictor
$$v_n = v_{n-1} + s_{n-1}$$

Creates a stride derivative of any other predictor.

fields: `last, f`

predict`():`
```
  return last + f.predict ();
```
update`(value_t rv):`
```
  f.update (rv - last);
  last = rv;
```

Table 3.7: *Composite predictors.* Our software hybrid design is new, but conceptually similar to hardware hybrid designs. The composite stride predictor is a general implementation pattern for converting value predictions into stride predictions, rather than a specific predictor.

3.3 Experimental Setup

The first step in our experimental approach was to create an object-oriented C implementation of every predictor described in Section 3.2. For ease of experimentation, this work was done inside libspmt, but it would be reasonably straightforward to refactor the RVP code into a separate value prediction library. Like the rest of libspmt, the RVP code is open source, portable, and modular, and the predictors are all supported by unit tests that check for expected predictor behaviour. It currently runs on `x86_64` and `ppc64` architectures. It also includes profiling support, which we used to generate the raw data for our experimental results. We then modified SableSpMT to communicate directly with the RVP code, bypassing the method level speculation fork and join interface. Below we describe the basic system structure and client configuration, followed by initial analysis of our benchmarks and overhead costs. Sections 3.4 and 3.5 provide more detailed experimentation on individual and hybrid predictors respectively.

Figure 3.3 gives an overview of the general RVP library structure and client–library communication process. At the library core is a map between physical callsite addresses and callsite probe objects. Each probe contains a hybrid predictor instance as well as callsite identification and profiling information. When the VM client allocates a non-void callsite during method preparation, it sends the callsite address, class, method, program counter, and target method descriptor to the library in exchange for a reference to a callsite probe object. This reference is used for all subsequent communication to avoid unnecessary table lookups.

We modified the VM to call `predict` and `update` RVP functions before and after non-void callsite execution respectively. The former takes method arguments, including any implicit `this` reference, and returns a predicted value, whereas the latter takes the actual return value and updates the predictors associated with the callsite. In the event of escaping exceptions, no update occurs. To minimize VM changes, the library parses arguments from the VM call stack using the target descriptor, zeroing out unused bytes and arranging the arguments contiguously in memory. Internally, the hybrid and all sub-predictors subclass a predictor class with `update` and `predict` methods. This design allows for easy composition and hybrid specialization, as described in Section 3.5.

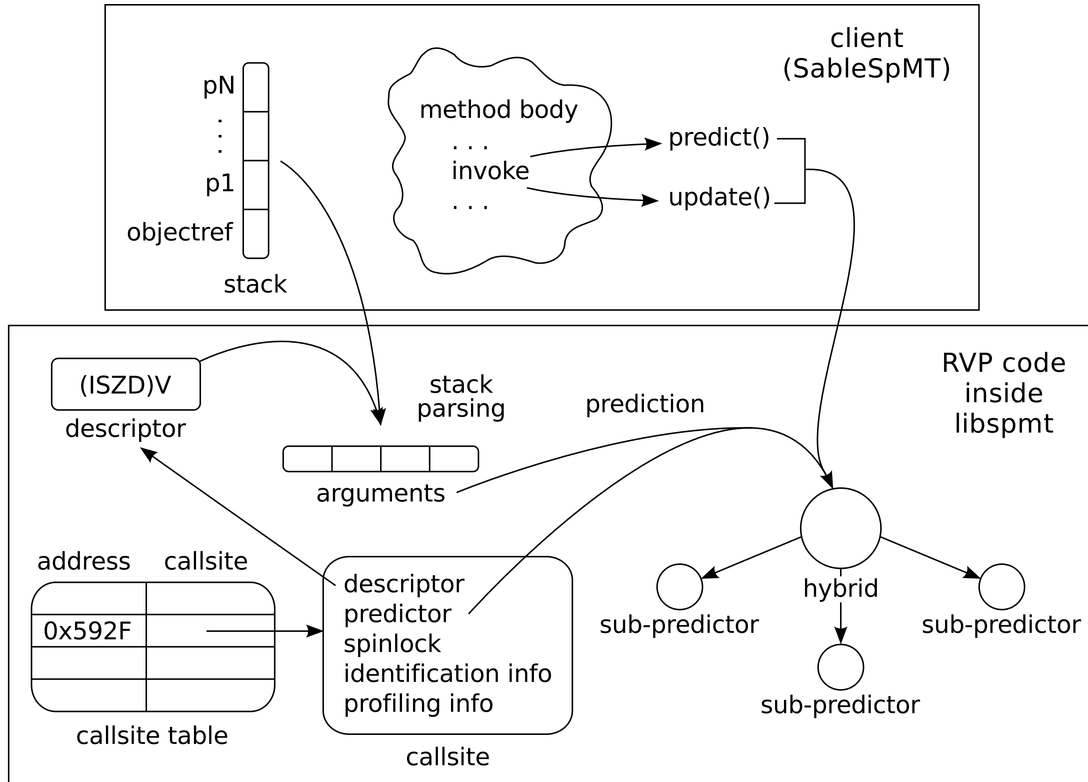


Figure 3.3: *Client–library communication.*

3.3.1 Benchmarks

As in Chapter 2, we used the SPEC JVM98 benchmarks with input set S100 for experimental evaluation [Sta98]. These benchmarks are not as complex or memory-intensive as the more recent DaCapo benchmarks [BGH$^+$06]. However, they are fast to execute, an important factor in performing a large number of experiments, and more than sufficient for a software RVP study as they use over 800 million non-void method calls in the absence of method inlining. Our choice of benchmark suite also directly extends previous work on RVP for Java, which used the same benchmarks but alternatively ran only the tiny S1 dataset in a restricted hardware context that only considered boolean, int, and reference return types [HBJ03] or ignored specific predictor behaviour [SB06]. It also extends the work in Chapter 2 that focuses on our MLS client application of the results.

benchmark	comp	db	jack	javac	jess	mpeg	mtrt
methods	670	714	936	1.51K	1.15K	838	863
callsites	2.48K	2.79K	4.56K	7.20K	4.32K	2.94K	3.71K
invokes (`V`)	93.4M	54.4M	35.0M	39.9M	23.3M	45.2M	28.4M
invokes (`NV`)	133M	116M	62.9M	82.3M	102M	65.8M	259M
escapes (`V`)	0	0	608K	0	0	0	0
escapes (`NV`)	0	0	68	41.5K	0	0	0
returns (`V`)	93.4M	54.4M	34.4M	39.9M	23.3M	45.2M	28.4M
returns (`NV`)	133M	116M	62.9M	82.3M	102M	65.8M	259M
booleans `Z`	6.70K	11.1M	17.3M	19.5M	35.8M	13.2M	3.07M
bytes `B`	0	0	580K	39.3K	0	0	0
chars `C`	8.85K	25.2K	8.53M	3.80M	24.4K	6.96K	20.8K
shorts `S`	0	0	0	73.0K	0	18.0M	0
ints `I`	133M	48.1M	17.9M	35.9M	20.7M	34.6M	4.54M
longs `J`	440	152K	1.23M	818K	100K	15.7K	2.07K
floats `F`	102	704	296K	104	1.04K	7.82K	162M
doubles `D`	0	0	0	160	1.77M	56	214K
references `R`	17.0K	56.2M	17.0M	22.2M	43.5M	24.3K	89.6M

Table 3.8: *Benchmark properties.* `V`: void; `NV`: non-void; escapes: escaping exceptions.

Table 3.8 presents relevant benchmark properties. The first section shows the number of methods and callsites in the dynamic call graph. In principle, we can associate predictors with methods, callsites, or the invocation edges that join them. We choose here to use callsites exclusively, mostly to limit the scope of our evaluation. Callsites seem like a reasonable choice because they capture the calling context without being type sensitive. In future work, it would be interesting to study how performance differs when methods or invocation edges are used instead.

The second section shows dynamic void and non-void invokes, escapes, and returns. An invoke is a method call, a return is normal method completion, and an escape is abnormal method termination due to an uncaught exception in the callee. We exclude void method calls from our analysis because they do not return values, but present them here for the sake

of completeness. We make predictions on all non-void invokes, but only send updates on normal returns, because for escapes there is no return value and control does not return to the callsite. We thus report accuracy measures over the total number of non-void returns. As the data show, escaping exceptions are relatively rare, even for supposedly exception-heavy benchmarks such as `jack`, which means they do not have a large impact in any case.

The third section classifies non-void returns according to the eight Java primitive types and also reference types. Return type information is interesting because some types are inherently more predictable than other types, suggesting specialization and compression strategies, and because it describes behaviour to some extent. In our initial return value prediction study, we found that boolean, byte, and char methods were highly predictable, whereas the remaining types had a predictability that ranged from low to high [PV04b]. In other words, the distribution and sequence of values matters more than the type, except where the type constrains the distribution by nature of its short width or typical value range. We see that `mtrt` relies heavily on float methods, `mpegaudio` uses a surprising number of methods returning shorts, `compress` returns almost exclusively ints, and the remaining benchmarks use more or less equal mixes of int, boolean, and reference calls.

3.3.2 Communication Overhead

Our design emphasizes modularity and ease of experimentation over performance. The use of an external library, multiple calls, portable argument parsing, and so forth has an obvious performance impact, much of which could be ameliorated by incorporating the RVP code directly into the VM, interleaving RVP code in generated code in the case of a JIT compiler client, and generally optimizing its performance along with other VM activities. We thus performed an initial experiment to isolate and measure the overhead of our framework.

Figure 3.4 shows the slowdown due to communication overhead with the predictor module of the library. These data are gathered using a *null* predictor that simply returns zero for every prediction, and performs no actual update computation. In future experiments we control for this overhead by using the null predictor performance results as a baseline. The large slowdown for `mtrt` is due to contention in our simple predictor locking strategy and a

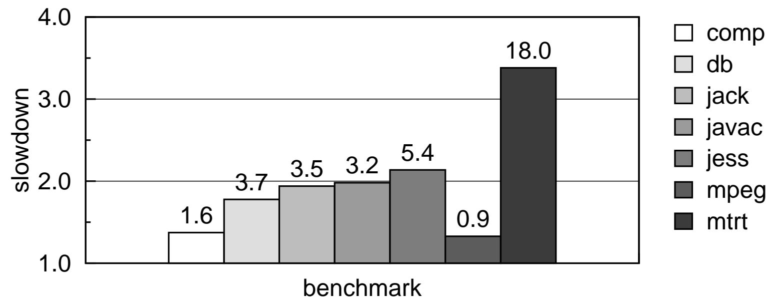


Figure 3.4: *Null predictor slowdowns, relative to vanilla SableVM.* Above each bar is the vanilla SableVM non-void invoke density for that benchmark in terms of millions of calls per second.

high call density. Improved lock-based or even lock-free designs would help, and in general multithreaded predictor interactions are an interesting direction for future work. Overhead scales primarily with call density, but the cost of argument parsing does make some calls more expensive than others. In practice, performance critical applications should tailor their use of RVP to the locations where it is most useful.

3.4 Initial Performance Evaluation

We used our software library implementation of the predictors in Section 3.2 to measure their accuracy, speed, and memory consumption performance over our benchmark suite. Knowing the specific performance characteristics of individual predictors is useful when given a constrained resource budget. We expect the more complex predictors to have better accuracy but with higher speed and memory costs. The naïve hybrid predictor we study here does not specialize, visiting every sub-predictor on each call to `predict` and `update`. The next section contains a detailed exploration of hybrid adaptivity.

It is important to keep in mind while considering these results that a prediction is made for every single invocation in the program and that there is no inlining. We chose this approach to gather the most comprehensive set of data possible and to make our study generally useful, because different clients of RVP will invariably make different decisions about where to predict. Individual callsite prediction accuracies and overhead costs differ widely, which means that disabling prediction selectively can significantly affect the results. The actual runtime speed and memory costs in any practical scenario will scale with usage.

3.4.1 Accuracy

Figure 3.5 shows basic prediction accuracies for each predictor and for each benchmark. Accuracy is calculated as the number of correct predictions over the number of non-void calls that returned to their callsite. The benchmarks are clustered in alphabetical order from left to right for each predictor. The predictors are arranged in the top-to-bottom order given by Tables 3.1–3.7, excluding the final composite stride pattern for constructing predictors. For comparison we include our baseline null predictor (N) that simply returns 0.

As expected, the hybrid beats individual predictor accuracies for every benchmark because it allows sub-predictors to complement each other. Accuracy otherwise scales roughly with complexity, at least for the non-memoization predictors. A basic last value predictor significantly improves on a null predictor, is in turn improved on by last N predictors, and which themselves are overshadowed by context-based designs. Interestingly the stride versions of non-context predictors do not show significant differences from the last value predictors, suggesting that extending the predictors to higher level derivative forms does not necessarily improve accuracy. Including value history context has a significant impact. The finite context method and its differential form have the highest individual predictor accuracies, and even memoization is noticeably improved by adding value history. Notably, the accuracy of the DFCM predictor is within 5% of the hybrid. Argument based approaches are not as successful as history based approaches in isolation, but as we show later memoization can complement the FCM and DFCM predictors nicely in a hybrid. In summary, highly predictable computations fall into one of three categories: those where the return value fits some function of recent inputs and outputs (fixed-space predictors), those that exhibit input repetition (memoization-based predictors), and those that exhibit output repetition (history-based predictors).

Interesting differences also show up in terms of benchmark behaviour. `db`, `jack`, `javac`, and `jess` respond well overall, with even simple predictors reaching 40–60% accuracy levels. `mpegaudio` and `mtrt` are more resilient to prediction, due to their use of irregular short and floating point types respectively. `compress` improves dramatically with table-based prediction, indicating longer term patterns exist, even if `mpegaudio` and `compress` are naturally expected to be less predictable since they handle compressed data.

3.4.2 Speed

Figure 3.6 shows slowdowns due to predictor overhead for each predictor and for each benchmark. Slowdown is calculated as predictor performance relative to the null predictor, factoring out any overhead inherent in our experimental setup, per the control experiment in Figure 3.4. The graph is structured similarly to Figure 3.5, although on a logarithmic scale and without the null predictor. As expected, predictor speeds vary with complexity, with the table-based predictors being considerably slower than the fixed-space predictors. The table-based predictors are expensive for two reasons. First, hashing arguments or return value histories to table lookup keys is an expensive operation. Second, the memory requirements of the larger tables introduce performance penalties due to memory hierarchy latencies. The naïve hybrid is unsuprisingly very slow, incurring the summed cost of all sub-predictors. When compared against the naïve hybrid, the DFCM predictor alone is an obviously better choice: it is at least three times as fast, its accuracy is at most 5% worse (Figure 3.5), and it typically requires about half as much memory (Table 3.9). However, per Figures 3.17, 3.18, and Table 3.10 in the next section, by substituting less expensive predictors when feasible, adaptive versions of the hybrid predictor can outperform not only the naïve hybrid but also DFCM for speed and memory while maintaining similar accuracy.

3.4.3 Memory Consumption

The memory consumption of each predictor for each benchmark is shown in Table 3.9. The memory requirements of the fixed-space predictors are calculated by summing the number of bytes used by each predictor and multiplying by the number of callsites. The table-based predictor memory requirements are calculated in the same manner for the fixed-space fields, and then the actual final sizes of the hashtables at individual callsites upon program completion are used to calculate the variable-sized fields. The main observation here is that the table-based predictors can consume large amounts of memory. This effect is compounded in the hybrid that has five table-based sub-predictors at each callsite, in the worst case reaching a huge 6.37G for `mtrt`. These data confirm that memory latencies are likely to contribute to predictor slowdowns for table-based prediction.

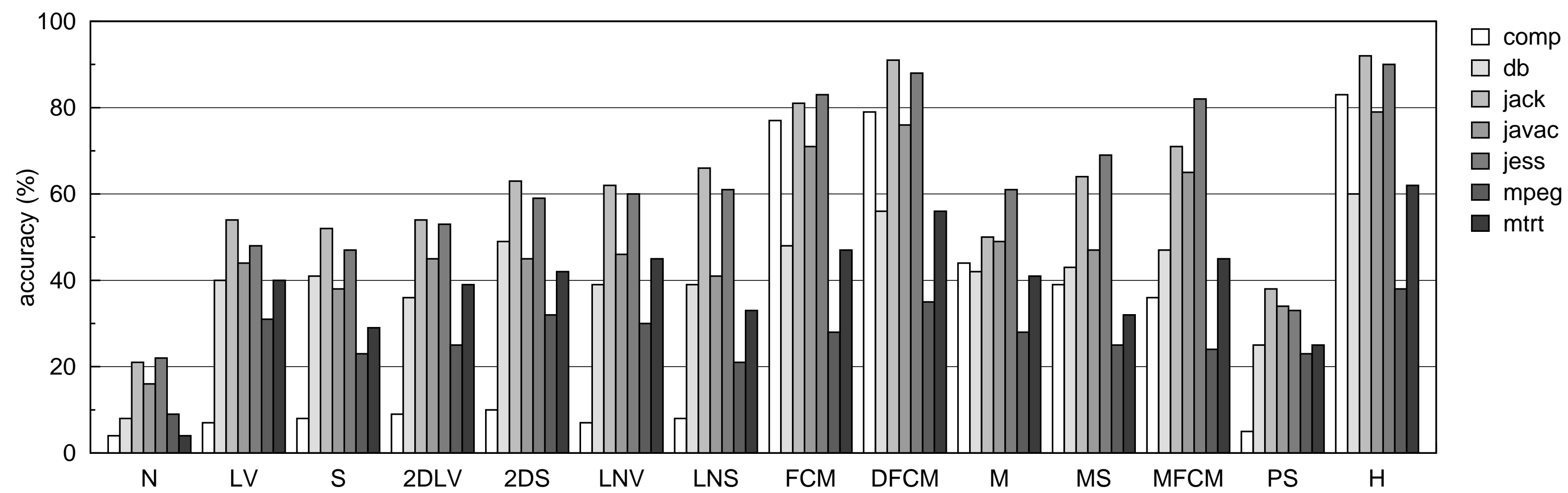


Figure 3.5: *Predictor accuracies for a null predictor (N) and all predictors in Tables 3.1–3.7.*

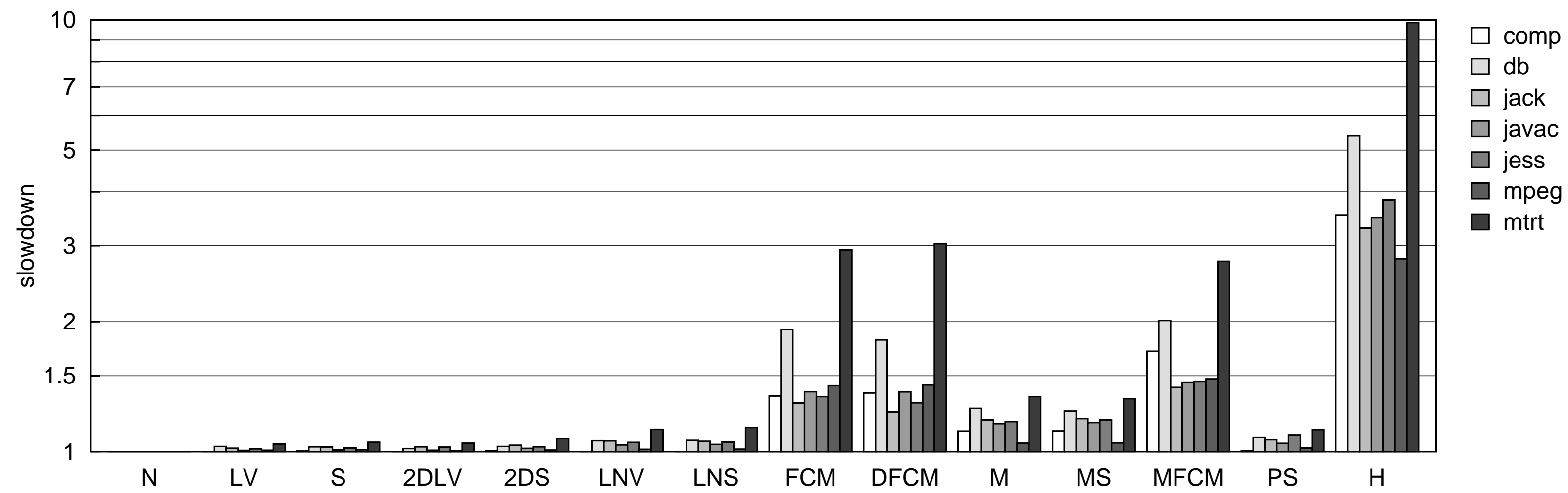


Figure 3.6: *Predictor slowdowns for a null predictor (N) and all predictors in Tables 3.1–3.7, relative to N.*

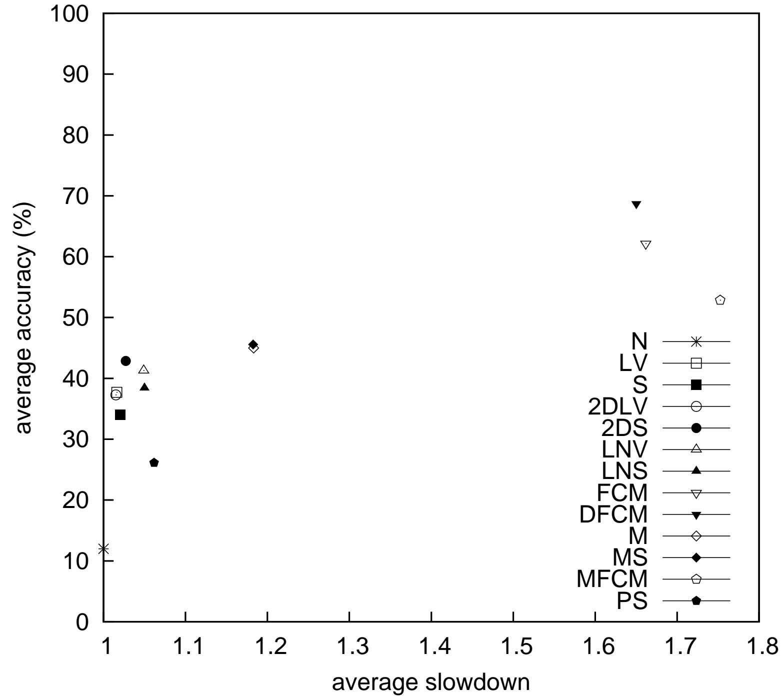


Figure 3.7: *Average sub-predictor accuracy vs. average slowdown.*

predictor	comp	db	jack	javac	jess	mpeg	mtrt
N	4.67K	5.23K	10.5K	20.9K	10.1K	6.08K	11.0K
LV	9.34K	10.5K	21.0K	41.7K	20.2K	12.2K	21.9K
S	18.7K	20.9K	42.0K	83.4K	40.4K	24.3K	43.9K
2DLV	14.0K	15.7K	31.5K	62.6K	30.3K	18.2K	32.9K
2DS	23.4K	26.1K	52.5K	104K	50.5K	30.4K	54.8K
LNV	23.9K	26.8K	53.8K	107K	51.7K	31.2K	56.2K
LNS	33.3K	37.2K	74.8K	149K	71.9K	43.3K	78.2K
FCM	625M	0.97G	50.7M	205M	14.6M	1.61G	2.97G
DFCM	673M	784M	7.26M	197M	10.1M	1.60G	3.31G
M	6.81M	99M	7.75M	1.51M	4.03M	25.4M	7.19M
MS	6.82M	99M	7.77M	1.55M	4.05M	25.5M	7.21M
MFCM	31.1M	893M	16.6M	4.79M	13.4M	1.72G	80.6M
PS	12.4K	13.8K	29.5K	59.6K	26.9K	16.2K	28.0K
H	1.31G	2.80G	90.9M	411M	47.1M	4.98G	6.37G

Table 3.9: *Predictor memory consumption.*

The data in Table 3.9 and Figures 3.5 and 3.6 assume hashtable sizes are unbounded, and so the tables grow as necessary to accommodate new values. This is obviously unrealistic, but if the sizes are bounded then new values overwrite old values once the maximum size is reached, which reduces overall accuracy if the old value is ever requested. We thus explored predictor accuracy as a function of maximum table size, as shown in Figures 3.8–3.13. Here maximum table sizes are varied from 2^0 to 2^{25} entries, one power of 2 larger than the largest size any predictor was observed to expand to naturally, and accuracy examined for each table predictor and benchmark combination. In general, accuracy increases as table size increases, although only up to a point. After this point accuracy remains mostly constant, indicating no further impact from collisions, and in some cases may actually decrease due to the absence of lucky collisions that returned a correct value at smaller sizes.

Figures 3.8–3.13 also indicate that individual predictors can have complex interactions in a hybrid. For a given benchmark and table size, individual predictors often have noticeably different performance: memoization (stride) may work well in some instances whereas the (differential) finite context method works well in others. Interestingly, although the context predictors usually have the highest accuracies, the predictor complementation provided by the hybrid predictor can be seen in the shapes of the curves for individual benchmarks. The hybrid behaviour for `compress`, `jack`, `javac`, and `jess`, for example, combines the better accuracy of M(S) designs at low table sizes with the higher accuracy of (D)FCM at higher sizes.

3.4.4 Sub-Predictor Comparisons

The naïve hybrid predictor in Figures 3.5, 3.6, and Table 3.9 has an average accuracy of 72%, an average slowdown of 4.6x, and an average memory consumption of 2.3G. This is clearly unusable, but could be made much better by simply limiting the number of sub-predictors. Figure 3.7 shows the average accuracy for each sub-predictor over SPEC JVM98 plotted against its average slowdown. If choices are limited, the three clear winners here are the 2-delta stride (2DS), memoization stride (MS), and differential finite context method (DFCM) predictors, where increased accuracy is traded for increased slowdown and memory consumption. Although 2DS is only slightly worse in terms of accuracy than

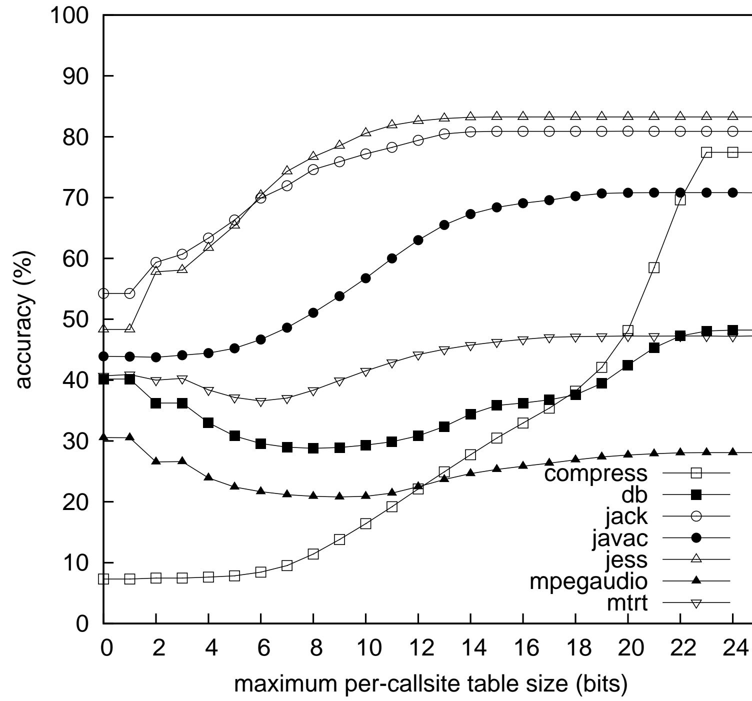


Figure 3.8: *Finite context method (FCM) accuracy vs. maximum table size.*

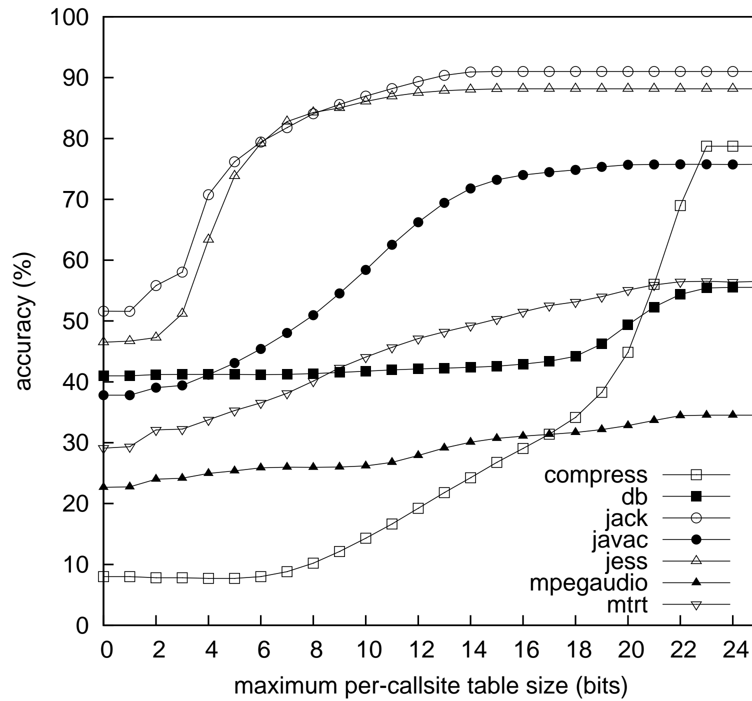


Figure 3.9: *Differential finite context method (DFCM) accuracy vs. maximum table size.*

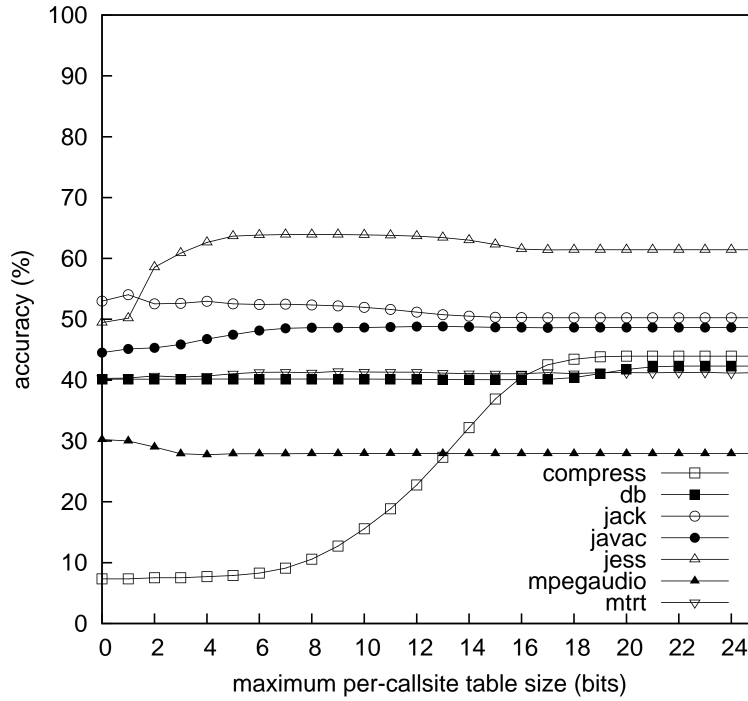


Figure 3.10: *Memoization (M) accuracy vs. maximum table size.*

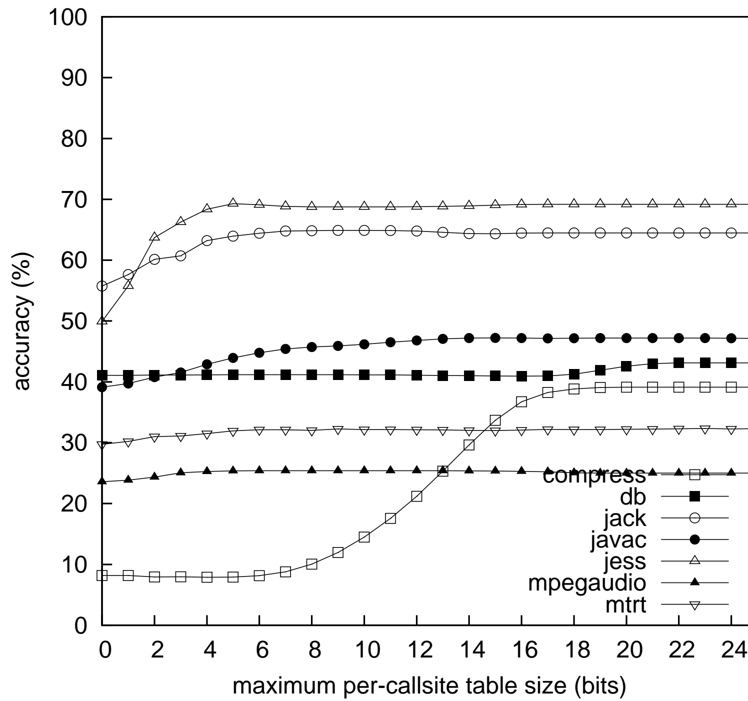


Figure 3.11: *Memoization stride (MS) accuracy vs. maximum table size.*

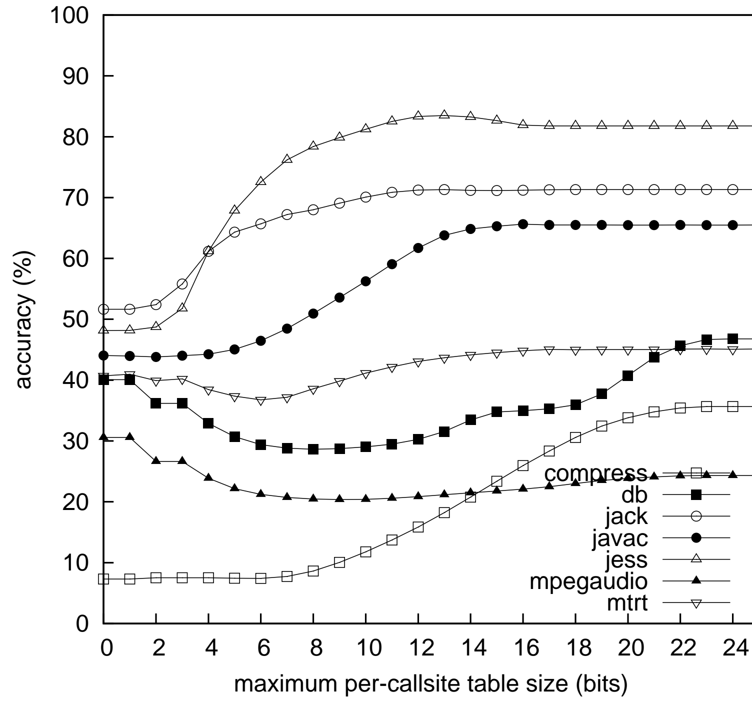


Figure 3.12: *Memoization finite context method (MFCM) accuracy vs. maximum table size.*

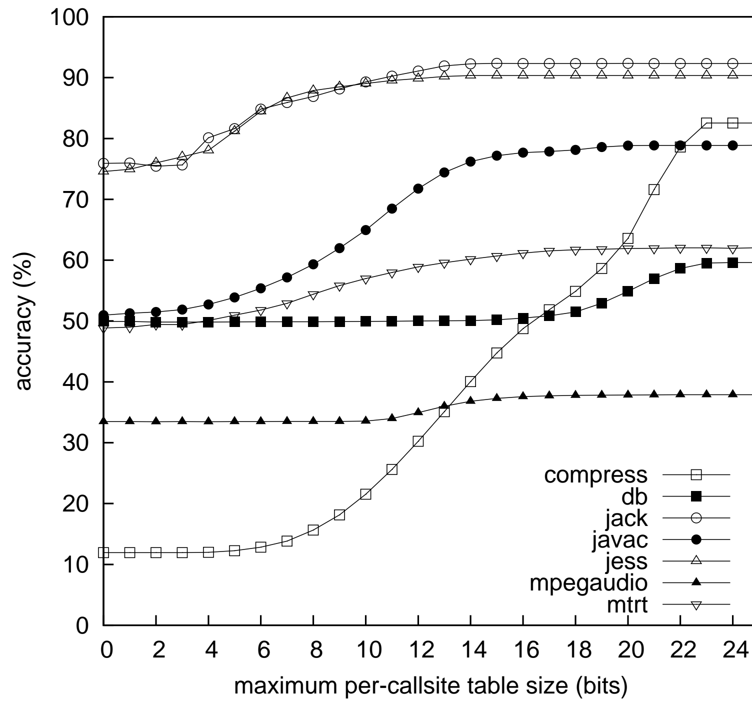


Figure 3.13: *Hybrid (H) accuracy vs. maximum table size.*

MS and significantly better in terms of slowdown and memory consumption, it occupies a different algorithmic space and will end up complementing MS in a hybrid. These three predictor types correspond to the three types of highly predictable computations we previously highlighted, namely those that fit some function (2DS), those with input repetition (MS), and those with output repetition (DFCM). Note that despite this analysis, we still chose to use all 13 sub-predictors in the adaptive hybrid experiments that follow. One reason is that we wanted to demonstrate that our adaptive hybrid design can accomodate a large number of sub-predictors, which is true at least in theory because only one remains in use after specialization. Another reason is that even the less robust predictors in Figure 3.7 may contribute uniquely accurate predictions in some situations. Nevertheless, a systematic exploration of potential hybrid compositions would likely find that rejecting certain sub-predictors yields even better performance. We leave this as an opportunity for future work.

3.5 Hybrid Adaptivity

The naïve hybrid design in Table 3.7 achieves very high accuracy. However, its speed suffers because it employs twelve different sub-predictors in series to make and update predictions, and its memory consumption suffers because it retains the memory for large table-based predictors even if they are never selected for prediction. We would like to maintain this high accuracy while optimizing for speed and memory consumption. We do this by specializing individual hybrid instances to particular sub-predictors and releasing the resources required by the other unused sub-predictors. This optimization relies on an important hypothesis: *for a given callsite, there is likely to be an ideal sub-predictor*.

We first tested this hypothesis with an offline profiling based experiment to identify ideal sub-predictors on a per-callsite basis. The ideal sub-predictor for a callsite is simply the one that performed best over the entire course of execution. If a subsequent run in which the hybrid immediately specializes to these predictors matches the accuracy of the naïve version, then it indicates that ideal sub-predictors are likely to exist. The performance of this offline hybrid can then provide an oracle for online optimization.

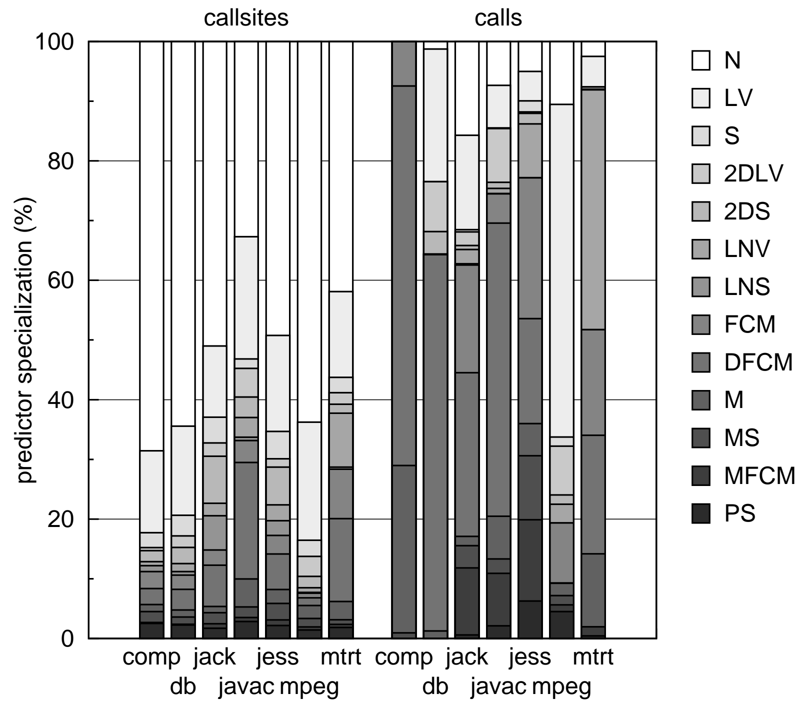


Figure 3.14: *Ideal predictor distributions.*

3.5.1 Offline Specialization

We first ran each benchmark to completion using the naïve predictor, and processed the results to create a profile for offline specialization. Figure 3.14 shows the distribution of ideal predictors for each benchmark in terms of dynamically reached callsites and the number of dynamic calls. At the callsite level, most ideal predictors are null or last value predictors. In this analysis, cold callsites with one call are weighted equally with hot callsites that have 50 million calls, and they tend to specialize to simple predictors. Most of these cold callsites are found in initialization code, and there is simply no chance for sufficient history to develop such that the more complex predictors outperform the simple ones.

At the level of actual calls, the simple predictors still work well in many cases, particularly for methods returning constants or accessing static data structures. However, hot callsites tend to benefit from complex table-predictors predictors, indicating an important role for them in maximizing accuracy. This reconfirms the result in Figure 3.13, where a low cap on table size in the hybrid predictor can suppress accuracy significantly. `mpegaudio`

provides a notable exception to the dominance of table predictors. It decodes an mp3 file, and so its return values are mostly random. It has very low overall predictability, and when there is repetition it is generally found in the last few values, meaning that simple predictors dominate.

3.5.2 Online Specialization

We next attempted to determine ideal sub-predictors dynamically, without ahead-of-time profiling data. Online adaptivity is critical in dynamic compilation environments, where ahead-of-time techniques are not well accepted in practice. In this case online specialization can also accommodate callsites that exhibit phase-like behaviour, where the ideal sub-predictor is not constant throughout the program run.

There are three basic parameters we considered in constructing our online specializing hybrid. The first is a warmup period, w. The hybrid predictor will not specialize until $u \geq w$, where u is the number of predictor updates. The second is a confidence threshold for specialization, s. For the number of correct predictions c over the last n calls, if $c \geq s \wedge u \geq w$ then the hybrid specializes to the best performing sub-predictor, favouring cheaper predictors in the event of ties. We use a value of $n = 64$, the number of bits in a word on our machines. The third parameter is a confidence threshold for despecialization, d. If $c < d$ and the hybrid has already specialized, then it will despecialize again. We did not experiment with resetting the warmup period upon despecialization, although this could be a useful extension.

We performed a parameter sweep over w, s, d according to Figure 3.15. This generated 405 different experiments. For each, the average accuracy and slowdown were computed. The average accuracies were rounded to the nearest integer, and the minimum running time for each accuracy identified. These results are shown in Figure 3.16. From these data, we selected the point at accuracy 67% with slowdown of 1.35x for use in future experiments. This outperforms DFCM which has an accuracy of 69% but a slowdown of 1.65x. Here $\{W, S, D\} = \{4, 2, 0\}$, which corresponds to a warmup of $w = 4096$ returns, specialization threshold of $s = 16$ correct predictions (25% accuracy), and a despecialization threshold of $d = 0$, meaning no despecialization will occur. This choice is 5% worse than the optimal

for $W \leftarrow -1$ **to** 7 **do**
  for $S \leftarrow 0$ **to** 8 **do**
    for $D \leftarrow 0$ **to** S **do**
      if $W = -1$ **then** $w \leftarrow 0$ **else** $w \leftarrow 2^{3W}$
      $s \leftarrow 8S$
      $d \leftarrow 8D$
      measure (w, s, d)

Figure 3.15: *Online hybrid parameter sweep configuration.*

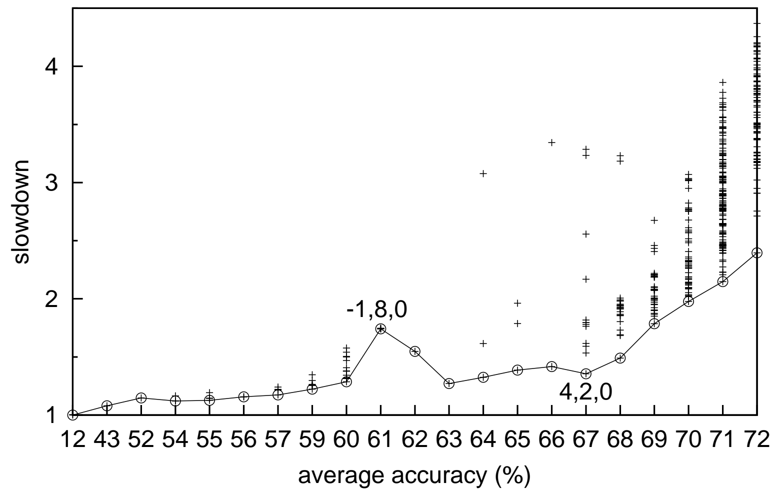


Figure 3.16: *Online hybrid parameter sweep.*

accuracy at 72% with slowdown of 2.40x. The cheapest configuration of $\{-1, 0, 0\}$ is equivalent to the null predictor and only achieves an accuracy of 12%.

The data point at accuracy 61% with slowdown 1.74x also stands out. The corresponding configuration, $\{-1, 8, 0\}$, means that $w = 0$, $s = 64$, and $d = 0$. This predictor has no warmup, nor does it despecialize, and it is quite slow. It was selected by the optimization for that data point for two reasons. First, its high specialization threshold did ultimately result in some good sub-predictor choices. Second, there were only four configurations to choose from at that accuracy level, because the distribution of experiments is not even along the x-axis and most experiments cluster in the upper accuracy range. Interestingly, in

all but the top three most accurate and slowest cases, $d = 0$. We conclude that although despecialization may offer slight accuracy benefits, they come with sharply increasing costs.

3.5.3 Performance Comparisons

We finally compared the behaviour of our offline and online adaptive hybrids with the naïve non-adapative hybrid. We show predictor accuracies, slowdowns, and memory consumption for all three in Figures 3.17 and 3.18 and Table 3.10 respectively. We used a maximum table size of 2^{25} entries in these experiments to prevent memory constraints from interfering with accuracy results.

In terms of accuracy, we expected the naïve hybrid to act as an oracle with respect to the offline hybrid, behaving like the online hybrid but configured with an infinite warmup period. The data in Figure 3.17 show that offline specialization is quite effective, for it is always within 3% accuracy of the naïve version. In some cases the accuracy is actually slightly better, because the constant availability of all predictors in the naïve version can lead to suboptimal choices. The close match between offline and naïve accuracies indicates two things. First, ideal sub-predictors do in fact exist for the vast majority of callsites. Second, for these benchmarks, significant program phases are either rare or non-critical with respect to adaptive RVP performance, because the offline hybrid uses a fixed set of sub-predictors over the entire program run. Accuracy is not significantly compromised in the online hybrid, dropping by at most 11% when compared to offline accuracy.

Predictor slowdowns are dramatically reduced by both offline and online hybrids, as shown in Figure 3.18. Online performance is on average equivalent to offline, where offline is worse when it chooses accurate but expensive table-based predictors, while online is worse when the cost of warmup is too high. This effect can also be seen in the memory consumption data in Table 3.10. Both offline and online hybrids greatly reduce memory requirements, with the best case for offline being the reduction of `mpegaudio` by over 24 times. Online memory usage tends to be somewhat larger than offline, with the exception of `db`, an extreme example where the online hybrid is orders of magnitude cheaper. The bottom half of Table 3.10 shows the further memory reductions that straightforward elimination of wasteful memory use in our system would provide.

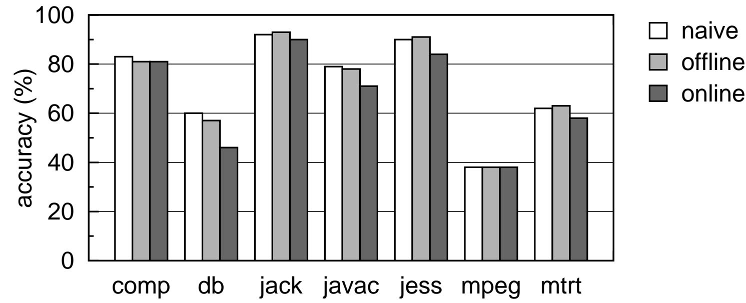


Figure 3.17: *Naïve vs. offline vs. online accuracies.*

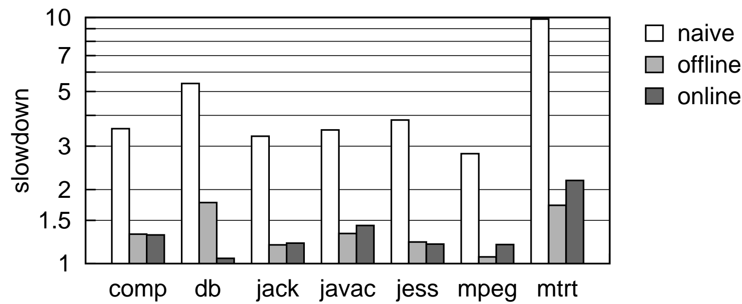


Figure 3.18: *Naïve vs. offline vs. online slowdowns.*

predictor	comp	db	jack	javac	jess	mpeg	mtrt
naïve	1.31G	2.80G	91.0M	412M	47.2M	4.98G	6.37G
offline	484M	771M	5.83M	190M	6.11M	206M	417M
online	486M	5.19M	9.16M	53.8M	7.43M	256M	1.07G
no logs	324M	3.62M	6.41M	36.4M	5.25M	171M	732M
32-bit keys	243M	2.90M	5.14M	27.8M	4.27M	128M	549M
type info	162M	2.18M	3.77M	19.2M	3.43M	86.1M	367M
perfect Z	162M	2.14M	3.70M	19.1M	3.40M	86.0M	367M

Table 3.10: *Naïve vs. offline vs. online memory consumption.* The four additional rows indicate the cumulative memory consumption benefits due to removing a backing log from hash tables, using 32-bit table keys instead of 64-bit keys, using VM knowledge about type widths, and using perfect hashing for booleans in the context-based predictors. Perfect boolean hashing means that an order-5 context-based predictor only requires 5 bytes, 1 byte to hold the 5-bit context and 4 bytes to hold the $2^5 = 32$ possible values.

3.6 Conclusions

The ideal choice of return value predictor varies widely, depending on dynamic benchmark and callsite properties. A flexible, software-based design for RVP thus has many advantages, permitting a wide variety of arbitrarily complex predictors and an adaptive mechanism for optimizing their application. The latter is especially important for software implementations, where a naïve design imposes memory and speed overheads that can easily outweigh any derived benefit. We found that using a variety of callsite-bound predictors that include complex, table-based predictors can result in very high accuracy. Our online adaptive hybrid is effective at maintaining this accuracy while reducing overhead costs to reasonable levels. It does so by identifying and specializing to ideal sub-predictors, which we found do generally exist at the callsite level. If the total runtime overhead of ubiquitous RVP in this study remains a concern, applications can easily tailor their usage to reduce it.

Our software-only focus played an important role in this work. The search for a simple hierarchical design led to the high level specialization optimization in our adaptive hybrid predictor, which suggests that clean design and object-orientation stand to benefit software analogues of hardware components in general. We found that after many years of research, history-based prediction studies covered the design space rather well, missing only the 2 delta last value predictor. This suggests that early attempts to formalize the design of runtime components may be beneficial. For example, our composite stride pattern makes it easy to create stride based derivatives of any predictor. Our software context allowed us to consider a large number of sub-predictors at low cost, and we found that they all had application at different points. Memoization is particularly effective when applied to RVP, and complements existing predictors nicely in a hybrid.

3.7 Future Work

There are many potential applications for this technology. Of course, return value prediction was originally conceived to support method level speculation, which as seen in Chapters 1 and 2 executes a function continuation speculatively and in parallel with the function call. RVP significantly improves method level speculation performance in both

hardware [CO98, OHL99, HBJ03] and software systems, the latter demonstrated in Chapter 2, by enabling longer thread lengths and thus greater parallelism. Close to the original motivation of speculative execution, return value prediction could also enhance *safe futures* [WJH05, ZKN07a], a source level continuation-based parallelization construct that supports speculative reads and writes, by allowing for speculation past the consumption of the return value. Aside from certain predictors that take function arguments, there is nothing preventing our design from also being used for more general load value prediction, which has application to both software thread level speculation [OM08] and transactional memory implementations [PB09, THG09].

More broadly, any instruction that produces a value can be considered a function, and so the technique is readily extended to non-return values. A key analysis in JIT compilers is value profiling, which enables method body specialization according to expected values [CFE97, SYK+01]. Thus software (return) value prediction could be used to generalize value profiling to support multiple concurrent profiles and hence multiple specializations of a method. A second use of RVP-based profiling is program understanding, where *post mortem* analysis of specific predictor behaviours can provide insight into the run-time behaviour of individual programs and functions. A third use of RVP-based profiling is in software self-healing, which seeks to repair damage from network attacks [LSC+08]. Apart from profiling and speculative execution, value prediction can be used to prevent stalls due to memory latencies, both in distributed and multi-core systems [LG09], and to support prefetching [ZC05]. Finally, outside of programming languages, our fast, accurate, and memory efficient software RVP design could apply to the field of machine learning, where making future predictions based on past behaviour is often important, for example in robotics, stock market prediction, competitive game-playing or multi-agent cooperative settings.

In terms of design, predictor accuracy could be improved by identifying hot but unpredictable callsites and designing new predictors to accommodate them. Generalized software value prediction using our framework may benefit from several additional predictors not suitable for return values. Attaching predictors to methods and invocation edges instead of callsites may alternatively improve accuracy or reduce overhead. Various static analyses and program transformations to support prediction are also possible, building on previous

work in this area [BDH02]. Finally, there are undesirable interactions between RVP and MLS that would benefit from mitigating techniques; Hu *et al.* previously observed that the hybrid predictor accuracy of their RVP system dropped by about 10% when MLS was enabled [HBJ03]. One problem is that MLS changes the execution order such that predictions and / or updates may happen out of order. Another is that in the case of in-order nesting, MLS introduces speculative predictions and updates that might be later aborted. Finally, there is the additional question as to whether predictions and / or updates at a callsite should always be made even if threads are only sometimes forked there.

In terms of implementation, a mixture of software and hardware support may be appropriate [BSMF08]. Our design could certainly accommodate hardware versions of specific sub-predictors when available. Furthermore, a general purpose hardware hash function could improve the performance of table-based predictors, and have broad applicability outside of value prediction. We are also interested in refactoring the RVP code from libspmt into a separate library for general purpose reuse; a disadvantage of this approach is that *inter*-library whole program optimization is much less commonly supported than *intra*-library whole program optimization. A JIT compiler integration of RVP which weaves intermediate representations of predictor code into generated code instead of inserting calls to library functions may be worthwhile in terms of eliminating library overhead; certainly the baseline cost of the null predictor shown in Figure 3.4 should be much smaller in a production system. Lock-based or lock-free optimizations for concurrent predictor access could also significantly reduce overhead. Finally, we are particularly interested in the impact of JIT compiler method inlining on overall predictor behaviour, since it will result in profound changes to benchmark properties.

Chapter 4
Nested Method Level Speculation & Structural Fork Heuristics

In Chapter 2 we described an MLS system that allowed for out-of-order nesting, wherein non-speculative parent threads can have multiple speculative children, but not in-order nesting, where speculative children can create speculative children of their own. During profiling data analysis, we found that this led to idle processors, and that support for in-order nesting was necessary for maximizing processor usage. Here a producer / consumer memory allocation problem presented itself: a child thread would allocate the memory for a new child in one thread but that memory would get freed by the parent in a different thread. To this end we designed a custom multiprocessor memory allocator based on recycling aggregate thread data structures. In this chapter we first present our memory allocator as a practical solution for any software MLS system supporting in-order nesting. Once in-order nesting was enabled, we found that far too many threads were created when using our dynamic fork heuristics, precluding meaningful analysis of even simple benchmarks. We concluded that the runtime performance of MLS strongly depends on the interaction between program structure and MLS system configuration, making it difficult to compare approaches or understand in a general way how programs behave under MLS, particularly with respect to thread nesting.

Accordingly, in this chapter we seek to establish a basic framework for understanding and describing nested MLS behaviour. We present a stack-based abstraction of MLS that

encompasses major design choices, including in-order and out-of-order nesting. This abstraction is drawn directly from our implementation inside SableSpMT and libspmt. We then use this abstraction to develop the structural operational semantics for a series of progressively more flexible MLS models. Assuming the most flexible such model, we provide transition-based visualizations that reveal the speculative execution behaviour for a number of simple imperative programs. These visualizations show how specific parallelization patterns can be induced by combining common programming idioms with precise decisions about where to speculate, forming a set of structural fork heuristics. We find that the runtime parallelization structures are complex and non-intuitive, and that both in-order and out-of-order nesting are important for exposing parallelism. We also show how the parallelization patterns used by the Olden suite of benchmarks can be expressed in our framework. Our primary conclusion here is that either programmer or compiler or profiler knowledge of how the structure of implicit parallelism develops at runtime is necessary to maximize performance. At the language level this could mean introducing explicit "try this in parallel" keywords, or it could mean simply writing programs to be more amenable to automatic techniques. At any rate, we believe a balance between explicit and implicit approaches is likely to be most useful.

4.1 Introduction

The profiling work in Chapter 2 revealed two performance issues that are addressed in this chapter. First, processors are often idle when only out-of-order nesting is allowed. Our solution to that problem was to implement in-order nesting in SableSpMT, along with support for multiprocessor memory management of child thread data structures. Although we found that in-order nesting effectively eliminates the problem of idle speculative processors in the system, we also found that arbitrary in-order nesting increases the total overhead to the point that most programs require so much time to complete with our "always fork" heuristic that systematic experimentation with dynamic fork heuristics is simply not viable. The second performance issue is that most speculative threads are short-lived. Given that we want to include in-order nesting in any approach to creating longer threads, a systematic experimental investigation based on varying dynamic fork heuristics is already ruled

out. Our solution here is to identify positive and negative interactions between code structure and speculative thread behaviour using an abstract model of our various nesting forms. Thus, a lack of experimental insight into the performance issues and runtime behaviour of SableSpMT is the primary motivation for the material in this chapter.

Our initial hope with MLS, as with most work on automatic parallelization, was that irregular programs would run faster, exploiting parallelism with no additional programmer intervention. However, due to variability in the balances between parent and child thread lengths, value predictability, and the likelihood of dependence violations, some fork points end up being much better than others, and the overhead of bad forking decisions can easily dominate execution time. Naturally, one's first thought is to change the set of fork points to accomodate. Although this does have an effect on parallelism, it does so not only by eliminating the overhead from unprofitable speculation, but also by changing the dynamic thread structure and hence enabling parallelism where it was previously precluded. The complication is that changing the dynamic thread structure in turn changes the suitability of fork points. For an online or offline adaptive system that creates threads based on the suitability of fork points, this creates a feedback loop. The end result is that sometimes parallelism is obtained, sometimes not, but ultimately it is difficult to explain *why* things play out the way they do.

There has been significant work on selecting fork points and the final effect on performance. There has been much less focus, at least in the case of MLS, on studying the relationship between program structure, choice of fork point, and the resultant parallel behaviour. For this we need an abstract way to describe the program structure and choice of fork point, and we need a way to "see" the behaviour, which in this case is a visualization of how parallel structures evolve over time. We also need a model of MLS that is flexible enough to allow for exhaustive exploration. The initial MLS implementation described in Chapter 2 allowed parent threads to allocate multiple child threads, thus providing out-of-order nesting. However, it did not allow for child threads to create speculative child threads of their own; in other words, in-order nesting was prohibited. Figures 1.3 and 2.7 clarify the differences between these two kinds of nesting.

In the first part of this chapter we describe a multithreaded memory allocator that makes in-order nesting practical and efficient. It is a simple design based on freelists, and the core

mechanism is recycling entire child thread data structures at once. In the next part of this chapter, we provide an abstract description of MLS via a unified model based on call stacks that encompasses all possible nesting scenarios. This abstraction is drawn from our practical implementation in SableSpMT, and in particular from the refactoring work done to create libspmt, our library for speculative multithreading. We then provide a series of sub-models referring to our larger unified one that are each described by their structural operational semantics, where the structures involved are the precise relationships between call stacks and threads. We then take an abstract view of program structure that isolates useful work from higher-level organizational concerns, enabling a focus on the effects of code layout. Finally, we provide and employ a method for visualizing runtime parallel behaviour that relies on showing the state evolution that arises from repeated application of our structural rules in combination with choice of fork point. This forms a series of speculation patterns, which in turn imply a set of structural fork heuristics.

Given such a framework, we can compare the parallelization models used by MLS-like systems directly, and begin to understand at a non-superficial level why the results differ between them and between the programs they parallelize. Our vision is that this understanding can be used to inform programmers and compiler writers trying to structure or restructure programs for efficient implicit parallelism, and to guide the design of runtime systems. A final benefit of our approach provides a basis for inventing novel extensions: it allows for rapid specification and visualization without the burden of implementation.

4.1.1 Contributions

We make the following specific contributions:

- We present a multithreaded allocator that provides a simple solution for two constrained memory management problems that arise under software MLS. First, it recycles constant-shape child thread aggregate data structures at once, dramatically reducing the number of calls to the system allocator. Second, it accounts for a producer / consumer problem under in-order nesting where memory allocated in one thread is freed in another.

- We propose a stack-based operational semantics as a unified model of MLS. This model is derived from the working implementation described in Chapter 2. Our model provides support for lazy stack buffering at the frame level, a unique optimization designed to improve the speed of local variable access; other systems have relied instead on general dependence buffer or transactional memory support. The semantics of MLS have not been considered by previous work beyond basic fork and join matching.

- We provide several MLS sub-models, each of which is described by its structural operational semantics, and relate them to our unified stack model. These sub-models are suitable for direct visualization of programs executing under MLS.

- We examine the behaviour of a number of common coding idioms in relation to our stack formalism. We show how these idioms map to specific parallel runtime structures depending on code layout and fork point choice. We derive several guidelines as to how parallelism can be exposed, investigate a complex example that demonstrates implicit workload scheduling using recursion, and study the Olden benchmarks for similar patterns.

In Section 4.2, we describe our memory allocator, a prerequisite for efficient child allocation and the in-order nesting described in future sections. In Section 4.3, we present our unified stack model, which we use to develop a series of MLS sub-models in Section 4.4. We explore coding idioms and behaviour in Section 4.5, and finally conclude and discuss future work. Related work specific to memory allocation, nested MLS, irregular parallelism, and fork heuristics is described in Chapter 5.

4.2 Child Thread Memory Allocation

In this section we describe a simple memory allocator with two distinct features that allows for efficient child thread allocation when using either out-of-order or in-order nesting. The first feature is the use of freelists to manage aggregate child thread data structures, and the second is the migration of freelist blocks between processors and a global runtime pool.

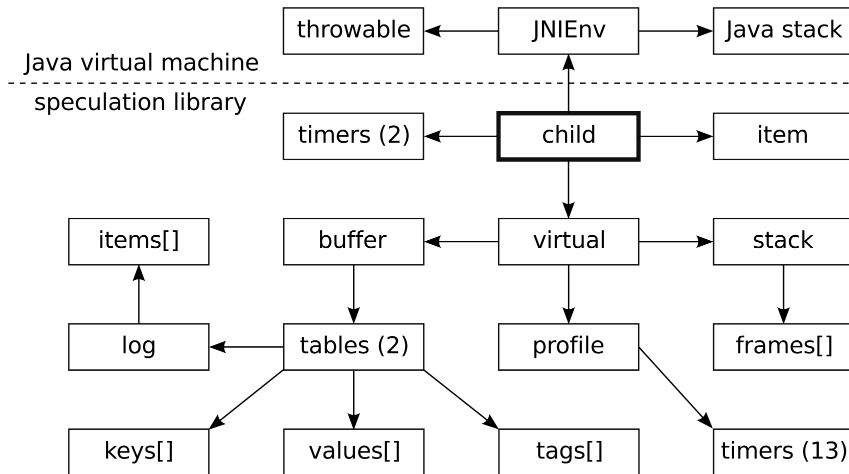


Figure 4.1: *Runtime child data structure.*

4.2.1 Ownership Based Allocation

The data structure for a single child thread is shown in Figure 4.1. This is the implementation in libspmt after refactoring the code from SableSpMT; it contains 37 separately allocated sub-objects. On the library side there is memory for a dependence buffer, profiling information, and nested MLS stack information, and on the VM side there is memory for a thread environment and the actual Java call stack. We experimented with allocating a new child instance on every thread fork and freeing the memory on every commit or abort, which implied 37 calls to `malloc` and `free` respectively. When combined with a high frequency of allocation, as permitted by our out-of-order nesting model and liberal fork heuristics, this quickly overwhelmed the capabilities of the Lea allocator on our system [Lea00].

We observed that each child instance was an *ownership dominator tree* [Mit06] rooted by the `child` sub-object. This means that all other sub-objects are reachable only through it, at least at allocation and deallocation time. We also observed that the only differences between children were the size of the dynamically resizable `stack` and `Java stack` nodes. Since the memory of dead child instances is guaranteed to be unreachable by the rest of the system, and since all instances have the same basic shape, we were able to use one child freelist for each parent thread and avoid excessive calls to `malloc` and `free`. The only

additional need is a new child "reset" operation that gets called on allocation and zeroes out important elements of the child state.

4.2.2 Multithreaded Freelist Allocation

After the initial implementation of out-of-order nesting in Chapter 2, we profiled the system and found that processors were mostly idle. For the SPEC JVM98 benchmark suite running on 4 processors, helper threads were idle on average for 73% of their running time. Our conclusion was that not enough parallelism was being exposed by out-of-order speculation alone and that support for in-order speculation was necessary for maximal parallelism.

However, in-order speculation introduces a memory allocation problem when children are allocated from child freelists. Consider the simple example where child thread C1 allocates its own child C2. After some time, C1's parent P joins C1, inheriting C2, and then later joins C2, resulting in P freeing C2's memory. The problem here is that P did not allocate C2's memory; for out-of-order speculation this problem does not arise because children allocated by P are always freed to P's freelist. This kind of producer / consumer pattern can lead to memory pooling up in one thread if there is an imbalance between calls to `malloc_child` and `free_child`. Experiments with a version of our system that still had this problem quickly exhausted all available memory.

Our solution is a custom multithreaded freelist allocator with thread local and global blocks of children, as depicted in Figure 4.2. On the left a child is freed to a thread local block of children. If that block becomes full it is exchanged for an empty one via global synchronization at the runtime level. The malloc process is exactly the inverse, exchanging an empty block for a full one if necessary and then producing a child for the current parent or helper / worker thread. Larger block sizes reduce the need for global synchronization, albeit at the expense of extra memory consumption. Figure 4.3 provides an implementation where the only actual calls to `malloc` are inside `create_set` and `create_child`, which in turn are inside `block_create`. This function allocates an entire block of children at once, and is only called when no full blocks are locally or globally available.

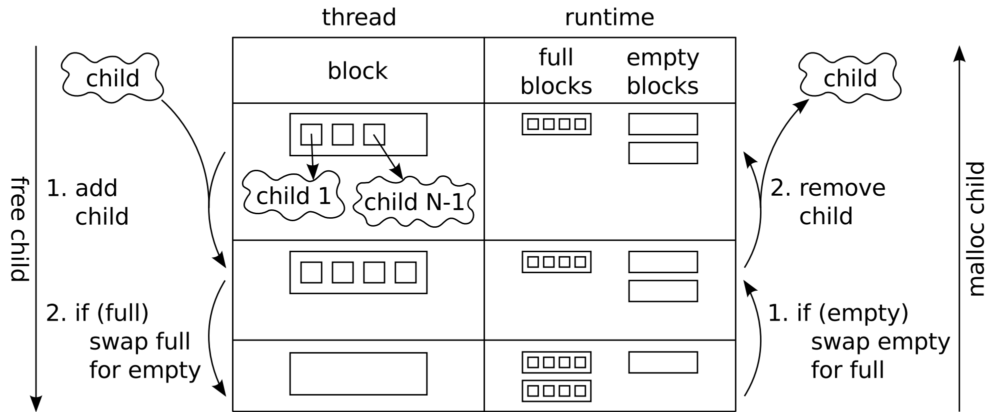


Figure 4.2: *Multiprocessor child malloc and free.* On the left a child is freed to a local block. The block becomes full and so it is exchanged for an empty one. On the right an allocation request is made. However, the local block is empty, and so must first be exchanged for a full one.

4.2.3 Discussion

This scheme has the following advantages: 1) functionality across a library–VM interface: our library calls back into a JVM to create an appropriate thread context; 2) support for child sub-structure type opacity; 3) minimal initialization costs; 4) implementation simplicity; 5) support for dynamically resizable sub-structures, here the dependence buffer and call stacks; 6) portability; 7) no external library dependences; 8) no synchronization operations in the common case, namely allocating or freeing a child task from or to a local thread block; 9) memory consumption proportional to the number of processors and the maximum number of live children.

The scheme also has its disadvantages: 1) potential lack of locality between child substructures; 2) lack of locality between processors: an individual child task may visit 3 different cores, the allocating, executing, and freeing ones; 3) no reclamation of excess child task memory; 4) lock-based synchronization in the uncommon case, namely exchanging empty and full blocks between a thread and the global pool of blocks; 5) lack of automation and general purpose applicability.

As far as alternative approaches are concerned, a typical solution might be to rewrite the child data structure to be contained in one contiguous region of memory and then manage it using an existing SMP malloc replacement such as Hoard [BMBW00]. However,

```
set_t *
create_block (void)
{
  set_t *block = create_set ();
  while (!is_full (block))
    add_child (block, create_child ());
  return block;
}

set_t *
swap_empty_for_full (runtime_t *runtime,
                     thread_t *thread, set_t *empty)
{
  set_t *full;
  acquire (runtime->blockset_lock, thread);
  add_block (runtime->empty_blocks, empty);
  full = (is_empty (runtime->full_blocks)) ?
    create_block () : remove_block (runtime->full_blocks);
  release (runtime->blockset_lock, thread);
  return full;
}

set_t *
swap_full_for_empty (runtime_t *runtime,
                     thread_t *thread, set_t *full)
{
  set_t *empty;
  acquire (runtime->blockset_lock, thread);
  add_block (runtime->full_blocks, full);
  empty = remove_block (runtime->empty_blocks);
  release (runtime->blockset_lock, thread);
  return empty;
}
```

Figure 4.3: *Source code for child malloc and free.*

```
child_t *
malloc_child (thread_t *thread)
{
  if (is_empty (thread->block))
    thread->block =
      swap_empty_for_full (thread->runtime, thread, thread->block);
  return remove_child (thread->block);
}

void
free_child (thread_t *thread, child_t *child)
{
  add_child (thread->block, child);
  if (is_full (thread->block))
    thread->block =
      swap_full_for_empty (thread->runtime, thread, thread->block);
}
```

Figure 4.3: *Source code for child malloc and free (continued).* `malloc_child` allocates a child from a thread-local block if one is available, otherwise exchanges an empty block for a full one by calling `swap_empty_for_full`. `free_child` frees a child to a thread-local block, and if a full block is created it then calls `swap_full_for_empty` to exchange it for an empty one.

there are three good reasons for working with the existing structure. First, some elements of the structure may be dynamically resizable, in this case the dependence buffers and call stacks. Second, the data structure to be recycled is actually split across a library–VM interface: when allocating a child, the library calls back into the VM to create an appropriate thread context for execution. Third, there are software engineering arguments: modularity, minimizing source changes, and the benefits of type opacity. Although it would be straightforward to merge *some* of the sub-structures using inheritance, composition is more straightforward in C.

4.3 Nested MLS Stack Abstraction

The preceding memory allocation algorithm is a necessary prerequisite for efficient child task allocation that also solves the in-order nesting producer / consumer problem. We now present a core stack abstraction that directly encodes the call stack and thread manipulations central to all MLS designs. This abstraction is flexible and supports in-order nesting, out-of-order nesting, in-order speculative commits, and any combination thereof. Specific models that implement these features using our abstraction are developed in Section 4.4.

The standard sequential call stack model found in most languages has two simple operations that manipulate stack frames: PUSH for entering methods, and POP for exiting methods. Frames store local variables and other context required for correct method execution, and for well-behaved languages the operations must be matched. For languages that support multithreading, START and STOP operations for creating and destroying non-speculative threads are also necessary. Our parallel call stack model for MLS is simply a parallel extension of this standard. It introduces three new operations, FORK, COMMIT, and ABORT. These new operations manipulate stack frames, but they also have the power to create and destroy speculative threads. FORK can now be called instead of PUSH, pushing a frame *and* creating a new child thread, and upon return COMMIT or ABORT will be called to match the FORK instead of POP.

We make several assumptions: 1) well-ordered PUSH and POP nesting is provided by the underlying language, even in the case of exceptional control flow; 2) stack operations complete atomically; 3) non-stack operations, while not explicitly modelled, may be freely interleaved with stack operations on running threads; 4) speculative accesses to global state variables, if they exist, are handled externally, for example via some transactional memory or dependence buffering system such as that described in Section 2.4.4; 5) register values are spillable to a frame on demand; and 6) stacks grow upwards. We also say that parent threads are *less speculative* than their children because they are either non-speculative or closer to a non-speculative thread than their children; similarly, children are *more speculative* than their parents.

The model has two unique features that separate it from naïve speculation where all reads and writes go through a dependence buffer or transactional memory subsystem. First,

child threads buffer stack frames from their parents, such that all local variable accesses go directly through a local frame. This is intended to reduce the load on the dependence tracking system. Second, stack frames are buffered as lazily as possible: on forking, only the frame of the current method is copied to the child. If the child ever needs lower down frames from some parent thread, it retrieves and copies them on demand. This lazy copying introduces significant complexity: the POP operation may need to buffer a frame, and the COMMIT operation needs to copy back only the range of live frames from the child thread stack. We include it as a practical measure intended to make our abstraction useful: our experience with the SableSpMT software implementation described in Chapter 2 indicates a steep performance penalty for copying entire thread stacks. If a child needs to buffer a frame but its parent is itself speculative, it might not have a copy of the frame in question. In this case the parent's ancestors are searched for the frame until the initial non-speculative thread is reached, which is guaranteed to have a copy. Just as parents are less speculative than their children, the unbuffered versions of stack frames in those parents are less speculative than the buffered versions in their children.

The main abstraction is described via its operational semantics in Figure 4.4. Here operations act like functions, requiring a list of arguments. In some cases they also return values, which are separated from arguments by $|$. There are seven publicly available operations, each marked with $[*]$. These in turn use a number of internal operations, for both clarity and logic reuse. A summary of the public operations and their observable behaviour follows:

$[*]$START$(|t)$: create a new non-speculative thread t with an empty stack.

$[*]$STOP(t): destroy non-speculative thread t, provided its stack is empty.

$[*]$PUSH(t, f): add a new frame with unique name f to the stack of thread t.

$[*]$FORK$(t, f|u)$: execute PUSH(t, f) and then create a new child thread u that starts executing the method continuation using a buffered version of the previous frame from thread t. Cannot be issued on an empty stack.

$[*]$POP(t): remove the top frame from the stack of thread t. For speculative threads there

must be a frame to pop to, either in the thread itself or some ancestor. The matching operation must be a PUSH.

$[*]$ABORT(t): execute POP(t) (internally JOIN$(t|u)$) and abort the child thread u attached to the frame underneath, recursively aborting all of its children. The matching operation must be a FORK.

$[*]$COMMIT(t): execute POP(t) (internally JOIN$(t|u)$) and commit the child thread u attached to the frame underneath, copying all of its live stack frames and any associated child pointers. Committed children with children of their own are kept on a list attached to t until no references to them exist, lest another speculative thread attempt to copy a stack frame from freed memory. The matching operation must be a FORK.

We now turn to a detailed description of the operations in Figure 4.4. We model threads as unique integers, and maintain several thread sets: T is the set of all threads, T_n is the set of non-speculative threads, T_s is the set of speculative threads, T_l is the set of live threads, T_d is the set of dead threads, and T_c is the set of committed threads that are still reachable as the ancestor of some $t \in T_s$. Some invariants apply to these sets in between public operations: $T_n \cup T_s = T_l$, $T_n \cap T_s = \emptyset$, $T_l \neq \emptyset \rightarrow T_n \neq \emptyset$, $T_l \cup T_d = T$, $T_l \cap T_d = \emptyset$, and $T_c \subseteq T_d$. Elements are never removed from T, such that each new thread gets a unique ID based on the current size of T, namely $|T|$. There is also a special set variable T_p, the current thread pool, which is only used internally and bound to either T_n or T_s. Stack frames are modeled by a set of unique frames F, such that each newly pushed or buffered frame is not already in F. This invariant is maintained by removing frames from F when threads are aborted. Given a frame $f \in F$, buffering creates a new frame f' by appending $'$ to the name. Given a frame f', f is the less-speculative version of the same frame in some ancestor thread. Note that for notational convenience, e' and f' in POP(t) may belong to a non-speculative thread, in which case no such ancestor exists. Variables d, e, f and their primed derivatives represent individual frames, whereas σ, ρ, π represent lists of frames. Similarly, variables c, p, t, u represent individual threads, whereas γ, δ represent lists of threads.

In addition to these sets, there are several functions that maintain mappings between them. $stack(t \in T)$ maps t to a thread stack, which is a list of frames in F, $child(f \in F)$

$$\textsc{create}(T_p, \sigma|t)\,\frac{T_p = T_n \oplus T_p = T_s}{t = |T|, T \cup\!= \{t\}, T_l \cup\!= \{t\}, T_p \cup\!= \{t\}, stack(t) = \sigma}$$

$$\textsc{destroy}(t)\,\frac{t \in T_p \quad T_p = T_n \oplus T_p = T_s}{T_l \setminus\!= \{t\}, T_p \setminus\!= \{t\}, T_d \cup\!= \{t\}}$$

$$[*]\textsc{start}(|t)\,\frac{}{\textsc{create}(T_n, \emptyset|t)}$$

$$[*]\textsc{stop}(t)\,\frac{t \in T_n \quad stack(t) = \emptyset}{\textsc{destroy}(t)}$$

$$[*]\textsc{push}(t, f)\,\frac{t \in T_l \quad f \notin F \quad \sigma = stack(t)}{stack(t) = \sigma : f, F \cup\!= \{f\}}$$

$$\textsc{buffer}(t, e|e')\,\frac{t \in T_l \cup T_c \quad e \in stack(t) \quad e \in F \quad child(e) \notin T_s}{e' = e, F \cup\!= \{e'\}}$$

$$[*]\textsc{fork}(t, f|u)\,\frac{\dfrac{\textsc{push}(t, f)}{\sigma : e : f = stack(t),\ \textsc{buffer}(t, e|e')} \quad \textsc{create}(T_s, e'|u)}{parent(u) = t, child(e) = u}$$

$$[*]\textsc{pop}(t)\,\frac{\dfrac{t \in T_l \quad \sigma : e' : f' = stack(t) \quad f' \in F \quad child(e') \notin T_l}{\dfrac{t \in T_n}{e' \in F \oplus \sigma : e' = \emptyset} \oplus \dfrac{t \in T_s}{e' \in F \oplus \nu p\ .\ \textsc{buffer}(p, e|e')}}}{stack(t) = \sigma : e'}$$

Figure 4.4: *Stack operations.* Externally available operations are marked with $[*]$. START and STOP create and destroy non-speculative threads, PUSH, POP, FORK, COMMIT, and ABORT operate on existing threads, and all other operations are internal.

$$\textsc{join}(t|u)\frac{t \in T_l \quad \sigma : e : f = \mathit{stack}(t) \quad e, f \in F \quad \mathit{child}(e) \in T_l}{\mathit{stack}(t) = \sigma : e, u = \mathit{child}(e)}$$

$$\textsc{merge_stacks}(t,u)\frac{\dfrac{d' : \rho = \mathit{stack}(u)}{\dfrac{d \in \mathit{stack}(t)}{\sigma : d : \pi : e = \mathit{stack}(t)} \oplus \dfrac{d \notin \mathit{stack}(t)}{\sigma = \emptyset}}}{\mathit{stack}(t) = \sigma : d' : \rho}$$

$$\textsc{merge_commits}(t,u)\frac{\gamma = \mathit{commits}(t) \quad \delta = \mathit{commits}(u)}{\mathit{commits}(t) = \gamma : u : \delta, T_c \cup= \{u\}}$$

$$\textsc{purge_commits}(t)\frac{\gamma : \delta = \mathit{commits}(t) \,.\, \delta = \nu\delta \,.\, \forall c \in \delta \; \forall f \in \mathit{stack}(c), \mathit{child}(f) \notin T_l}{\mathit{commits}(t) = \gamma, T_c \setminus= \{\delta\}}$$

$$\textsc{cleanup}(t,u)\frac{\text{DESTROY}(u)}{\text{PURGE_COMMITS}(t)}$$

$$[*]\textsc{commit}(t)\frac{\text{JOIN}(t|u) \quad \text{MERGE_STACKS}(t,u) \quad \text{MERGE_COMMITS}(t,u)}{\text{CLEANUP}(t,u)}$$

$$\textsc{abort_all}(t)\frac{\dfrac{\forall f \in \mathit{stack}(t) \,.\, u = \mathit{child}(f) \in T_l}{\text{ABORT_ALL}(u)}}{F \setminus= \{\mathit{stack}(t)\}, \text{CLEANUP}(t,u)}$$

$$[*]\textsc{abort}(t)\frac{\text{JOIN}(t|u) \quad \text{ABORT_ALL}(u)}{\text{CLEANUP}(t,u)}$$

Figure 4.4: *Stack operations (continued).*

maps f to a speculative child thread $u \in T_s \cup T_c$, $parent(u \in T_s \cup T_c)$ maps u to the $t \in T$ that forked it, and $commits(t \in T_l)$ maps t to a list of threads in T_c. Initially all mappings and sets are empty.

Our rules make use of a few specific operators and conventions. The use of exclusive or ($\oplus$) indicates a choice between one rule and another or one set of premises and another. We use $S \cup= \{s\}$ and $S \setminus= \{s\}$ to indicate set additions and removals respectively. We use a period (.) to indicate "such that". Finally, we define a maximization operator ν that searches either for the greatest natural number or the longest list with a given property. Formally, $\nu y R(y)$ means the greatest or longest y such that predicate $R(y)$ is true, if $\exists y$ such that $R(y)$; otherwise 0 (for natural numbers) or $\emptyset$ (for lists). This definition is inspired by the μ or minimization operator from primitive recursive functions in computability theory [Kle52] and the ν or greatest fixed point operator from the μ-calculus in model checking [CGP99].

Lastly, we give a brief description of each rule. CREATE takes a thread pool T_p and a stack σ, checking that T_p is bound to either T_n or T_s. It then initializes a new thread t with ID $|T|$. Next, it adds t to T, T_l, and T_p, and initializes the stack of t to σ. Finally, it returns t. DESTROY conversely takes a thread $t \in T_p$ and binds T_p to either T_n or T_s based on whether $t \in T_n$ or $t \in T_s$. It then removes t from T_l and T_p, and adds it to T_d. START calls CREATE to make a new non-speculative thread $t \in T_n$ with an empty stack, and then returns it. STOP takes a thread t, checks that t is non-speculative and that its stack is empty, and then calls DESTROY to remove it. Note that here DESTROY will bind T_p to T_n since $t \in T_n$.

PUSH takes a fresh f and appends it to $stack(t)$, where t is live, also adding f to F. BUFFER takes either a live or committed thread, the name of a frame e in its stack, and provided there is no child attached to e creates e' for use by its caller, which is either FORK or POP. FORK first calls PUSH, buffers e' from e, creates u, and sets t as u's parent and u as e's child.

POP takes the stack of t, and checks that the top frame f' is valid and there is no child attached to the frame e' underneath. There is now a set of nested choices leading to one of four possible outcomes. If t is non-speculative, then either e' is a valid frame or the stack below f' is empty. If t is speculative, then either e' exists and can be found in $stack(t)$,

or e' needs to be retrieved and buffered lazily from the most speculative parent thread p that contains e. After one of these four cases is chosen, f' is popped by simply adjusting $stack(t)$ to exclude it. Note that this rule will prevent a speculative thread from popping the last frame on the stack, because in this case $e \notin stack(p)$ and the call to BUFFER will not complete. Similarly, it prevents a speculative thread from buffering and returning to a parent frame that has a child attached to it. JOIN has similar conditions to POP, except that here e must both exist and have a speculative child.

The expression νp . BUFFER$(p, e|e')$ in POP says to find the greatest or most speculative thread ancestor p that has a copy of the required stack frame e, and then use it to create e'. Threads are numbered in increasing order by CREATE, such that a parent thread always has a lower ID than its child. Thus by searching for the maximal p we find the parent closest to the child that has e, stopping when $p = 0$. The first thread in the system is non-speculative and created with ID 0. We have a guarantee that $e \in stack(0)$ if it is not found in any other thread, because a continuation will never attempt to return to a stack frame that was not previously entered by either itself or some ancestor.

MERGE_STACKS is called by COMMIT. It copies the live range of stack frames $d' : \rho$ from the child u to the parent t. There is now a choice between two cases. If d, the less-speculative version of the child's bottom frame d', exists in $stack(t)$, then the range of stack frames $d : \pi : e$ in t is replaced with the child stack $d' : \rho$. Otherwise $d \notin stack(t)$ and the entire parent stack is replaced with $d' : \rho$. Note that d will always be found if $t \in T_n$, since non-speculative threads must have complete stacks. d will only not be found if $t \in T_s$ and u has returned to some frame beyond the bottom of t.

MERGE_COMMITS, as called by COMMIT, takes the commit list γ from the parent, appends the child u and the child commit list δ, and adds u to T_c. PURGE_COMMITS is called every time CLEANUP is called. It removes threads without child dependences from the most speculative end of a commit list until either all committed threads have been purged or it encounters a dependency. The expression $\gamma : \delta = commits(t)$. $\delta = \nu\delta$. $\forall c \in \delta\ \forall f \in stack(c), child(f) \notin T_l$ in PURGE_COMMITS says to divide $commits(t)$ into two parts, γ and δ, where δ is the longest sub-list at the end of $commits(t)$ such that for every thread in δ, there are no live children attached to any of its stack frames.

CLEANUP simply destroys u and then purges t. It is called after the COMMIT and

ABORT operations, and internally from ABORT_ALL. Whereas JOIN contains the common logic that precedes commits and aborts, CLEANUP contains the common logic that follows them. COMMIT is a composite operation that joins t, merges stacks and commit lists using u from JOIN, and then cleans up. ABORT has a similar structure, calling ABORT_ALL internally, which performs a depth-first search looking for live children, and destroying them post-order. In any real-world implementation that uses this stack abstraction, child threads must be stopped before they can be committed or aborted.

4.4 Individual MLS Models

Using the stack abstraction from Figure 4.4 we now develop a series of concentric and progressively more flexible MLS models, each described by their structural operational semantics. The models are shown individually in Figures 4.5–4.11. Each consists of a group of rules, and each of these rules maps to a specific sub-case of a public ($[*]$) operation in Figure 4.4. The primary motivation for the rules is the stepwise depiction of stack and thread graph evolution in 2D. The primary motivation for their grouping into models is to specify the different types of MLS design available. In essence, these models provide an exhaustive visual reference to the MLS design considerations implicit in our unified abstraction by exposing the core state evolution patterns that define execution. A valid speculation for a given program at the call stack level is defined by a sequence of rule applications, each of which acts atomically. This sequence can be used straightforwardly to construct the thread and stack interleavings. In Section 4.5, we use these models to explore, visualize, and understand the behaviour of various code idioms under speculation.

Before describing the models, we will first explain the structure of the rules. With the exception of START and STOP, each rule is named using a combination of abbreviations. First there is a qualifier for the operation, which may non-speculative (N), speculative (S), in-order (I), or out-of-order (O). In some cases these may be combined, such that SI means "speculative in-order" and indicates an in-order operation performed by a speculative thread, IO means "in-order out-of-order" and indicates a nesting structure where an out-of-order fork follows an in-order fork, and OI is the inverse of IO. Next there is a symbol for the operation itself, which may be PUSH ($\downarrow$), POP ($\uparrow$), FORK ($\prec$), COMMIT ($\succ$),

or ABORT ($\not\succ$). Some of the POP ($\uparrow$) rules have a stack bottom ($\bot$) suffix, which means that the thread is performing a pop operation on the bottom frame of its stack. The MERGE suffix for I$\succ$MERGE indicates that two lists of committed threads created by in-order forks are being merged via the private MERGE_COMMITS operation in Figure 4.4. Finally, square brackets ([ ]) surrounding a rule name mean that its behaviour is actually provided by another rule, as indicated in the figure captions; their purpose is to illustrate specific details of stack evolution.

Each rule also consists of two parts. Above the inference line is the corresponding $[*]$ command from Figure 4.4, followed by model-specific restrictions on behaviour and local variable mappings. For a mapping $x = y$, x is a value found in the transitive expansion of the $[*]$ command from Figure 4.4 and y is the local value. Below the line is a visual depiction of the transition from one stack state to the next. For each stack state, threads appear horizontally along the bottom and are named in increasing order by τ, α, β, γ, δ, such that $\tau \in T_n$ and $\{\alpha,\ldots,\delta\} \subseteq T_s$, with a single exception in rule I$\uparrow\bot$ from Figure 4.9 where τ may be in T_s. Shown above each thread t is the value of $stack(t)$, which grows upwards. If $commits(t)$ is non-empty, it grows from left to right starting at t, with horizontal lines joining its elements. Finally, for each $f \in stack(t)$, a horizontal line joins it to the initial stack frame of its child thread if $child(f) \in T_l$. Thus the figures consist of threads, stacks, links between stacks at fork points, and links between threads and their committed children. The lengths of the horizontal lines between stacks are not fixed, and adjust to accomodate other threads. For example, in Figure 4.8, τ and α are linked together via π and π' both before and after the O$\prec$ transition, but afterwards the line grows to accomodate the new thread β, which is nested more closely to τ. This growth could be avoided by changing the horizontal order of α and β. However, β must appear before α to prevent stack frames from crossing horizontal lines: consider the state representation if β appeared after α, and α subsequently pushed some new frame g on its stack.

As in Figure 4.4, variables d, e, f and their primed derivatives are given to individual frames, whereas $\sigma, \varphi, \pi, \omega, \upsilon$ and their primed derivatives represent lists of frames. Note that ρ also appears, but only in the left side of the mappings from Figure 4.4 to these rules and never below the inference line for reasons of clarity and consistency. The rationale for the choice of different Greek letters varies. For threads, τ is simply 't' for 'thread', and α

through δ are the first four letters of the Greek alphabet that map to the first four speculative threads. The letters representing lists of stack frames were chosen primarily because they are suitably compact for 2D stack depictions, but there is also some intention here. σ is 's' for 'stack', and appears in order before φ and π as stacks grow upwards. Any of these three may be used at the stack bottom provided this ordering is maintained. ω is the last letter in the alphabet and so represents the top end of the stack. If ω is already in use for a different thread then υ can be used instead.

We now describe the actual models. Figure 4.5 contains a simple structured non-speculative stack model common to many languages, including Java. Non-speculative threads can START and STOP, delimiting the computation. In N$\downarrow$, a new frame can be pushed, where $\sigma \subseteq F$ and so may be $\emptyset$. N$\uparrow$ and N$\uparrow\perp$ match $e \in stack(\tau)$ and $e \notin stack(\tau)$ respectively to the two cases $e' \in F$ and $\sigma : e' = \emptyset$ of POP(t) in Figure 4.4. Note that N$\uparrow\perp$ is the penultimate operation on a thread, followed by STOP.

Figure 4.6 contains the simplest MLS stack model, one that extends Figure 4.5 to allow non-speculative threads to fork and join a single child at a time. In this model, speculative threads cannot perform any operations, including simple method entry and exit. For N$\prec$, there is a restriction on children being attached to prior stack frames, which prevents out-of-order speculation. N$\succ$ is the simplest COMMIT(t) possible, with the child stack containing only one frame, and N$\not\succ$ is similarly simple with no recursion required in ABORT(τ). Finally, the restriction $\tau \in T_n$ in N$\downarrow$ and N$\prec$ is sufficient to prevent speculative child threads from doing anything other than local computation in the buffered frame e': N$\succ$ and N$\not\succ$ must match with N$\prec$, N$\uparrow$ must match N$\downarrow$, and N$\uparrow\perp$ is precluded for speculative threads because BUFFER$(\tau, e|e')$ will not complete. This model is simplest to implement, but is only useful if the speculative work remains in the initial continuation frame.

The model in Figure 4.7 extends Figure 4.6 to allow speculative children to enter and exit methods. A speculative push S$\downarrow$ simply creates a new frame for α, specifying that π' is linked to π via some frame e' at the bottom of π' to the corresponding $e \in \pi$. S$\uparrow$ takes the left-hand case in POP(t) where $e' \in F$, whereas S$\uparrow\perp$ takes the right-hand case and so buffers e' from its parent. Finally, this model updates N$\succ$ and N$\not\succ$ to handle situations where the child may have left e' via S$\uparrow\perp$ or S$\downarrow$, now representing the child thread stack by φ' instead of e'. This model permits a variety of computations that contain method calls

in the continuation and use only a single speculative thread; example patterns involving straight-line code (Figure 4.12d) and if-then branching (Figures 4.13d and 4.13e) are given in the next section.

The next model in Figure 4.8 simply adds one operation to allow out-of-order nesting in non-speculative threads, O$\prec$. This rule specifies that if there is some lower stack frame d in π with a child attached, a new thread can be forked from e, complementing N$\prec$ in Figure 4.6 which prohibits this. All other existing operations continue to work as expected in this model. As seen in Chapter 2, this model is relatively straightforward to express in software, but does not expose maximal parallelism, as processors executing speculative threads are for the most part idle. Nevertheless, scalable out-of-order speculation involving head recursion (Figure 4.16d) and a mixture of head and tail recursion (Figure 4.17) is now possible.

After out-of-order nesting comes in-order nesting in Figure 4.9. I$\prec$ allows speculative thread α to create β independently of its parent. N$\not\succ$ will recursively abort these threads without modification, but I$\succ$ is required to allow a parent thread to commit child thread α with a grandchild β, maintaining the link to β and merging α onto the commit list of the parent. After β gets committed via N$\succ$, α will be freed, assuming there are no more children. I$\uparrow\perp$ is yet more complex, specifying that in order to buffer frame e', parent threads will be searched backwards starting from the grandparent until e is found. Here $\rightsquigarrow$ indicates that there is a path of buffered frames from π' backwards to π, and ... similarly indicates the possibility of intermediate threads between τ and α. This rule is an extended version of S$\uparrow\perp$, which only handles buffering from the immediate parent. S$\uparrow$ works nicely as is with in-order speculation, and S$\uparrow\perp$ works not only in the simple case above but also when the buffered frame is in some committed thread $c \in T_c$. This model works well for any speculation pattern that depends on in-order nesting, including ones for iteration (Figure 4.14c), tail recursion (Figure 4.15d), and head recursion (Figure 4.16e).

In Figure 4.10, speculative commits are now permitted. There are two simple rules, S$\succ$ and SI$\succ$, which complement N$\succ$ and I$\succ$ respectively. In the former β is purged from $commits(\alpha)$, whereas in the latter it is kept because of dependency γ. [I$\succ$MERGE] is implied by I$\succ$, and so adds nothing, but is shown to illustrate the full process of merging committed thread lists, where α and γ were already committed and β gets added between

$$\textsc{start}\ \frac{\textsc{start}(|\tau)}{\begin{array}{ll} & \Rightarrow \\ & \quad \tau \end{array}} \qquad \textsc{stop}\ \frac{\textsc{stop}(\tau)}{\begin{array}{ll} & \Rightarrow \\ \tau & \end{array}} \qquad \textsc{n}{\downarrow}\ \frac{\textsc{push}(\tau, f) \quad \tau \in T_n}{\begin{array}{lcl} & & f \\ \sigma & \Rightarrow & \sigma \\ \tau & & \tau \end{array}}$$

$$\textsc{n}{\uparrow}\ \frac{\begin{array}{c}\textsc{pop}(\tau) \quad e \in stack(\tau) \\ e' = e \qquad f' = f\end{array}}{\begin{array}{lcl} f & & \\ e & & e \\ \sigma & \Rightarrow & \sigma \\ \tau & & \tau \end{array}} \qquad \textsc{n}{\uparrow}{\perp}\ \frac{\begin{array}{c}\textsc{pop}(\tau) \quad e \notin stack(\tau) \\ e' = e \qquad f' = f\end{array}}{\begin{array}{lcl} f & \Rightarrow & \\ \tau & & \tau \end{array}}$$

Figure 4.5: *Adults-only model.* No speculation.

$$\textsc{n}{\prec}\ \frac{\begin{array}{c}\textsc{fork}(\tau, f|\alpha) \quad \tau \in T_n \\ \forall d \in \sigma, child(d) \notin T_l\end{array}}{\begin{array}{lcl} & & f \\ e & & e\text{—}e' \\ \sigma & \Rightarrow & \sigma \\ \tau & & \tau \quad \alpha \end{array}} \qquad \textsc{n}{\succ}\ \frac{\begin{array}{c}\textsc{commit}(\tau) \\ \rho = \emptyset \\ d = e \quad d' = e'\end{array}}{\begin{array}{lcl} f & & \\ e\text{—}e' & & e' \\ \sigma & \Rightarrow & \sigma \\ \tau \quad \alpha & & \tau \end{array}} \qquad \textsc{n}{\not\succ}\ \frac{\textsc{abort}(\tau)}{\begin{array}{lcl} f & & \\ e\text{—}e' & & e \\ \sigma & \Rightarrow & \sigma \\ \tau \quad \alpha & & \tau \end{array}}$$

Figure 4.6: *Totalitarian model.* One speculative child allowed, but only non-speculative threads can perform stack operations.

$$\textsc{s}{\downarrow}\ \frac{\begin{array}{c}\textsc{push}(\alpha, f) \\ \sigma = \pi' \qquad \omega \neq \emptyset \\ e' = car(\pi') \ .\ e \in \pi\end{array}}{\begin{array}{lcl} \omega & & \omega \quad f \\ \pi\text{—}\pi' & \Rightarrow & \pi\text{—}\pi' \\ \tau \quad \alpha & & \tau \quad \alpha \end{array}} \qquad \textsc{s}{\uparrow}\ \frac{\begin{array}{c}\textsc{pop}(\alpha) \qquad f' = f \\ \sigma : e' = \pi' \quad \omega \neq \emptyset \\ d' = car(\pi') \ .\ d \in \pi\end{array}}{\begin{array}{lcl} \omega \quad f & & \omega \\ \pi\text{—}\pi' & \Rightarrow & \pi\text{—}\pi' \\ \tau \quad \alpha & & \tau \quad \alpha \end{array}}$$

$$\textsc{s}{\uparrow}{\perp}\ \frac{\begin{array}{c}\textsc{pop}(\alpha) \qquad f' = \pi' \\ \sigma : e' = \emptyset \quad \omega \neq \emptyset \\ f = car(\pi)\end{array}}{\begin{array}{lcl} \omega & & \omega \\ \pi\text{—}\pi' & & \pi\text{—} \\ e & & e \quad e' \\ \varphi & \Rightarrow & \varphi \\ \tau \quad \alpha & & \tau \quad \alpha \end{array}} \qquad \textsc{n}{\succ}\ \frac{\begin{array}{c}\textsc{commit}(\tau) \\ d' : \rho = \varphi' \\ d : \pi : e = \varphi\end{array}}{\begin{array}{lcl} f & & \\ \varphi\text{—}\varphi' & & \varphi' \\ \sigma & \Rightarrow & \sigma \\ \tau \quad \alpha & & \tau \end{array}} \qquad \textsc{n}{\not\succ}\ \frac{\begin{array}{c}\textsc{abort}(\tau) \\ \sigma : e = \sigma : \varphi\end{array}}{\begin{array}{lcl} f & & \\ \varphi\text{—}\varphi' & & \varphi \\ \sigma & \Rightarrow & \sigma \\ \tau \quad \alpha & & \tau \end{array}}$$

Figure 4.7: *Kid-friendly model.* Allows PUSH and POP actions on speculative threads, overriding $\textsc{n}{\succ}$ and $\textsc{n}{\not\succ}$ to accomodate. The car function returns the first element of a list.

$$\textsc{o}\!\prec\frac{\text{FORK}(\tau, f|\beta) \quad \tau \in T_n \qquad d' = car(\pi')\ .\ d = car(\pi)}{\begin{array}{llll} & f & & \\ e & e\!-\!e' & & \\ \pi\!-\!\pi' & \pi\!-\!\!-\!\!-\!\pi' & & \\ \sigma \quad \Rightarrow & \sigma & & \\ \tau\ \ \alpha & \tau\ \ \beta\ \ \alpha & & \end{array}}$$

Figure 4.8: *Catholic model.* Provides out-of-order nesting via O≺ to allow an arbitrary number of speculative children for non-speculative threads.

$$\textsc{i}\!\prec\frac{\text{FORK}(\alpha, f|\beta) \quad \sigma = \pi' \quad \omega \neq \emptyset \quad d' = car(\pi')\ .\ d \in \pi}{\begin{array}{ll} & f \\ \omega\ \ e & \omega\ \ e\!-\!e' \\ \pi\!-\!\pi' \Rightarrow & \pi\!-\!\pi' \\ \tau\ \ \alpha & \tau\ \ \alpha\ \ \beta \end{array}}$$

$$\textsc{i}\!\succ\frac{\text{COMMIT}(\tau) \quad \tau \in T_n \quad \omega \neq \emptyset \quad d' : \rho = \varphi' : \omega \quad d : \pi : e = \varphi}{\begin{array}{ll} f\ \ \omega & \omega \\ \varphi\!-\!\varphi'\!-\!\varphi'' & \varphi'\!-\!\!-\!\!-\!\varphi'' \\ \sigma \quad \Rightarrow & \sigma \\ \tau\ \ \alpha\ \ \beta & \tau\!-\!\alpha\ \ \beta \end{array}}$$

$$\textsc{i}\!\uparrow\!\perp\frac{\text{POP}(\beta) \quad f' = \pi'' \quad \sigma : e' = \emptyset \quad \omega, \upsilon \neq \emptyset \quad f = car(\pi') \quad car(\pi') \rightsquigarrow car(\pi) \quad \nu p_{p \geq 0}\ .\ \text{BUFFER}(p, e|e') = \tau}{\begin{array}{ll} \omega\ \ \upsilon & \omega\ \ \upsilon \\ \pi\cdots\pi'\!-\!\pi'' & \pi\cdots\pi'\!- \\ e & e \qquad e' \\ \varphi \quad \Rightarrow & \varphi \\ \tau\ \ \alpha\ \ \beta & \tau\ \ \alpha\ \ \beta \end{array}}$$

Figure 4.9: *One big happy model.* Provides in-order nesting via I≺ to allow speculative children of speculative threads. Note that in I↑⊥, τ may be speculative.

$$\textsc{s}\!\succ\frac{\text{COMMIT}(\alpha) \quad \omega \neq \emptyset \quad d' : \rho = \varphi'' \quad d : \pi : e = \varphi'}{\begin{array}{ll} \omega\ \ f & \omega \\ \varphi\!-\!\varphi'\!-\!\varphi'' & \varphi\!-\!\varphi'' \\ \sigma \quad \Rightarrow & \sigma \\ \tau\ \ \alpha\ \ \beta & \tau\ \ \alpha \end{array}}$$

$$\textsc{si}\!\succ\frac{\text{COMMIT}(\alpha) \quad \omega, \upsilon \neq \emptyset \quad d' : \rho = \varphi'' : \upsilon \quad d : \pi : e = \varphi'}{\begin{array}{ll} \omega\ \ f\ \ \upsilon & \omega\ \ \upsilon \\ \varphi\!-\!\varphi'\!-\!\varphi''\!-\!\varphi''' & \varphi\!-\!\varphi''\!-\!\!-\!\!-\!\varphi''' \\ \sigma \quad \Rightarrow & \sigma \\ \tau\ \ \alpha\ \ \beta\ \ \gamma & \tau\ \ \alpha\!-\!\beta\ \ \gamma \end{array}}$$

$$[\textsc{i}\!\succ\!\text{MERGE}]\frac{\text{COMMIT}(\tau) \quad \omega \neq \emptyset \quad d' : \rho = \varphi''' : \omega \quad d : \pi : e = \varphi'}{\begin{array}{ll} f\ \ \omega & \omega \\ \varphi'\!-\!\!-\!\varphi'''\!-\!\!-\!\varphi'''' & \varphi'''\!-\!\!-\!\!-\!\!-\!\varphi'''' \\ \sigma \quad \Rightarrow & \sigma \\ \tau\!-\!\alpha\ \ \beta\!-\!\gamma\ \ \delta & \tau\!-\!\alpha\!-\!\beta\!-\!\gamma\ \ \delta \end{array}}$$

Figure 4.10: *Nuclear model.* Allows speculative threads to commit their own children. [I≻MERGE]'s behaviour is provided by I≻.

$$\text{IO}\prec \frac{\text{FORK}(\alpha, f|\gamma) \quad \omega \neq \emptyset \quad d' = car(\pi')\ .\ d = car(\pi)}{f}$$

$$\begin{array}{l} \omega \;\; e \\ \pi - \pi' - \pi'' \\ \sigma \\ \tau \;\; \alpha \;\; \beta \end{array} \Rightarrow \begin{array}{l} \omega \;\; e - e' \\ \pi - \pi' —— \pi'' \\ \sigma \\ \tau \;\; \alpha \;\; \gamma \;\; \beta \end{array}$$

$$[\text{IO}\not\prec] \frac{\text{ABORT}(\tau) \quad \omega \neq \emptyset \quad \sigma : e = \sigma : \varphi}{f}$$

$$\begin{array}{l} \quad\; \omega \\ f \;\; \pi - \pi' \\ \varphi - \varphi' —— \varphi'' \\ \sigma \\ \tau \;\; \alpha \;\; \gamma \;\; \beta \end{array} \Rightarrow \begin{array}{l} \varphi \\ \sigma \\ \tau \end{array}$$

$$[\text{OI}\prec] \frac{\text{FORK}(\beta, f|\gamma) \quad \sigma = \pi' \quad \omega \neq \emptyset \quad d' = car(\pi')\ .\ d \in \pi}{f}$$

$$\begin{array}{l} \omega \;\; e \\ \pi - \pi' \\ \varphi —— \varphi' \\ \tau \;\; \beta \;\; \alpha \end{array} \Rightarrow \begin{array}{l} \omega \;\; e - e' \\ \pi - \pi' \\ \varphi ———— \varphi' \\ \tau \;\; \beta \;\; \gamma \;\; \alpha \end{array}$$

Figure 4.11: *Libertarian model.* Allows both in-order and out-of-order nesting. $[\text{IO}\not\prec]$ and $[\text{OI}\prec]$ are provided by $\text{N}\not\prec$ and $\text{I}\prec$ respectively.

them. This model is good because it allows parallel commit operations, reducing the burden on the non-speculative parent thread. The disadvantage is that there is an increased risk of failure, because an otherwise valid speculation can become invalid by merging with a dependent invalid speculation.

Finally, in Figure 4.11, the last restrictions are removed so that all of the features in the main abstraction in Figure 4.4 are available. In this case, it suffices to provide $\text{IO}\prec$, which allows speculative threads to create child threads out-of-order. This was formerly prohibited by $\text{O}\prec$, which only applied to non-speculative threads. The other two rules are again shown only for purposes of illustration: $[\text{IO}\not\prec]$ shows a recursive abort on a thread with both in- and out-of-order nesting, and $[\text{OI}\prec]$ shows in-order nesting after out-of-order nesting has taken place, as already allowed by $\text{O}\prec$ followed by $\text{I}\prec$. This model is useful for more complex computations that depend on both in-order and out-of-order nesting, for example binary tree traversals (Figure 4.18) and divide and conquer algorithms (Figure 4.19).

The above models illustrate the core behaviour patterns of common speculation strategies. In the next section, we explore a series of stack evolutions that assume support for the final combined stack model in Figure 4.11, although in some cases one of the less flexible models will suffice.

4.5 Speculation Patterns

Simple changes in the structure of input programs and choice of fork points can dramatically affect the dynamic structure of the speculative call stack. In this section we explore

several common code idioms and their behaviour under MLS using the full stack abstraction. This exploration is done with a view towards discovering idiomatic code structures and speculation decisions that yield interesting parallel execution behaviours. These idioms, forking decisions, and runtime behaviour combine to form speculation patterns, from which we can identify structural fork heuristics. We first examine the simplest constructs in imperative programs, namely straight-line code, if-then conditionals, iteration, head recursion, and tail recursion. We then examine more complicated examples, including a mixture of head and tail recursion, binary tree traversals, and divide and conquer algorithms. We present a series of linear state evolutions to visualize the results, each of which is automatically generated from a list of operations given to an Awk script implementation of our model. We also examine the source code of the Olden benchmark suite for instances of these patterns.

In the examples that follow, we assume that useful computation can be represented by calls to a `work` function whose running time is both constant and far in excess of the running time of all non-work computation. Thus we can reason that if a thread is executing a `work` function, it will not return from that function until all other non-work computations in other threads possible before its return have completed. This reasoning guides the stack evolutions in cases where more than one operation is possible. These simplistic execution timing assumptions yield surprisingly complex behaviour, which indicates that our work here is a good basis for attempting to understand the behaviour of more complex programs with variable length `work` functions.

4.5.1 Straight-Line

The simplest code idiom in imperative programs is straight-line code, where one statement executes after the next without branching, as in Figure 4.12. In 4.12a, two sequential calls to `work` are shown, with the non-speculative stack evolution in 4.12b. In future evolutions we omit the initial N$\downarrow$ and final N$\uparrow\perp$. In 4.12c, speculation occurs on all calls to `work` : the parent thread τ executes `work(1)`, α executes `work(2)`, and β executes a continuation which does nothing useful. τ returns from w1 and commits α, then returns from w2 and commits β, and finally pops s" to exit the program. We label this execution as *inefficient*

```
straightline () {
  work (1);
  work (2);
}
```

(a) *Code.*

```
       w1        w2
 ⇒ s ⇒ s ⇒ s ⇒ s ⇒ s ⇒
τ   τ   τ   τ   τ   τ   τ
```

(b) *Do not speculate.*

```
     w1        w1 w2       w2
s ⇒ s—s' ⇒ s—s'—s"⇒ s'——s"⇒ s"
τ    τ  α    τ  α  β    τ—α  β    τ
```

(c) *Speculate on* `work(1)` *and* `work(2)` *(inefficient).*

```
     w1        w1 w2   w2
s ⇒ s—s' ⇒ s—s' ⇒ s' ⇒ s'
τ    τ  α    τ  α    τ    τ
```

(d) *Speculate on* `work(1)` *(good).*

```
     w1         w2
s ⇒ s ⇒ s ⇒ s—s' ⇒ s'
τ    τ    τ    τ  α    τ
```

(e) *Speculate on* `work(2)` *(bad).*

```
barrier () {
  work (1);
  stop; /* unsafe */
  work (2);
}
```

(f) *Speculation barrier code.*

```
     w1             w2
s ⇒ s—s' ⇒ s' ⇒ s' ⇒ s'
τ    τ  α    τ    τ    τ
```

(g) *Speculation barrier: speculate on* `work(1)`.

Figure 4.12: *Straight-line.*

because of the wasteful thread β. In 4.12d, w1 and w2 are effectively parallelized, without the wasteful β from 4.12c, and so we label the execution as *good*. Conversely, the execution in 4.12e is *bad* because w1 and w2 are serialized.

Even in this simple example, the choices between PUSH and FORK clearly affect which threads execute which regions of code, and whether they have useful work to do. In 4.12f, the keyword `stop` is introduced which acts as a speculation barrier. This keyword is a catch-all for any unsafe instruction and can be implemented as such. The result in 4.12g is that w2 is not executed speculatively. Again, although simple, the impact of unsafe instructions on speculative parallelism is important to consider; in some cases, speculation barriers might even be useful for inducing desired parallel runtime behaviours.

4.5.2 If-Then

Another simple code idiom is if-then conditional branching, as shown in Figure 4.13, with example code in 4.13a. 4.13b and 4.13c show the non-speculative execution depending on whether `work(1)` returns true or false respectively. If the value of the conditional is spec-

```
if_then () {
  if (work (1))
    work (2);
  work (3);
}
```

(a) *Code.*

$$\begin{array}{ccccccc} & w1 & & w2 & & w3 & \\ i \Rightarrow & i \Rightarrow & i \Rightarrow & i \Rightarrow & i \Rightarrow & i \Rightarrow & i \\ \tau & \tau & \tau & \tau & \tau & \tau & \tau \end{array}$$

(b) *Do not speculate,* `work(1)` *returns true.*

$$\begin{array}{ccccc} & w1 & & w3 & \\ i \Rightarrow & i \Rightarrow & i \Rightarrow & i \Rightarrow & i \\ \tau & \tau & \tau & \tau & \tau \end{array}$$

(c) *Do not speculate,* `work(1)` *returns false.*

$$\begin{array}{ccccccc} & w1 & w1\ w2 & w2 & & w3 & \\ i \Rightarrow & i - i' \Rightarrow & i - i' \Rightarrow & i' \Rightarrow & i' \Rightarrow & i' \Rightarrow & i' \\ \tau & \tau\ \alpha & \tau\ \alpha & \tau & \tau & \tau & \tau \end{array}$$

(d) *Speculate on* `work(1)`*, predict true correctly (good).*

$$\begin{array}{ccccc} & w1 & w1\ w3 & w3 & \\ i \Rightarrow & i - i' \Rightarrow & i - i' \Rightarrow & i' \Rightarrow & i' \\ \tau & \tau\ \alpha & \tau\ \alpha & \tau & \tau \end{array}$$

(e) *Speculate on* `work(1)`*, predict false correctly (good).*

$$\begin{array}{cccccc} & w1 & w1\ w2 & & w3 & \\ i \Rightarrow & i - i' \Rightarrow & i - i' \Rightarrow & i \Rightarrow & i \Rightarrow & i \\ \tau & \tau\ \alpha & \tau\ \alpha & \tau & \tau & \tau \end{array}$$

(f) *Speculate on* `work(1)`*, predict true incorrectly (bad).*

$$\begin{array}{cccccccc} & w1 & w1\ w3 & & w2 & & w3 & \\ i \Rightarrow & i - i' \Rightarrow & i - i' \Rightarrow & i \Rightarrow & i \Rightarrow & i \Rightarrow & i \Rightarrow & i \\ \tau & \tau\ \alpha & \tau\ \alpha & \tau & \tau & \tau & \tau & \tau \end{array}$$

(g) *Speculate on* `work(1)`*, predict false incorrectly (bad).*

Figure 4.13: *If-then.*

ulative, then the particular code paths followed depending on the value themselves become speculative. In Figure 4.13, when speculating on the call to `work(1)` it is necessary to predict a boolean return value. If the speculation is correct, as in 4.13d and 4.13e, then the speculative work w2 or w3 respectively is committed. Otherwise, that work is aborted, as in 4.13f and 4.13g.

For this speculation idiom to be useful, the function producing the return value should take a long time to execute. Nested ifs have similar behaviour to this example, although the prediction for the outer test will be more important than the inner test in terms of limiting wasted computation, since the inner speculation is under its control. Extensions to our speculation model could allow for multiple predicted return values, associating one speculative thread with each. This would provide a kind of speculative hedging, and may be worthwhile given excess resources.

4.5.3 Iteration

The most common code idiom considered for speculation is loop iteration. Chen & Olukotun demonstrated that if a loop body is extracted into a method call, then method level speculation can subsume loop level speculation [CO98]. We explore an example loop under different speculation assumptions in Figure 4.14 to better understand the behaviour. Outlining or extracting a loop body to convert it to the example form in 4.14a makes the loop amenable to method level speculation, with 4.14b showing basic non-speculative execution; conversely, inlining some or all of the code from `work` can be used to limit the amount of parallelism. Speculating on all calls to `work` in 4.14c, the loop is quickly divided up into one iteration per thread for as many threads as there are iterations.

To limit this aggressive parallelization, we explored speculating on every m in n calls. In 4.14d, a child is forked every 1 in 2 calls. The stack evolves to a point where both w1 and w2 are executing concurrently and no other stack operations are possible. Once w1 and w2 complete, a number of intermediate evolutions open up, but they all lead to the same state with w3 and w4 executing concurrently. Effectively, the loop is parallelized across two threads, each executing one iteration at a time. In 4.14e, speculating on every 1 in 3 calls, a similar pattern emerges, except that a non-parallel execution of w3 is interjected. In 4.14f, speculating on every 2 in 3 calls, w1, w2, and w3 execute in parallel, and once they complete the stack evolves until w4, w5, and w6 execute in parallel.

A general rule for iteration under MLS then is that speculating on every $n-1$ in n calls to `work` will parallelize the loop across n threads, each executing one iteration. To support multiple subsequent iterations executing in the same thread, there are two options. First, the parent thread could pass `i + j` to the child thread when speculating, where `j` is the number of iterations per thread; however, our model would need an explicit mechanism for modifying specific local stack frame variables to support this. Second, the loop could be unrolled such that multiple iterations were pushed into the parent method body, as shown in 4.14g.

```
iterate (n) {
  for (i = 1; i <= n; i++)
    work (i);
}
```

(a) *Code.*

```
     w1      w2      w3
i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ ...
τ   τ   τ   τ   τ   τ
```

(b) *Do not speculate.*

```
     w1       w1 w2        w1 w2 w3
i ⇒ i—i' ⇒ i—i'—i'' ⇒ i—i'—i''—i''' ⇒ ...
τ   τ α    τ α β      τ α β γ
```

(c) *Speculate on all calls to* `work`.

```
     w1       w1 w2   w2           w3          w3 w4
i ⇒ i—i' ⇒ i—i' ⇒ i' ⇒ i' ⇒ i'—i'' ⇒ i'—i''
τ   τ α    τ α    τ    τ    τ  β    τ  β
⇒ ...
```

(d) *Speculate on 1 in 2 calls to* `work`.

```
     w1       w1 w2   w2           w3                w4
i ⇒ i—i' ⇒ i—i' ⇒ i' ⇒ i' ⇒ i' ⇒ i' ⇒ i'—i''
τ   τ α    τ α    τ    τ    τ    τ    τ  β
   w4 w5
⇒ i'—i'' ⇒ ...
   τ  β
```

(e) *Speculate on 1 in 3 calls to* `work`.

```
     w1       w1 w2        w1 w2 w3     w2     w3
i ⇒ i—i' ⇒ i—i'—i'' ⇒ i—i'—i'' ⇒ i'——i''
τ   τ α    τ α β      τ α β      τ—α  β
   w3          w4          w4 w5              w4 w5 w6
⇒ i'' ⇒ i'' ⇒ i''—i''' ⇒ i''—i'''—i'''' ⇒ i''—i'''—i'''' ⇒ ...
   τ     τ     τ  γ       τ  γ   δ         τ  γ   δ
```

(f) *Speculate on 2 in 3 calls to* `work`.

```
unrolled (i) {
  work (i);
  work (i + 1);
}

iterate (n) {
  i = 1;
  while (i <= n)
    unrolled (i += 2);
}
```

(g) *Unrolled iteration code (*`n` *must be even).*

Figure 4.14: *Iteration.*

4.5.4 Tail Recursion

Tail recursion is explored in Figure 4.15, with example code shown in 4.15a and non-speculative execution shown in 4.15b. It is well known that tail recursion can be efficiently converted to iteration, and we see why in these examples: the only difference in stack behaviour is the interleaving `recurse` frames. Speculating on both `recurse` and `work` in 4.15c usefully populates the stack with successive calls to `work`. However, this also creates just as many wasteful threads that only ever fall out of the recursion, although they stop almost immediately as they encounter elder siblings. Speculating on just `work` in 4.15d is good, and yields a stack structure identical to that produced by speculating on all

```
recurse (i, n) {
  work (i);
  if (i < n)
    recurse (i + 1, n);
}
```

(a) *Code.*

```
                              w2
         w1          r2       r2
r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ . . .
τ      τ      τ      τ      τ
```

(b) *Do not speculate.*

```
                                  w2
        w1          w1 r2         w1 r2–r2'
r1 ⇒ r1–r1'⇒ r1–r1'–r1''⇒ r1–r1'——r1''⇒ . . .
τ      τ  α      τ  α  β      τ  α  γ  β
```

(c) *Speculate on all calls (inefficient).*

```
                              w2
        w1          w1 r2     w1 r2–r2'
r1 ⇒ r1–r1'⇒ r1–r1'⇒ r1–r1'    ⇒ . . .
τ      τ  α      τ  α      τ  α  β
```

(d) *Speculate on all calls to* `work` *(good).*

```
                                          w2
        w1            r2              r2
r1 ⇒ r1 ⇒ r1 ⇒ r1–r1'⇒ r1–r1'⇒ . . .
τ      τ      τ      τ  α      τ  α
```

(e) *Speculate on all calls to* `recurse` *(bad).*

```
                                                                          w4
                                                        w3        w3 r4     w3 r4
                              w2     w2          r3     r3–r3'    r3–r3'    r3–r3'
        w1          w1 r2     w1 r2     r2      r2      r2      r2         r2         r2
r1 ⇒ r1–r1'⇒ r1–r1'⇒ r1–r1'⇒ r1'⇒ r1'⇒ r1'⇒ r1'    ⇒ r1'    ⇒ r1'    ⇒ . . .
τ      τ  α      τ  α      τ  α      τ      τ      τ      τ  β      τ  β      τ  β
```

(f) *Speculate on 1 in 2 calls to* `work` *(good).*

```
                                                                                                w4
                                                                  w3              w3 r4           w3 r4
                              w2     w2          r3          r3       r3–r3'          r3–r3'          r3–r3'
        w1          w1 r2     w1 r2     r2      r2      r2–r2'     r2–        r2——            r2——            r2——
r1 ⇒ r1–r1'⇒ r1–r1'⇒ r1–r1'⇒ r1'⇒ r1'⇒ r1'     ⇒ r1' r1''⇒ r1'      r1''⇒ r1'      r1''⇒ r1'      r1''⇒ . . .
τ      τ  α      τ  α      τ  α      τ      τ      τ  β       τ  β       τ  γ  β       τ  γ  β       τ  γ  β
```

(g) *Speculate on 1 in 2 calls to* `work` *and* `recurse` *(inefficient).*

```
                                                                                           w4
                                              w3        w3                       r4        r4        r4
                            w2         r3       r3        r3        r3        r3        r3        r3
        w1          r2      r2     r2     r2–r2'    r2–r2'    r2–       r2–       r2–       r2–       r2–
r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1      ⇒ r1      ⇒ r1  r1'⇒ r1  r1'⇒ r1  r1'⇒ r1  r1'⇒ r1  r1'
τ      τ      τ      τ      τ      τ      τ  α      τ  α      τ  α      τ  α      τ  α      τ  α      τ  α

              w5            w5
   r5         r5            r5
   r4–r4'     r4–r4'        r4–
   r3         r3            r3  r3'
   r2——       r2——          r2——
⇒ r1     r1'⇒ r1     r1'⇒ r1     r1'⇒ . . .
   τ  β  α    τ  β  α       τ  β  α
```

(h) *Speculate on 1 in 2 calls to* `recurse` *(bad).*

Figure 4.15: *Tail recursion.*

calls in iteration, as in 4.14c, modulo the interleaving `recurse` frames. On the contrary, speculating on just `recurse` in 4.15e is bad, because calls to `work` are never parallelized.

Speculating on 1 in 2 calls to `work` in 4.15f yields another structure directly comparable to iteration, this time mirroring 4.14d, where w1 and w2 execute in parallel before the stack evolves to w3 and w4. Speculating on 1 in 2 calls to `work` and `recurse` in 4.15g is similar but introduces inefficiency. Speculating on 1 in 2 calls to `recurse` in 4.15h is bad, but yields interesting behaviour which sees speculative children unwind the stack by one frame before stopping.

4.5.5 Head Recursion

Head recursion is considered in Figure 4.16. Comparing the code shown in 4.16a with tail recursion, the call to `work` now comes after the call to `recurse` instead of before. This means that in the non-speculative execution shown in 4.16b, `work` is not executed until there are `n` `recurse` frames on the stack. Speculating on all calls to `recurse` and `work` in 4.16c is inefficient, just as for tail recursion, whereas speculating on just `recurse` in 4.16d is good, allowing for calls to `work` to be executed out-of-order. This is expected given that head recursion is seen as dual to tail recursion. Surprisingly, however, speculating on just `work` in 4.16e is also good: the stack gets unwound in-order. For head recursion, the support for both out-of-order and in-order nesting in our stack model, needed by 4.16d and 4.16e respectively, ensures that all available parallelism is obtained.

Speculating on 1 in 2 calls to `recurse` and `work` in 4.16f yields unbounded parallelism, where pairs of two calls are unwound in-order within a pair, and out-of-order between pairs. Speculating on 1 in 2 calls to `work` in 4.16g yields a bounded parallelism structure comparable to the iteration in 4.14d and the tail recursion in 4.15f, where first wn and wm execute in parallel, and then the stack evolves to a state where wl and wk execute in parallel.

We were again surprised by speculating on 1 in 2 calls to `recurse` in 4.16h: α executes w2, and after returning the stack evolves until it executes w1. This pattern is strikingly similar to the loop unrolling in 4.14g, where two successive calls execute in the same thread. This particular example is unbounded, however, because nothing prevents the growth of τ up the stack, such that every two calls to `work` start all together and are then unrolled all to-

```
recurse (i, n) {
  if (i < n)
    recurse (i + 1, n);
  work (i);
}
```

(a) *Code.*

```
                            wn
                    rn      rn      rn              wm
            rm      rm      rm      rm      rm      rm
    :       :       :       :       :       :       :
r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ ...
τ    τ    τ    τ    τ    τ    τ    τ
```

(b) *Do not speculate.*

```
                          r3
      r2       r2  w1     r2—r2' w1
r1 ⇒ r1—r1'⇒ r1—r1'—r1''⇒ r1——r1'—r1''
τ    τ  α    τ  α  β      τ  γ  α  β

    r3  w2
    r2—r2'—r2'' w1
⇒ r1————r1'—r1''⇒ ...
    τ  γ  δ  α  β
```

(c) *Speculate on all calls (inefficient).*

```
                                r3          r3  w2
         r2       r2  w1        r2—r2' w1   r2—r2' w1
r1 ⇒ r1—r1'⇒ r1—r1'⇒ r1———r1'⇒ r1———r1'⇒ ...
τ    τ  α    τ  α    τ  β  α    τ  β  α
```

(d) *Speculate on all calls to* `recurse` *(good).*

```
                               wn          wn         wn
                        rn     rn—rn'      rn—        rn—wm
                 rm     rm     rm          rm rm'     rm rm—rm''
     :      :      :      :              :          :
r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1       ⇒ r1       ⇒ r1
τ    τ    τ    τ    τ  α       τ  α       τ  α  β
⇒ ...
```

(e) *Speculate on all calls to* `work` *(good).*

```
                                          :
              r3          r3  w2          r3  w2
      r2      r2—r2'      r2—r2'—r2''     r2—r2'—r2''
r1 ⇒ r1 ⇒ r1       ⇒ r1          ⇒ r1
τ    τ    τ  α       τ  α  β       τ  α  β

    :            :
    r3  w2       r3  w2
    r2—r2'—      r2—r2'—w1
⇒ r1      r1'⇒ r1       r1'⇒ ...
    τ  α  β      τ  α  β
```

(f) *Speculate on 1 in 2 calls to* `recurse` *and* `work` *(unbounded parallelism).*

```
                                      wn        wn        wn        wn        wn        wn              wn
                               rn     rn—rn'    rn—       rn—wm     rn—       rn—       rn—             rn—
                        rm     rm     rm        rm rm'    rm rm'    rm rm'    rm        rm wl           rm wl
                 rl     rl     rl     rl        rl        rl        rl        rl  rl'   rl  rl'—rl''    rl  rl'—
          rk     rk     rk     rk     rk        rk        rk        rk        rk        rk              rk      rk'
    :      :      :      :      :      :         :         :         :         :         :               :
r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1    ⇒ r1     ⇒ r1     ⇒ r1     ⇒ r1     ⇒ r1          ⇒ r1
τ    τ    τ    τ    τ    τ    τ  α    τ  α    τ  α    τ  α    τ  α    τ  α  β       τ  α  β

    wn
    rn—
    rm wl         wl
    rl  rl'—wk    rl'———wk
    rk      rk'   rk      rk'
    :             :
⇒ r1          ⇒ r1          ⇒ ...
    τ  α  β       τ—α  β
```

(g) *Speculate on 1 in 2 calls to* `work` *(compare with Figures 4.14d and 4.15f).*

```
                             :          :         :          :          :
           r3         r3  w2  r3  w2    r3        r3         r3         r3
     r2    r2—r2'     r2—r2'  r2—r2'    r2—r2'    r2—        r2—w1      r2—
r1 ⇒ r1 ⇒ r1     ⇒ r1      ⇒ r1      ⇒ r1      ⇒ r1  r1'⇒ r1  r1'⇒ r1  r1'⇒ ...
τ    τ    τ  α     τ  α      τ  α      τ  α      τ  α       τ  α       τ  α
```

(h) *Speculate on 1 in 2 calls to* `recurse` *(comparable to loop unrolling in Figure 4.14g).*

Figure 4.16: *Head recursion.*

gether. In general, calls to work can be divided into batches of size b and distributed evenly across t threads, where $b = n/t$, by choosing to speculate on every 1 in b calls to `recurse`. The unrolling within a given batch is in-order, but the creation of batches themselves is out-of-order.

4.5.6 Mixed Head and Tail Recursion

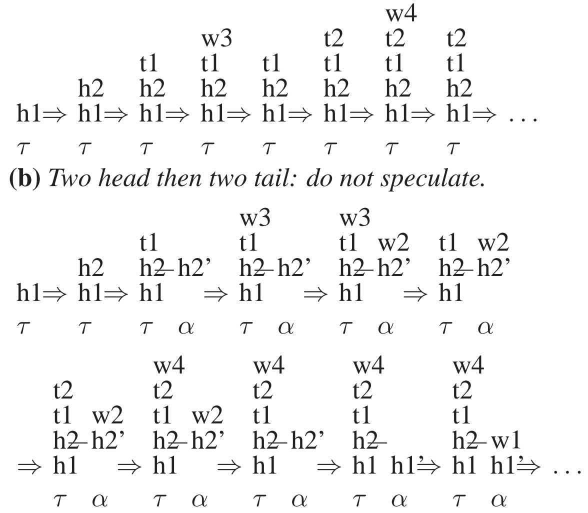
We next experimented with a mix of head and tail recursion, as shown in Figure 4.17. Given the interesting behaviours seen for these kinds of recursion in isolation, it seemed reasonable that a combination might yield even more interesting results. Tail recursion has two distinguishing properties under speculation: it provides in-order distribution across threads, and it prevents the forking thread from proceeding immediately to the top of the stack because useful work must complete first. On the other hand, head recursion is able to provide behaviour comparable to loop unrolling in a single thread. However, head recursion is uncapped and will always proceed immediately to the top of the stack.

The code in 4.17a constitutes a minimal example that uses head recursion to provide batch processing and tail recursion to limit stack growth. In 4.17b, the non-speculative evolution skips past the work in the head recursive calls, but stops to execute w3 and w4 tail recursively. In 4.17c, the repeating pattern is again two head recursive calls followed by two tail recursive calls, additionally speculating on `tail1` inside `head2`, the first tail recursive call. This creates a thread α that executes the first two calls to `work` out-of-order, while the parent thread τ executes the second two calls to `work` in-order. Except during brief periods of stack state evolution, there will only ever be two threads actively executing code, and the pattern established in the first four calls to `work` will repeat itself.

We can use this pattern to schedule batches of size b across t threads when the depth of the recursion is unknown or when only $b \times t$ calls should be scheduled at once. We need a pattern of $b \times (t-1)$ head recursive calls followed by b tail recursive calls, speculating on every $(cb+1)^{\text{th}}$ head recursive call in the pattern for $c \in \mathbb{N}_1$ and on the first tail recursive call in the pattern. A generalized `recurse` function that provides this behaviour is given in 4.17d; note that we have introduced a `spec` keyword here to indicate speculation points. As an example, to distribute work in batches of size 3 across 4 threads, use a pattern of 9

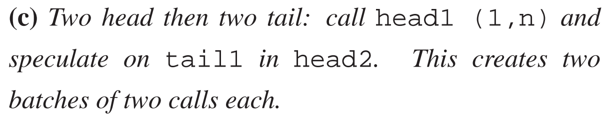
```
head1 (i, n) {
  head2 (i + 1, n);
  work (i);
}

head2 (i, n) {
  tail1 (i + 1, n);
  work (i);
}

tail1 (i, n) {
  work (i);
  tail2 (i + 1, n);
}

tail2 (i, n) {
  work (i);
  head1 (i + 1, n);
}
```

(a) *Two head then two tail code; call* `head1 (1,n)`.



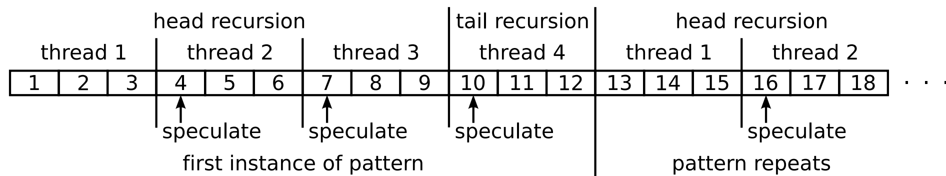
(b) *Two head then two tail: do not speculate.*

(c) *Two head then two tail: call* `head1 (1,n)` *and speculate on* `tail1` *in* `head2`. *This creates two batches of two calls each.*



```
recurse (i, n, b, t) {
  if (i < n && (i - 1) % (b * t) < b * (t - 1))
    if (i % b == 1 && i % (b * t) > b)
      spec recurse (i + 1, n, b, t);
    else
      recurse (i + 1, n, b, t);
  work (i);
  if (i < n && (i - 1) % (b * t) >= b * (t - 1))
    if (i % b == 1 && i % (b * t) > b)
      spec recurse (i + 1, n, b, t);
    else
      recurse (i + 1, n, b, t);
}
```

(d) *Mixed head and tail recursion code. To split work into multiple threads, call* `recurse (1,n,b,t)`, *where* `n` *is the number of calls to* `work`, `b` *is the batch size, and* `t` *is the number of threads. Speculation points are indicated by the* `spec` *keyword.*



(e) *Mixed recursion example, where* $b = 3$ *and* $t = 4$.

Figure 4.17: *Mixed head and tail recursion.*

head recursive calls followed by 3 tail recursive calls, and speculate on the 4th and 7th head recursive calls and the first tail recursive call, repeating the pattern for the next 12 calls. This example is illustrated in Figure 4.17e.

4.5.7 Olden Benchmark Suite

At this point we decided to examine the Olden benchmark suite [Car96] for speculation patterns. These are well-known parallelizable benchmarks written in C that manipulate irregular data structures. They were originally intended for parallelization with non-speculative futures. The differences between futures and MLS are that futures do not allow for any unsafe operations in the method continuation, such as consuming a predicted return value, futures do not typically allow the continuation code to return from the calling function, futures are an annotation based programming model that requires a programmer to insert them into the code, and the programmer must ensure the safety of all heap accesses. However, these differences are not irreconcilable, and there are many transferable results between futures and MLS. Manual parallelization of the Olden benchmarks using SableSpMT is part of our future work, as described in Section 6.2.2.

The results of a manual source code analysis are shown in Table 4.1. We analysed Olden version 1.01, the last published version of the suite. For each benchmark, we identified the parallel data structures being constructed, the functions that build the data structures, and the functions that traverse the data structures. We then looked for patterns in the parallel, future-based traversals, and mapped them to the speculation patterns outlined in this section. There were five different kinds of traversal in total: straight-line code speculation, iteration, head recursion, tree traversal, and divide and conquer. The first three kinds of traversal have already been considered in Figures 4.12, 4.14, and 4.16 respectively. We now consider tree traversal and divide and conquer algorithms.

4.5.8 Tree Traversals

An important consideration in parallelization is the traversal of irregular data structures. In Figure 4.18 we consider binary tree traversals. Here the traversals can be either preorder or postorder, and either fully parallel or only parallelized one tree level at a time. In 4.18a–

benchmark	data structure	parallel construction	parallel traversal	traversal pattern
bh	array of lists	uniform_testdata()	computegrav()	forall iteration
			stepsystem()	
	octtree	hackcofm()	freetree()	level-based postorder
bisort	binary tree	RandTree()	BiMerge()	level-based preorder
			Bisort()	level-based postorder
			SwapTree() 1	
	node	none	SwapTree() 2	straight-line
em3d	array of lists	do_all()	do_all_compute()	divide and conquer
health	quadtree of lists	alloc_tree()	get_results()	level-based postorder
mst	array of lists	AddEdges()	ComputeMst()	forall iteration
			Do_all_BlueRule()	divide and conquer
perimeter	quadtree	MakeTree()	perimeter()	level-based postorder
power	array of lists	build_lateral()	Compute_Tree()	forall iteration
	nested lists	build_lateral()	Compute_Lateral()	head recursion
		build_branch()	Compute_Branch()	
treeadd	binary tree	TreeAlloc()	TreeAdd()	level-based postorder
tsp	binary tree	build_tree()	tsp()	level-based postorder
union	quadtree	MakeTree()	copy()	level-based preorder
			TreeUnion()	
voronoi	binary tree	none	build_delaunay()	level-based postorder

Table 4.1: *Parallel traversal patterns in the Olden benchmarks.*

4.18d, we again use a `spec` keyword to indicate speculation points. These functions use speculation over straight-line code to split the traversal at each node into two threads. In 4.18e–4.18h the results of traversing a three-node tree with two leaves are shown. The pre-order traversals execute `work` before descending into the tree, whereas the post-order traversals execute `work` on ascending out of the tree. w1 is the work done by the parent node, whereas w2 and w3 represent the work done by the two child nodes. Although only binary trees are shown, these patterns are straightforwardly extended to trees with arbitrary arity.

```
tree (node *n) {
  if (leaf (n))
    work (n);
  else {
    spec work (n);
    spec tree (n->left);
    tree (n->right);
  }
}
```

(a) *Pre-order binary tree traversal, fully parallel.*

```
tree (node *n) {
  work (n);
  if (!leaf (n)) {
    spec tree (n->left);
    tree (n->right);
  }
}
```

(b) *Pre-order binary tree traversal, one level at a time.*

```
tree (node *n) {
  if (!leaf (n)) {
    spec tree (n->left);
    spec tree (n->right);
  }
  work (n);
}
```

(c) *Post-order binary tree traversal, fully parallel.*

```
tree (node *n) {
  if (!leaf (n)) {
    spec tree (n->left);
    tree (n->right);
    pause;
  }
  work (n);
}
```

(d) *Post-order binary tree traversal, one level at a time.*

```
                                  w2                w2                w2 w3
       w1           w1 t2         w1 t2         w1 t2  t3         w1 t2  t3
t1 ⇒ t1–t1'⇒ t1–t1'–t1''⇒ t1–t1'–t1''⇒ t1–t1'–t1''⇒ t1–t1'–t1''
τ      τ  α      τ  α  β      τ  α  β      τ  α  β      τ  α  β
```

(e) *Pre-order binary tree traversal with 2 leaves, fully parallel.*

```
                            w2          w2          w2 w3
       w1          t2       t2          t2  t3      t2  t3
t1 ⇒ t1 ⇒ t1 ⇒ t1–t1'⇒ t1–t1'⇒ t1–t1'⇒ t1–t1'
τ      τ      τ      τ  α     τ  α      τ  α      τ  α
```

(f) *Pre-order binary tree traversal with 2 leaves, one level at a time.*

```
                    w2          w2               w2 w3            w2 w3
       t2           t2          t2  t3           t2  t3           t2  t3  w1
t1 ⇒ t1–t1'⇒ t1–t1'⇒ t1–t1'–t1''⇒ t1–t1'–t1''⇒ t1–t1'–t1''
τ      τ  α      τ  α      τ  α  β      τ  α  β      τ  α  β
```

(g) *Post-order binary tree traversal with 2 leaves, fully parallel.*

```
                    w2          w2          w2 w3          w3     w3
       t2           t2          t2  t3      t2  t3     t2  t3     t3      t3              w1
t1 ⇒ t1–t1'⇒ t1–t1'⇒ t1–t1'⇒ t1–t1'⇒ t1–t1'⇒ t1'⇒ t1'⇒ t1'⇒ t1'
τ      τ  α      τ  α      τ  α      τ  α      τ  α      τ      τ      τ      τ
```

(h) *Post-order binary tree traversal with 2 leaves, one level at a time.*

Figure 4.18: *Tree traversals.*

The difference between the fully parallel traversal in 4.18a and the level-based traversal in 4.18b is that in 4.18a the call to `work` only stops stack evolution if it occurs at a leaf node, whereas in 4.18b evolution stops on every call to `work`. The difference between the fully parallel traversal in 4.18c and the level-based traversal in 4.18d is that in 4.18c both sub-trees are traversed while `work` is executed speculatively, whereas in 4.18d the call to `work` is not executed until both sub-trees have finished evaluation. To enforce this, we introduce a new `pause` keyword in 4.18d that ensures both the left and right sides of the tree have completed before beginning work on the parent node. This keyword can be implemented simply by stopping speculation until the most immediate ancestor thread has committed the current child. There is a subtle difference here with stopping speculation altogether, which would necessitate waiting for a non-speculative parent to commit the state, and would in turn serialize otherwise independent parallelism. `pause` is equivalent to “touch” in future-based systems. Adding this keyword to our system would allow for MLS to subsume futures.

The tree traversals in the Olden suite are the most complicated of the patterns we identified. In many cases there are additional conditions controlling the traversals, such as whether the node in question is a leaf or non-leaf node, or more generally what its relationship to other nodes is. In all cases level-based traversals are done. For the postorder traversals, this is presumably due to a lack of support for futures returning from the stack frame in which they were created; this would require a stack buffering mechanism like we describe for MLS. For the preorder traversals, it appears more a matter of convention not to speculate on the `work` parts.

4.5.9 Divide and Conquer

The divide and conquer pattern used by the Olden benchmarks is shown in Figure 4.19. In 4.19a, the work is divided among the number of processors `n`, with one call to `work` for each processor. The speculative stack in 4.19c evolves to form a tree structure, such that w0, w1, w2, and w3 execute in parallel. This code assumes an array of length `n`, such that the work could just as easily be parallelized using the forall iteration in Figure 4.14c; it is unclear why the authors chose a divide and conquer approach here. A more general

function that allows for per-processor batch sizes `b` that can be larger than 1 is shown in 4.19b. Here the batch size is most appropriately calculated as the array length divided by the number of processors.

```
divide (i, n) {
  if (n > 1) {
    spec divide (i + n/2, n/2);
    divide (i, n/2);
  } else
    work (i);
}
```

(a) *Divide and conquer code from Olden.*

```
divide (i, n, b) {
  if (n > b) {
    spec divide (i + n/2, n/2, b);
    divide (i, n/2, b);
  } else
    for (j = i; j < i + b; j++)
      work (j);
}
```

(b) *Generalized divide and conquer code.*

```
                                                                                                   w3
                              31            31       11       31  21, 11       31  21, 11  01,     31  21, 11  01,
      22        22  02,       22—22' 02,    22—22' 02—02'     22—22' 02—02'    22—22' 02—02'      22—22' 02—02'
04 ⇒ 04—04' ⇒ 04—04' ⇒ 04———04' ⇒ 04———04'  ⇒ 04———04'    ⇒ 04———04'    ⇒ 04———04'
τ     τ  α     τ  α     τ  β  α     τ  β  α  γ    τ  β  α  γ      τ  β  α  γ      τ  β  α  γ

   w3 w2              w3 w2 w1           w3 w2 w1 w0
   31  21, 11  01,    31  21, 11  01,    31  21, 11  01,
   22—22' 02—02'      22—22' 02—02'      22—22' 02—02'
⇒ 04———04'        ⇒ 04———04'        ⇒ 04———04'
   τ  β  α  γ         τ  β  α  γ         τ  β  α  γ
```

(c) *Divide and conquer called with* `divide (0, 4)`. Non-leaf stack frames are named after arguments `i` and `n`, concatenated in order to form two-digit strings.

Figure 4.19: *Divide and conquer.*

4.5.10 Discussion

We can see from these examples that the dynamic parallelization behaviour induced by MLS is not obvious, and that there are surely more interesting patterns to be found. The key lesson here is that we cannot take ordinary programs with call and return semantics, provide a set of parallelization operations that significantly perturbs the normal execution order, and expect to obtain dramatic performance results, especially if we do not understand the underlying behaviour. We can however use investigations of sequential program behaviour under our stack model to derive generalizations about program structure and the correlation with performance or lack thereof. We can also look at successful manual

parallelization, such as that found in the Olden benchmark suite, and use it to both validate patterns discovered through exploratory analysis and mine for new patterns that have applicability to both non-speculative and speculative code.

Method level speculation is a powerful system for automatic parallelization, particularly when relatively arbitrary speculation choices are permitted. The challenge is to restructure sequential code so that any inherent parallelism can be fully exploited. In general, parallel programming is an optimization, and thus cannot be divorced from knowledge of what different code structures imply for the runtime system if performance is to be maximized. Just as tail-recursion is favoured in sequential programs for its efficient conversion to iteration, so should other idioms in sequential programs be favoured for their efficient conversion to parallel code. Of course, the end goal is for a compiler to remove this optimization burden from the programmer wherever possible.

4.6 Conclusions and Future Work

Empirical studies of language implementation strategies can only provide so much understanding. For a strategy such as MLS, there is obviously significant performance potential, but the results can be confusing and moreover mired in system-specific performance details. At some point, formalizing the model and exploring behaviour in abstract terms can provide a fresh perspective.

As an enabling step for arbitrary child nesting, we presented a simple multithreaded custom allocator for child thread data structures that relies on knowledge about ownership dominator trees. It eliminates a major performance bottleneck in our system involving repeated allocation across a library interface of child thread structures each having 37 nodes. It also solves a producer / consumer problem where memory allocated from a freelist in one thread is freed to a freelist in another, which came to light in our attempt to support in-order nesting. The synchronization overhead and memory locality in our scheme are probably sub-optimal, and it would be interesting to see how well state-of-the-art general purpose multithreaded allocators could compete with the 37-fold reduction in calls to `malloc` and `free` that our solution provides. We would be quite excited to see recycling of aggregate data structures evolve into a general purpose memory management paradigm.

In the next major part of this chapter we developed a unified model of MLS that accounts for the major design variations involved in building such a system. The individual sub-models run from completely sequential to highly speculative, and exhaustively cover the core patterns of behaviour encoded in the unified model. This model is valuable purely from a specification standpoint, facilitating system comprehension, comparison, testing, and implementation. Preliminary work suggests our model is also suitable for a proof of MLS correctness. Once obtained, showing equivalence with other continuation-based parallelization systems could then be used to transfer proof results. Our model finally lends itself to rapid design prototyping of future extensions, which might normally require significant implementation effort to explore. One such extension is support for out-of-order speculative commits. These would work like in-order speculative commits, but instead allow for a speculative thread to merge with a child of some ancestor that it encountered upon returning from a call and attempting to buffer a stack frame.

The last part of this chapter details an exploration of MLS behaviour using our stack model as a tool for insight. We identified some key relationships between program structure, choice of fork point, and resultant speculation behaviour. In some cases we labelled them as good, bad, or inefficient in terms of exposing parallelism, and in others we used them to synthesize desirable higher level behaviours. These experiments demonstrated that all features of the speculation model are useful for creating parallelism, including both in-order and out-of-order nesting, and that robustness and flexibility in an MLS system are important. Our experience here is that accurately predicting how the parallel state will evolve without actually trying scenarios out on paper is overwhelmingly impractical. In the future, automated explorations of this nature may yield further insights. In general, we found that MLS behaviour is fragile, but that if understood it can be beneficially controlled. This is best demonstrated by our analysis of the Olden benchmark suite whose source code contains much domain-specific programming knowledge with respect to futures, a non-speculative parallelization construct closely related to MLS. It would be interesting to apply the speculation patterns identified so far to unparallelized benchmarks. We believe that maximizing parallelism will require a combination of programmer awareness, compiler transformation, profile information, and judicious static and/or dynamic forking decisions.

Finally, our visualization methods could be equally well applied to output from actual MLS implementations, including our own Java-based SableSpMT presented in Chapter 2. Our stack diagrams are unique for their compactness, linear scalability, uniform symbol density, lack of overlapping lines, and relation to actual data structures. The state evolutions require only a simple event trace along with unique thread and frame identifiers as inputs. This avenue could be useful for empirical studies of real-world programs.

Chapter 5
Related Work

Automatic parallelization has been a primary focus of research on parallel computing, compilers, and programming languages for many years [BCF+88]. Existing approaches have been most successful when analysing loop based, highly structured scientific applications, typically C or Fortran based [CLS96, GPZ00], though Java experiments have also been done [AGMM00]. Wolfe gives a comprehensive survey of standard techniques [Wol96]. Various studies have also examined analysis methods, seeking to better understand the applicability and performance of parallelization approaches [McK94, PW94, SMH98]. Designs and results for arbitrary, irregular, object-oriented programs remain less common, as even simple unknown information such as loop bounds can preclude parallelization.

Speculative multithreading (SpMT), also known as thread level speculation (TLS), is an optimistic technique that seeks to overcome the limitations of non-speculative parallelizing compilers, exposing parallelism in a broader class of applications. Kejariwal and Nicolau maintain an extensive bibliography of publications related to speculative execution [KN07]. Although work on SpMT dates back to the early 1990s and most experimental results are positive, there are still no production SpMT systems in either hardware or software.

At a high level, there are various granularities available for creating speculative threads. The coarsest model is method level speculation (MLS), for example [CO98] and of course this thesis, under which speculative method continuations execute in parallel with a non-speculative method body. Next is loop level speculation, for example [SCZM05], under which a parent thread will typically execute iteration i non-speculatively while iterations

$i+1, i+2, \ldots$ execute in successively more speculative child threads. Advanced schemes can handle nested loops and partition iterations more freely between threads. Basic block level speculation is the finest model, for example [BF04], wherein speculative threads begin and end on basic block boundaries, and can also be started at arbitrary points.

In addition to choice of speculation granularity, there are choices about whether to use a hardware or software architecture and which languages to handle. From the perspective of hardware loop level speculation for C, C++, and Fortran, SpMT is much more well-studied than software method level speculation for Java or comparably rich languages. Our initial software MLS for Java system is described in Chapter 2. On the one hand our choice of research topic is motivated by convincing prior work that includes one or two of the same high level choices, and on the other it is motivated by the desire to broaden the scope of SpMT applicability. Although most software studies have focused on loops in C programs, the viability of software loop level speculation for Java has been demonstrated [KL00]. Further, MLS has been identified as particularly appropriate for Java, given both the object-oriented method based structure of Java programs and the simplifications available due to explicit knowledge about stack, local, and heap operations [CO98]. Our choice of the Java language presents many challenges, which most related work treats as orthogonal and neglects to some degree. MLS can also subsume loop level speculation, as demonstrated in Chapter 4, albeit with extra invocation overhead. We discuss hardware SpMT approaches in Section 5.1, Java language issues in Section 5.2, software approaches in Section 5.3, and method level speculation in Section 5.4.

All SpMT implementations include some mechanism for handling dependence violations. We use a cache-like dependence buffer, sending speculative reads and writes through thread-local storage, as described in Chapter 2, although other systems use undo logging, writing directly to main memory. The dependence buffering mechanism from SpMT systems is highly similar to transactional memory, a form of speculative storage that is typically used to create atomic sections in parallel programs. An alternative approach is speculative locking, which executes the critical sections in existing lock-based programs speculatively. We discuss dependence buffering in Section 5.5, transactional memory in Section 5.6, and speculative locking in Section 5.7.

A naïve implementation of software or VM-based SpMT can result in relatively high

overhead costs, and so while SpMT itself is an optimization, second order optimization of the different SpMT operations involved is also critical, both in terms of final performance and suitability as a research tool. In this thesis the primary optimizations we propose are adaptive return value prediction, nested speculation and a supporting memory allocator, and structural fork heuristics based on observations of irregular program behaviour under speculation. In Section 5.8 we discuss related work on return value prediction, in Section 5.9 memory management, in Section 5.10 nested speculation, and in Section 5.11 high level strategies for handling irregular parallelism. More general heuristics for deciding where to fork threads are discussed in Section 5.12, and techniques for reducing misspeculations are discussed in Section 5.13.

Finally, there are several areas that are closely related to method level speculation, without falling under the general SpMT umbrella. In Section 5.14 we discuss non-speculative method level parallelism for imperative languages, which informs much of the design of our system. In Section 5.15 we discuss speculation for functional languages, which is actually non-speculative with respect to the potential for dependence violations. Lastly, we briefly consider other uses of speculative techniques in Section 5.16.

5.1 Hardware Architectures

Speculative multithreading approaches have been developed primarily in the context of novel hardware environments. A number of general purpose speculative architectures have been proposed, including the Multiscalar architecture [Fra93], the Superthreaded architecture [THA+99], trace processors [Rot99], MAJC [TCC+00], Hydra [HHS+00], and several other designs [KT99,SCZM00,FF01,OKP+01]. Hardware simulations have in general shown good potential speedups, given suitable timing assumptions; for example, Krishnan and Torrellas demonstrate that interprocessor communication speeds are a strong contributor to overall performance [KT01], and thread creation overhead is typically quite low, on the order of 10 cycles. Steffan *et al.* provide a more recent loop based implementation with STAMPede. They also give a good overview of the state of the art [SCZM05], as does Warg [War06]. Our choice to implement SpMT at the Java virtual machine level was motivated by hardware designs. In general, virtual machines are good platforms for explor-

ing software virtualizations of hardware techniques, especially considering that interpreter bytecode is often based on hardware instruction sets. Further, as an experimental platform, full hardware simulations often incur a 1000-fold slowdown [KT98], whereas we found a modified JVM to incur a 5-fold slowdown.

Only a few hardware studies consider Java programs explicitly. At the speculative hardware level, an executing Java virtual machine does not exhibit distinguished performance in comparison with other applications [WS01]. However, as an interpreted language, compiled Java programs can provide higher level abstractions and information than generic machine code, which has contributed to an interest in Java-specific studies. Chen & Olukotun pioneered work on method level speculation for Java using a modified Kaffe JVM and JIT running on the Hydra architecure [CO98]. They later developed TEST, which uses hardware speculative trace generation and analysis modules that cooperate with an online feedback-directed JIT compiler to improve runtime performance [CO03b]. Unlike their previous work on MLS, TEST focuses only on identifying candidates for loop level speculation. It is essentially a hardware profiling system that identifies speculative loop candidates for dynamic recompilation by a JIT compiler. The culmination of their work is Jrpm, an overall software / hardware hybrid design for dynamically parallelizing Java programs that uses the TEST hardware internally and runs on Hydra [CO03a]. They found speedups of 2-4x for a wide range of integer, floating point, and multimedia benchmarks using loop level speculation, observing only minor overhead for their particular hardware configuration. This work is for the most part comparable to other SpMT compilation efforts, with the biggest advantage of the JIT environment apparently being dynamic recompilation in response to online profiling. It is also the most robust demonstration that the JVM is a viable platform for speculation. As with their work on MLS, discussed in Section 5.4, they identified a number of manual changes that helped speculation. We examine their treatment of Java language features in Section 5.2. Traces of Java programs have also been applied to simulated architectures, by Hu *et al.* in their study of the impact of return value prediction on MLS performance [HBJ03], and by Whaley & Kozyrakis and Warg in their studies of heuristics for method level speculation [WK05, War06]. Finally, the cancelled MAJC processor was designed primarily for Java programs, and included SpMT support [TCC+00].

Most current hardware designs could in fact be classified as hybrid hardware / software

approaches since they rely to various extents on software assistance. Most commonly, compiler or runtime processing is required to help identify threads and insert appropriate SpMT directives for the hardware; in some cases software value prediction is used to reduce hardware costs. We discuss software support for hardware architectures as far as value prediction, fork heuristics, and misspeculation reduction are concerned in Sections 5.8, 5.12 and 5.13 respectively.

5.2 Language Semantics

Hardware speculation support, even with cooperating software support, largely obviates the consideration of high level language semantics: correct machine code execution implies correct program behaviour. Further, software-only SpMT architectures based on C or Fortran have relatively straightforward mappings to machine code. Accordingly, designs such as Softspec [BDA00], thread pipelining for C [Kaz00, KL01], and those by Rundberg *et al.* [RS01] and Cintra *et al.* [CL03] do not require a deep consideration of language semantics.

Supporting the Java language and virtual machine environment requires stronger guarantees and entails a much finer set of considerations. Cook considered many similar semantic issues in the context of supporting JVM rollback for debugging purposes [Coo02]. However, they have not been fully addressed by any prior Java SpMT implementation, including the details published by Sun for the cancelled Java-inspired MAJC processor [TCC$^+$00]. Largely this is because non-software or non-VM designs tend to elide treatment of complicated language safety issues to achieve results expediently at the (reasonable) cost of generality, whereas a software-only VM-only study presents no real choice. As part of their JIT compiler thread partitioning strategy in Jrpm, Chen & Olukotun do discuss Java exceptions, mark-and-sweep GC, and synchronization requirements [CO03a]. Jrpm allows speculative threads to throw and catch exceptions, whereas SableSpMT stops speculation on all exceptions. Jrpm also provides thread-local free lists for speculative object allocation, whereas SableSpMT synchronizes on a global heap lock. Finally, Jrpm ignores the constraints imposed by synchronization in speculative threads, a simplifying relaxation that is unsound for multithreaded applications, whereas SableSpMT is conservative in this re-

gard. The studies by Hu *et al.* [HBJ03] and Whaley & Kozyrakis [WK05] benefit from the same relaxation. Yoshizoe *et al.* describe a JVM with limited support for software-only loop speculation, but the short execution traces and limited environment under consideration preclude interactions with Java language issues [YMH98]. Kazi provides a pure Java source implementation and discusses exceptions, polymorphism, and GC, albeit without analysing them, but avoids the issue of dynamic class loading by assuming ahead-of-time whole program availability [Kaz00].

These are not *a priori* clearly insignificant differences; the effect of dynamic class loading in Java, for instance, has spawned a large number of non-trivial optimization considerations [AR02], and despite Kazi and Lilja's dismissal of GC as unimportant for applications with small footprints, many Java applications do have large memory requirements [DH99, DDHV03, BGH+06]. Differences and omissions such as the ones we have highlighted can make it difficult to compare Java studies, and leave important practical implementation questions open; our work here is meant to help rectify this situation.

5.3 Software SpMT

Whether applied to Java or not, hardware SpMT requires the expensive step of hardware construction and deployment, making an all-software system desirable. In practice, software-only approaches to SpMT are relatively uncommon. Rauchwerger & Padua developed the LRPD test, a first attempt at software-only loop level speculation, finding good results for previously unparallelizable loops in Fortran programs [RP95]. Gupta and Nim later improved on their work with a new set of runtime tests [GN98]. Papadimitriou and Mowry describe a system for C programs based on a virtual memory page protection mechanism [PM01]. Conflicting memory accesses between threads are caught and memory is synchronized using standard page trapping and signal handling. However, the high overheads encountered at this coarse granularity interfere with the viability of the approach.

Other approaches follow hardware designs more closely in terms of tracking individual memory access conflicts. Rundberg and Stenström describe a software approach to loop level speculation in C [RS01]. Their prototype implementation shows good speedup, but is verified only through hand done transformations and limited real world testing. Kazi and

Lilja describe a software model for *coarse-grained thread pipelining* and validate it through manual parallelization of loops in C programs [KL01], relying on their software library implementation of the SuperThreaded architecture by Tsai *et al.* [THA+99]. Softspec is a compiler and runtime system that parallelizes loops in C programs with stride-predictable memory references [BDA00]. The approach depends on machine code level offline profiling to identify independent loop bodies suitable for speculative execution. Cintra and Llanos describe a Fortran-based system that also speculates on loop bodies, exploiting both compiler analysis and runtime testing to identify shared variables and handle individual dependence violations [CL03]. They later explore the design space of software loop speculation more completely [CL05]. Finally, Frank describes the SUDS system for loop based parallelization that is a software-only system but designed for the specialized Raw processor [Fra03]. These approaches generally achieve good performance results, but none are based on Java or designed as experimental frameworks.

Only very limited studies on software-only speculation for Java have been done previously. Yoshizoe *et al.* give results from a partially hand-done loop level speculation strategy implemented in a rudimentary prototype VM missing core features such as garbage collection [YMH98]. They show good speedup for simple situations, but a lack of heap analysis limits their results. A more convincing analysis is given by Kazi and Lilja through manual Java source transformations of loops in Java programs [KL00, Kaz00]. Although source level transformations such as these are not ideal when compared to compiler or VM-based transformations, these studies showed that specialized hardware is not an absolute requirement, and that Java programs are viable candidates for speculation. The most interesting result of the study by Kazi and Lilja is that scalable speculation is even possible with native Java threads, provided thread granularities are coarse enough. Opposingly, Warg and Stenström argue that Java-based SpMT has inherently high overhead costs which can only be addressed through hardware support [WS01]. However, this conclusion is based on data from Java programs translated to C and subsequently executed on simulated hardware, not on an actual software system. Our data and analysis are significantly more comprehensive than prior studies. They indicate that while overheads can be quite high, there is sufficient potential parallelism to offset the cost. At the same time, exposing this parallelism is a challenge. Based on our experience, we believe high quality fork heuristics are more im-

portant than the performance of the underlying runtime system, but that key optimizations such as adaptive return value prediction and support for a combination of both in-order and out-of-order nesting are also important.

Our initial work on software MLS for Java directly inspired two M.Sc. theses. Costanza implemented a speculative system in Jikes RVM that uses static analysis to identify single-entry single-exit regions in leaf methods and execute them speculatively [Cos07]. The significant complexity of implementation is identified as a disadvantage of this model when compared with method level speculation. Schätti later partially implemented a method level speculation system called HotSpec in the Sun HotSpot JVM [Sch08]. The significant complexities of extending a production JVM ended up limiting development progress. On the basis of these two experience reports, we chose to pursue more generically applicable investigations into return value prediction, nested MLS, and fork heuristics rather than a complex JIT compiler implementation.

Libraries are a key mechanism for providing reusability in software development. As discussed briefly in this thesis, we have extracted the VM-independent speculation logic in SableSpMT into a separate library, libspmt [PVK07]. There are many software transactional memory libraries for a variety of languages that could provide efficient dependence buffer implementations; transactional memory is discussed more fully in Section 5.6. With respect to SpMT specifically, Oancea and Mycroft describe PolyLibTLS, a configurable library for loop level SpMT [OM08]. It provides configurable dependence buffer, thread behaviour, and thread management support through the use of C++ template meta-programming. They later use this library to evaluate *in-place* support for SpMT in the context of software loop level speculation for C++, showing that direct updates to main memory combined with an undo log can be more efficient than a traditional dependence buffer [OMH09].

There have been various approaches to software speculation that rely on programmer intervention. Ding *et al.* exploit offline profiles and manual source code changes to identify and expose *behaviour oriented parallelism* (BOP) by marking *possibly parallel regions*, achieving significant speedups using very coarse thread granularities [DSK+07]. Praun *et al.* describe a system for manual parallelization drawn from SpMT and transactions, also showing how profiling feedback is important in determining speculation points [vPCC07].

Welc *et al.* proposed *safe futures* for Java, making the non-speculative futures model speculative by allowing continuations to execute speculative reads and writes [WJH05]. These are an implementation of the Java 1.5 future construct that function similarly to MLS except that the programmer specifies where to create threads and speculation past consumption of the return value is not possible. They found good speedups of loops in easily parallelizable benchmarks. Zhang *et al.* considered safe exception handling for Java futures [ZKN07b], and Navabi *et al.* later presented a formal semantics for a higher-order functional language with first-class exceptions [NJ09]. The similarities at the VM level in terms of dependence buffering, stack buffering, exception handling, bytecode execution, scheduling, and rollback between MLS and safe futures mean that many of the advances and results are in turn transferable between them.

5.4 Method Level Speculation

There is significantly less work on method level speculation than loop level or basic block level speculation. According to Chen & Olukotun [CO98], Oplinger *et al.* were the first to propose the concept of MLS in a limit study for C programs that sought to identify the maximum amounts of loop and method level parallelism available [OHL99]. Similar limit studies of both loop and method level parallelism were done by Warg & Stenström [WS01] and Kreaseck *et al.* [KTC00]. Hammond, Willey, and Olukotun later designed the Hydra chip multiprocessor for SpMT that included MLS support [HWO98]. They later described techniques to improve performance via overhead reductions and compiler-assisted code restructuring [OHW99].

Chen & Olukotun concurrently described a more realistic method level speculation system for Java, which combined a modified version of the Kaffe JVM and JIT compiler running on the Hydra architecture [CO98]. They found encouraging amounts of speculative method level parallelism in JIT-compiled code, particularly for data parallel applications, with simulated speedups comparable to ours on a four-way machine. They also identified three types of manual source code changes that could improve speedups, sometimes dramatically. The first change was outlining or extracting loop bodies, effectively demonstrating how method level speculation can subsume loop level speculation, albeit at the

cost of extra invocation overhead. The second change was critical forwarding path reduction, which involves moving dependent reads down and the corresponding stores up, as discussed in Section 5.13. The third change was eliminating false dependences by moving writes from loop bodies into methods called by the loop. A limitation of their approach is that they did not consider call or operand stack dependences in their simulation; presumably this implies that speculative continuations cannot return from their initial frame, which requires stack buffering. Further, although their examples include support for in-order nesting, out-of-order nesting is not discussed. Finally, the only benchmark they considered from SPEC JVM98 was `javac`, although they did also examine `raytrace`, the single-threaded version of `mtrt`.

Since its initial development, support for method level speculation has been included in a number of SpMT systems, but has only been the primary focus of a few studies. Hu *et al.* considered the impact of return value prediction on method level speculation in a JVM [HBJ03]. Whaley & Kozyrakis considered a range of input sources and fork heuristics for a Java-based method level speculation system [WK05]. Warg explored techniques for reducing overhead costs in hardware method level speculation for C and Java programs that depends on a similar set of heuristics, specifically predicting thread lengths, success rates, and balances between parents and children [War06]. Finally, although not method level speculation in the sense of speculative continuation execution, Balakrishnan and Sohi describe a unique method based approach called program demultiplexing, in which methods are executed speculatively when their inputs become available, following a data-flow approach to parallelization [BS06].

Kejariwal *et al.* have performed various limit studies on the profitability of speculation which ultimately suggest that broader forms of speculative parallelism such as MLS are necessary to maximize performance. They showed that inner loop speculation for SPEC CPU2006 yields a 6% performance improvement for a thread creation overhead of 10 cycles, and only 1% if the overhead is 1000 cycles [KTG+07]. Results for loop speculation in SPEC CPU2000 were similarly pessimistic [KTL+06]. More recently they performed an MLS limit study using the SPEC CPU2006 benchmarks that was somewhat more optimistic [KGT+10b]. Here they exclude call graph cycles from their analysis to prevent unbounded dependence buffer growth. They found an upper bound of only 20% specu-

lative coverage for all benchmarks except `464.h264ref` which had a coverage of 50%. This suggests that speculative recursion may in fact be important for MLS peformance. Kejariwal *et al.* later analysed the impact of in-order nesting depth on speculation success [KGT+10a]. They considered that when a speculative thread creates a speculative thread, the doubly-speculative thread's success rate is dependent on its speculative parent's success rate. They observe that the success rates exponentially decrease with increased nesting depth. Of course, long dependence chains may be still viable if there is a high success rate for each participant thread.

5.5 Dependence Buffering

All SpMT systems rely on some kind of dependence buffering strategy to prevent out-of-order speculative execution from corrupting non-speculative execution. These strategies involve tracking both reads and writes and invalidating in the case of read-after-write, write-after-read, and write-after-write dependence violations. The specifics of our dependence buffering model are described in Chapter 2.

Garzaran *et al.* reviewed the extant literature and proposed a taxonomy for state buffering mechanisms in thread level speculation [GPL+05]. According to that taxonomy, our model supports Eager Architectural Main Memory (Eager AMM), as speculative threads write variables to a dependence buffer and not directly to main memory, and the buffer is committed immediately at join time along with the child stack. It also supports multiple tasks and multiple versions of variables per processor (MultiT&MV): per-processor helper threads begin execution of speculative children as soon as both a helper and child are available, and each child has its own dependence buffer. This design is recommended as the most effective in terms of benefits gained for the complexity of implementation.

With respect to software buffering approaches, Papadimitriou and Mowry described a software SpMT system based on a virtual memory page protection mechanism [PM01]. Conflicting memory accesses between threads are caught and memory is synchronized using standard page trapping and signal handling. Oancea and Mycroft reviewed software SpMT buffering mechanisms and provided configurable support for three kinds of buffer in PolyLibTLS, their software SpMT library [OM08]. First, a read-only buffer, where any

writes invalidate the speculation. Second, a lightweight serial commit buffer that exploits loop iteration behaviour to reduce dependency tracking costs. Third, a buffer that updates memory in-place and allows for parallel commits.

5.6 Transactional Memory

Transactional memory (TM) is another kind of optimistic concurrency that is complementary to SpMT, relying on a similar core mechanism of speculative code isolation based on dependence buffering and rollback. However, TM seeks to parallelize the critical sections in already multithreaded programs, whereas SpMT parallelizes single-threaded programs. Thus SpMT incurs the additional overheads of speculative thread management when compared to a TM system. Under TM, instead of using locks to create critical sections, programmers write *atomic sections* without specifying any particular monitor object. Larus & Rajwar reviewed the extensive research on both software and hardware TM systems [LR06], many of which could be used to provide an alternative to the SpMT dependence buffering discussed in Section 5.5. For example, Mehrara *et al.* and Raman *et al.* explore the use of software TM to parallelize loops in C programs [MHHM09, RKM+10]. Many of the same Java language considerations we consider arise in a TM environment, although the solutions differ because the underlying parallelization paradigm is different.

In Chapter 4 we provide a precise semantics for our MLS call stack model. There have been similar formalization efforts with TM, given widespread experience that although the transactional programming model appears simple, implementations vary considerably in terms of when concurrent operations may be performed, whether and how transactions may be nested inside each other, what visibility of intermediate calculations have, and how to ensure correctness with respect to underlying memory models. MLS and SpMT of course differ fundamentally from TM in that speculative execution is not user-specified and is also potentially unbounded. However the nesting models have some similarity, meaning that the correspondence between different MLS nesting strategies and transactional nesting could be interesting to explore. Although we have not formally proven safety and liveness properties, the stack abstraction we use is drawn from a rigourously tested implementation, and could form the basis for future proofs. Previously we developed an initial proof of

MLS correctness based on a simpler list-based abstraction [PVK09]. The correctness of SpMT systems in general appears more straightforward to demonstrate than correctness of TM systems; Kim *et al.* provide a simple proof of correctness of both hardware-only and compiler-assisted SpMT in the context of reference idempotency analysis [KlOE+01].

In terms of specific formal approaches, Jagannathan *et al.* use *Transactional Featherweight Java* to show serializability of both versioning and two-phase locking approaches to transaction control [JVWH05]. In another early effort, Scott gives a sequential specification of TM semantics [Sco06]. Other major differences exist in terms of transaction nesting and hence available parallelism. Harris *et al.* provide a composable abstraction for Haskell, including support for one form of nested transactions, although with limited parallelism [HMPJH05]. Moore & Grossman also use a small-step operational semantics to investigate different nesting forms, showing equivalence between weaker models that enable greater parallelism, and using a type system to verify correctness in terms of progress of transactional substeps [MG08]. Abadi *et al.* have a similar goal, also building a type-based approach to prove correctness. They develop a specialized calculus of automatic mutual exclusion, and use it to examine the impact of weak atomicity models. Guerraoui and Kapalka argue that *opacity* is a fundamental serialization criterion, and use that to show correctness, as well as complexity bounds [GK08].

5.7 Speculative Locking

Lock synchronization poses a potential problem for speculative execution: lock acquire and release operations affect the program globally, and conservatively require speculative threads to stop. However, locking itself is quite amenable to speculation, and optimizations are indeed possible. Speculative locking is closely related to transactional memory, the primary difference being that the locks are not written as transactions, but rather executed speculatively without programmer changes.

Martínez and Torrellas show how speculative locking can reduce the impact of contention on coarse-grained locking structures in a hardware system [MT02]. Rundberg and Stenström extended this model to allow *post facto* speculative lock acquisition reordering, which minimizes dependences to extract as much concurrency as possible [RS03]. In a re-

lated hardware context Rajwar and Goodman define a microarchitecture-based speculative lock elision system [RG01]. Welc *et al.* later demonstrated a software implementation in a Java virtual machine [WJH06].

Our current implementation causes speculative threads to stop speculating on synchronization. Under traditional speculative locking, non-speculative threads become speculative upon entering a critical section. However, a different approach becomes available when combined with an SpMT system, namely allowing already speculative threads to enter and exit critical sections. The profiling data in Chapter 2 reveal that even in single-threaded Java programs this second kind of speculative locking could be a useful optimization. We include both variants of speculative locking as potential future work in Section 6.2.1.

5.8 Return Value Prediction

Return value prediction is a kind of value prediction, a well-known technique for allowing speculative execution of various forms to proceed beyond normal execution limits. Value prediction has been researched for well over a decade, primarily in the context of novel hardware designs and constraints. A wide variety of value predictors have been proposed and examined, including simple computational predictors, more complex table-based predictors, machine learning based predictors, and hybrid implementations. Our work here extends previous investigations of RVP in a Java context [CO98,HBJ03,PV04b,PV04a,SB06] with practical explorations of accuracy, speed, and memory consumption in an adaptive, dynamic software-only environment. Further, our unification framework brings together many known value predictors that are suitable for RVP.

Burtscher *et al.* provide a good overview of basic value prediction techniques [BDH02]. As a general rule, accommodating more patterns and using more historical information can improve prediction accuracy, and generalizations of simple predictors, such as last N value prediction, have been studied by a number of groups [BZ99a, WF97, LS96]. Last N value prediction allows for short, repetitive sequences to be captured, and can yield good results; Burtscher and Zorn, for example, show a space-efficient last 4 value predictor can outperform other more complex designs [BZ99a]. Zhou *et al.* later provided the gDiff predictor, which is a global version of our last N stride predictor [ZFC03]. Yong *et al.*'s

revised stride data value predictor [YYX05] is essentially a last 2 stride predictor, such that its patterns are also captured by the last N value predictor. Most predictors can be further improved by incorporating statistical measures such as formal confidence estimates, although this does add extra complexity [BZ99b]. Limits on the possible success of value prediction strategies have also been analysed [WS01].

Gabbay introduced the stride predictor and last value predictor, as well as several more specialized predictors, such as the sign-exponent-fraction (SEF) and register-file predictors [Gab96]. Specialized predictor designs provide further ways to exploit value prediction where more general approaches work poorly. The SEF predictor, for instance, predicts the sign, exponent, and fraction parts of a floating point number separately. Although the sign and exponent are often highly predictable, the fraction is not, which usually results in poor prediction accuracy for floating point data. Tullsen and Seng extended Gabbay's register-file predictor to a more general register value predictor. It predicts whether the value to be loaded by an instruction into a register is already present in that register [TS99]. It may be worth considering a stack top predictor that is simply a register value predictor specialized for return values.

Pointer-specific prediction is also possible, an example being the address-value delta (AVD) prediction introduced by Mutlu *et al.* that predicts whether the difference between an address and the value at that address for a given pointer load instruction is stable [MKP06]. Marcuello *et al.* propose an increment-based value predictor [MTG99, MGT04] for value prediction within a speculative multithreading architecture. This predictor is like the 2-delta stride load value predictor, but is further differentiated by computing the storage location value stride between two different instruction address contexts.

Sazeides and Smith examine the predictability of data values produced by different instructions. They consider hardware implementations of last value, stride, and finite context method (FCM) predictors, showing the limits of predicability and the relative performance of context and computational predictors [SS97b]. Subsequent work considers the practical impact of hardware resource (table size) constraints on predictability [SS97a]. The original idea for the finite context method predictor comes from the field of text compression [BCW90]. Goeman *et al.* proposed the *differential* FCM (DFCM) predictor [GVdB01] as a way of further improving prediction accuracy. Burtscher later suggested an improved

DFCM index or hash function that makes better use of the table structures [Bur02]. We use Jenkins' fast hash to compute hash values because it is appropriate for a software context [Jen97].

Hybrid designs allow predictors to be combined, complementing and in some cases reinforcing the behaviour of individual sub-predictors. Wang and Franklin show that a hybrid value predictor achieves higher accuracy than its component sub-predictors in isolation [WF97]. To improve performance, Calder *et al.* studied techniques for filtering out instructions not worth predicting [CRT99] in the context of a hybrid predictor. The interaction of sub-predictors can be complex, and Burtscher and Zorn show that resource sharing as well as the impact of how the hybrid selects the best sub-predictor can significantly affect performance [BZ02]. Designs have thus been proposed to reduce hybrid storage requirements [BZ00], and to use selection mechanisms that reduce inappropriate bias, such as cycling between sub-predictors [SB05b], or the use of improved confidence estimators [JB06]. Sam and Burtscher argue that complex value predictors are not always necessary in optimal hybrid designs that maximize the efficiency of client applications [SB05a]. Examples of generic, non-hybrid predictors include those based on perceptrons from machine-learning [TK04, SH04].

Software value prediction, while less common, has also been investigated, usually in conjunction with a hardware design. For instance, Li *et al.* use static program analysis to identify value dependencies that may affect speculative execution of loop bodies, and apply selective profiling to monitor the behaviour of these variables at runtime [LDZN03]. The resulting profile is used to customize predictor code generation for an optimized, subsequent execution [DLL+04, LYDN05]. Liu *et al.* incorporated software value prediction in their POSH compiler for speculative multithreading and found a beneficial impact on performance [LTC+06]. The predictors are similar to those used by Li *et al.* [LDZN03], and handle return values, loop induction variables, and some loop variables. Hybrid approaches have also been proposed, which combine software with simplified hardware components in order to reduce hardware costs [BSMF08, Fu01, RVRA08].

Performance can also be improved through static compiler analysis. For example, Burtscher *et al.* analyse program traces to divide load instructions into *classes*, with different groupings having logically distinct predictability properties [BDH02]. Quiñones

et al. developed the Mitosis compiler for speculative multithreading that relies on pre-computation slices for child threads, predicting thread inputs in software but performing the speculation in hardware [QMS+05]. Du *et al.* use a software loop unrolling transformation to improve speculation efficiency, but also evaluate likely prediction candidates from trace data using a software cost estimation [DLL+04]. Code scheduling approaches that identify and move interthread dependencies so as to minimize the chance of a misprediction have been developed by Zhai *et al.* [ZCSM02]. A more general consideration of compiler optimization is given by Sato *et al.*, who analysed the effect of unrelated optimizations on predictability and found that typical compiler optimizations do not in general limit predictability [SHSA01]. Finally, in related work of our own, we developed a *return value use* analysis that determines if and how return values will be used, and a *parameter dependence* analysis that determines which parameters affect the return value of a method [PV04a].

Return value prediction is a basic component of MLS systems, where even simple last value and stride predictors can have a large impact on speculative performance [CO98, OHL99]. Hu *et al.* extend this early work by analysing data from Java traces, and use simulated hardware to make a strong case for return value prediction in MLS systems [HBJ03]. In particular, they find that return values are typically consumed between 10 and 100 machine instructions after a call, which means that accurate return value prediction can contribute significantly to increased thread lengths. They also introduce the parameter stride predictor we examine and give prediction results for SPEC JVM98. Singer and Brown consider theoretical limits on RVP by using information theory to determine the predictability of return values in Java programs, independent of any specific predictor design [SB06]. Our work builds on prior efforts by including new predictors, by extending the data collected, and by providing an optimized software implementation. In Chapter 2 we provide a software implementation of MLS that shows RVP has a beneficial impact on performance in a relative sense, but contributes to overall system slowdowns in an absolute sense. In Chapter 3, we provide optimizations to this base system that dramatically reduce overhead. In our system predictions are made by the speculative child thread in order to relieve the non-speculative parent thread of prediction overhead. In an even more aggressive approach to overhead reduction, Tuck and Tullsen explore multithreaded value prediction, which uses separate cores to predict the values for a single thread [TT05].

Finally, several of our new predictor designs are based on memoization, particularly suitable for RVP. Although our work is the first to address memoization in a value prediction setting, memoization is obviously a well known technique, one that has even been used to speed up speculative hardware simulations [SL98]. Effective memoization based compiler and runtime optimizations have also been described [DL04]. Note that unlike traditional memoization approaches, limitations due to conservative correctness are not necessary in our speculative environment. In a related investigation we developed a dynamic purity analysis for Java programs and used our memoization framework to non-speculatively memoize statically impure but dynamically pure methods [XPV07].

Type information is another vector for optimizing performance. Sato and Arita show that data value widths can be exploited to reduce predictor size; by focusing on only smaller bit-width values accuracy is preserved at less cost [SA00]. Loh demonstrates both memory and power savings by using data width information [Loh03], although the hardware context requires heuristic discovery of high level type knowledge. Sam and Burtscher later show that hardware type information can be efficiently used to reduce predictor size [SB04]. They also demonstrate that more complex and hence more accurate predictors have a worse energy-performance tradeoff than simpler predictors and are thus unlikely to be implemented in hardware [SB05a].

5.9 Memory Management

In Section 4.2, we describe a simple custom memory allocator for arbitrarily nested MLS. It uses per-thread and global freelists to recycle aggregate child thread data structures, allowing for memory to be allocated in one thread and freed in another without causing a producer / consumer problem.

Multiprocessor memory management is in general well-studied. Berger *et al.* designed Hoard, the first scalable malloc that uses per-processor and global heaps to bound memory consumption and avoid false sharing [BMBW00]. Dice and Garthwaite provided a mostly lock-free malloc that is 10 times faster than Hoard as originally published in some cases [DG02]. Michael later provided a completely lock-free allocator based on Hoard that offers further improvement [Mic04]. Schneider *et al.* demonstrated yet another scalable

multiprocessor malloc implementation [SAN06]. Hudson *et al.* described a scalable malloc for transactional memory [HSATH06]. Evans implemented `jemalloc`, the state-of-the-art multiprocessor malloc in FreeBSD [Eva06]. Larson and Krishnan observed that for server applications, only 2–3% of memory is allocated and freed in different threads [rLK98], which indicates that the producer / consumer allocation problem that arises under in-order MLS nesting is likely an outlier.

Freelists have also long been used in custom memory allocators. Data structure pooling and allocation-based ownership have been studied more recently. Hirzel *et al.* examine the connectivity of heap structures, finding that objects connected via pointers usually have similar lifetimes [HHDH02]. Boyapati *et al.* combine user-specified ownership types with region-based memory management [BSWBR03]. Lattner and Adve later provide a system for automatic pool or region allocation that segregates the memory required by an individual data structure into its own pool [LA05]. Mitchell subsequently examines the runtime structure of object ownership for many large real-world applications, identifying many dominator trees [Mit06].

Berger *et al.* found that custom memory allocation offered no improvement over the widely available Lea allocator for 6 out of 8 benchmarks in a survey, and that region-based allocation explained the performance improvement of the other two [BZM02]. Indeed, our solution to the producer / consumer allocation problem presented by in-order nesting is directly inspired by Hoard, and its multiprocessor behaviour is likely quite similar. Our model does present a specialization of the general purpose allocators in that it always uses one heap per thread and one thread per processor. This is an advantage in the common case of allocating from the local thread heap instead of the global heap because synchronization operations can be eliminated altogether. However, the real performance advantage comes from recycling the aggregate child thread data structures that form ownership dominator trees, because it reduces the number of allocator calls by a factor of the number of sub-objects in each tree. This eliminates a significant performance bottleneck in our system. If we were to use an existing general purpose allocator, we would need to rewrite the application to allocate much larger regions of memory and divide them up manually to accomodate sub-objects in order to achieve the same effect. Thus there are software engineering benefits in that a straightforward object-oriented application structure can be used. Further, the

allocator we describe is itself extremely simple to implement.

In future work, it would be interesting to generalize our memory management system by combining it with the different ideas from these works to create a general purpose multiprocessor allocator that returns usable pre-assembled data structures. One case where the data structure recycling might be immediately applicable comes from Suganuma *et al.*'s experience translating COBOL programs to Java [SYON08]. They found that in the initial automatic translation step, large classes containing many arrays of inner classes with multiple levels of nesting get generated to hold the original program data. If these classes are instantiated, used, and discarded frequently, for example once for each execution of a hot transaction, this can cause unacceptably frequent garbage collection. Their solution is to instantiate inner array elements lazily, since not all of them are necessary. An alternative approach mirroring our solution would be to maintain freelists of these frequently allocated, short-lifetime, large objects with unchanging structure and zero out fields as necessary.

5.10 Nested Speculation

There is prior work on both in-order and out-of-order nested speculation. Loop level speculation models almost always provide in-order nesting, such that one speculative loop iteration can spawn the next, whereas method level speculation often allows for out-of-order nesting. Renau *et al.* extend a model with unlimited in-order nesting to allow unlimited out-of-order nesting, for both methods and nested loops [RTL+05]. This contrasts with our work that began in Chapter 2 with unlimited out-of-order nesting and was extended in Chapter 4 to support unlimited in-order nesting. They propose a hardware architecture based on timestamps that is complex and does not translate easily to software. Our model, while not directly suitable for a hardware implementation due to its abstract nature, is quite suitable for software MLS and exploits the call stack to ensure correctly ordered commits, a nearly universal program structure.

Our stack abstraction also provides a simple framework for understanding, clarifying, and unifying method level speculation approaches. In general, the precise operations allowed on call stacks in most related work are somewhat obscured, which in turn makes performance comparisons difficult. For example, in their evaluation of fork heuristics for

Java MLS, Whaley and Kozyrakis claim to allow speculative threads to create speculative threads, which meets the definition of in-order nesting [WK05]. However, all of their examples actually demonstrate out-of-order nesting. Zhai briefly describes stack management for speculation [Zha05], but does not provide details on the complexities of entering and exiting stack frames speculatively. Zahran and Franklin examine return address prediction in a speculative multithreading environment [ZF02], and later consider entire trees of child tasks [ZF03], but do not provide a precise semantics for the return address stack they describe.

A key consideration in all of the work on SpMT and MLS that our approach in Chapter 4 elides is the impact of speculative data dependences. We consider the problem of managing data dependences not unimportant but orthogonal to the problem of creating efficient stack and thread interleavings. This makes our work on thread nesting applicable to non-speculative method level parallelism approaches as well, with a possible optimization being the elimination of unnecessary stack frame buffering. The most direct example of this is that many of our speculation patterns are found in the non-speculative Olden benchmarks, as discussed in Section 4.5.7. In general, the strong isolation properties assumed by speculative threads require some kind of dependence buffering or transactional memory subsystem, as described in Sections 5.5 and 5.6.

5.11 Irregular Parallelism

Method and continuation based parallelism can be particularly appropriate for programs based on recursive, dynamic data structures. However, Kulkarni *et al.* argue that this is a problem for speculative approaches, because many such programs employ worklist or fixed point iteration designs, where shared meta data structure updates can easily result in frequent conflicts or rollbacks [KPW+07]. As a solution they describe the Galois approach, which provides both ordered and unordered optimistic, concurrent iterators for specialized and high-level concurrency control. Mendez *et al.* extend this design with further optimizations to reduce rollbacks and workload processing costs [MLNP+10]. As an alternative, we argue it is possible to rewrite traditional while loop worklist and fixed point algorithms to use head recursion, which in turn enables head recursive speculation via the

techniques described in Section 4.5, such that the meta data structure manipulations can occur non-speculatively while the actual work on the underlying data structure is parallelized speculatively.

Data parallel solutions to the worklist problem have also been proposed. Lublinerman *et al.* describe a data parallel language wherein computation is performed locally and concurrently on *object assemblies*, which are disjoint pieces of the main data structure, following a strong data ownership model [LCČ09]. Assemblies are active objects, merging to acquire ownership of data required for computation and splitting to increase concurrency. This applies nicely to algorithms where data locality maps to computational locality, as in much of the Lonestar suite provided by Kulkarni *et al.* [KBPC09].

We have been unable to find a detailed *algorithmic* exploration of how irregular parallelism evolves at runtime that compares to our analysis in Section 4.5. Kulkarni *et al.* do provide a tool called ParaMeter for visualizing how available parallelism evolves over time from a performance perspective, but there is no consideration of the actual dynamic thread structures [KBI+09]. Since for any real program these thread structures will be incredibly complex, we consider our basic on-paper explorations of simple examples essential for an intuitive understanding of how to expose irregular parallelism. In terms of regular speculative parallelism, Prahbu and Olukotun demonstrate how TLS hardware support can simplify manual source level parallelization of loops in C programs, identifying some programming strategies [PO03], and later focus more explicitly on loop level speculation patterns [PO05].

High level knowledge of the underlying use of data structures provides an obvious advantage to any parallelization strategy. Cahoon and McKinley described compiler analyses to detect linked data structures in Java programs, and used this to improve the performance of JOlden through automatic prefetching [CM01]. Ghiya *et al.* also detected the use of parallelizable data structure traversals automatically [GHZ98]. They showed how to convert foreach to forall by traversing the data structure first and saving the nodes in a temporary array. Our suggestion above to convert while loop algorithms to head recursive algorithms is comparable, in that it saves the traversal information in the call stack.

Parallel program development and optimization can be facilitated by exploiting more general observations or patterns. For instance, Raghavachari and Rogers discuss properties

of several irregular benchmarks including Barnes-Hut, relating them to different language and algorithmic abstractions [RR95]. Specific pattern sets for concurrency have also been described; these extend from relatively simple paradigms such as master / worker, pipeline, and divide and conquer [CG90], to more complex designs based on specific synchronization constructs. Schmidt *et al.* provide an overview of many common designs [SSRB00]. Explicit high-level concurrency patterns provided by languages or their aspect-oriented extensions [CSM06] and by standardized libraries [Lea05a] enable direct concurrency control where it can be easily and appropriately applied. Our work on understanding and exposing implicit, speculative parallelism via a consideration of programming idioms and speculation points complements these explicit patterns, and could benefit from a more formal pattern-based approach, following Wilkins' general methodology for constructing pattern languages [Wil03].

5.12 Fork Heuristics

A program partitioning or thread decomposition strategy that chooses speculation points is necessary for all speculative parallelization. We refer to these strategies as *fork heuristics*. In Chapter 2 we experimented with a dynamic profile-based system that assigned priorities to speculative threads based on success rates and thread lengths. Given a lack of insight into the behaviour of this system, particularly when in-order nesting is enabled, we developed several *structural* fork heuristics in Chapter 4. These are essentially speculation patterns that recommend fork points based on program structure and on-paper knowledge of how the speculation will evolve at runtime.

Marcuello & González observe that at a coarse granularity, even the choice to speculate on loops or methods is a form of heuristic [MG02]. Hu *et al.* later observe that the method inlining performed by a JIT will change the set of fork points available under method level speculation [HBJ03]. They suggest that for this reason their system is able to extract thread lengths significantly longer than those observed by Warg and Stenström [WS01]. Our choice of interpreter-based MLS is thus a fork heuristic as well.

Several static techniques based on ahead-of-time compiler transformations for SpMT architectures have been suggested. Kim and Eigenmann describe a compiler that allows

for both explicit non-speculative multithreading as well as implicit inner loop-based speculative multithreading [KE01]. Bhowmik and Franklin describe a general compiler support framework including both loop-based and basic block-based thread partitioning that also supports out-of-order thread creation [BF04]. Johnson *et al.* transform and solve the partitioning problem using a weighted, min-cut cost model [JEV04]. Dou and Cintra developed another static approach based on estimated speedup that they claim could be parameterized by profiling data [DC07]. Indeed, various groups have investigated offline profile-based approaches to fork heuristics. Marcuello & González depend on a basic block profile to create speculative threads [MG02]. Liu *et al.* developed the GCC-based POSH compiler for C programs that again uses profiling data to enable loop and method level speculation [LTC+06]. Quiñones *et al.* developed the Mitosis compiler that relies on pre-computation slices to compute child thread inputs for threads partitioned at the basic block, loop, or method level, also using a profiler to identify dependences and model control flow [QMS+05]. Steffan *et al.* also use profiling support to identify suitable loops in STAMPede [SCZM05]. Finally, Du *et al.* use a misspeculation cost model, transforming loops to reduce dependence violations, with loop unrolling, value prediction, and profiling as enabling techniques.

Dynamic thread partitioning strategies have also been considered. These reduce preprocessing needs at the expense of runtime overhead. Codrescu and Wills describe a simple hardware algorithm based on dynamically partitioning the instruction stream into threads [CW00]. Gao *et al.* considered a model for dynamically creating threads in recursive programs, again targeting a hardware architecture [GLXN09]. They initially predict the structure of the recursion tree to create threads and then adapt the speculation to match actual outcomes. It could be interesting to combine this approach with our structural fork heuristics given their shared focus on irregular and often recursive programs. Finally, as discussed in Section 5.1, Chen & Olukotun's Jrpm uses TEST, a hardware tracing system, in combination with a JIT compiler to partition threads dynamically. Jrpm also transforms speculative code to reduce variable dependencies [CO03a].

Warg describes various MLS fork heuristics based on dynamic profile information gathered in a hardware environment [War06]. The first is *run-length prediction*, which measures parent thread lengths and disables forking if they do not meet some minimum

threshold. The second is *parallel overlap prediction*, which measures the time a parent and child spend executing in parallel, and again disables forking if it does not meet a certain minimum threshold. The third is *misspeculation prediction*, which unlike the techniques described in Section 5.13 that attempt to gracefully handle misspeculations, simply disables thread creation altogether once a misspeculation is identified. Whaley and Kozyrakis also studied MLS fork heuristics, and in particular how they behave when applied to Java programs [WK05]. They consider various sources of profile information in a systematic comparison. They use parent method execution time (analogous to Warg's run-length prediction), parent method store count, expected speedup obtained by executing parent and child in parallel as opposed to sequentially, and expected cycles saved by executing parent and child in parallel as inputs (analogous to Warg's parallel overlap prediction). The key result is that simple heuristics are actually more effective than multipass heuristics that take the program call graph into consideration, because they are more permissive and eliminate fewer threads.

All of the approaches described here target SpMT hardware, which is designed support relatively short thread lengths due to low overhead costs. For example, an average of 11–43 machine instructions per thread are obtained when the methodology in [JEV04] is applied to the SPEC INT2000 benchmarks. As such, these techniques are likely too low-level and fine-grained to be directly translated to a pure software environment, although the higher level concepts should be transferable to some degree. With respect to software systems based on dynamic profiling, such as our initial attempt at fork heuristics, Arnold *et al.* provide a comprehensive literature review of adaptive optimization in response to online feedback, a technique that is widely used in production virtual machines and dynamic compilation systems, particularly for languages such as Java where ahead-of-time profiling and analysis is impractical [AFG$^+$05].

5.13 Misspeculation Reduction

High thread overheads and limited CPU resources have motivated various attempts at reducing misspeculation rates. Steffan *et al.* demonstrated that when a speculative thread encounters a memory read that may violate a data dependence with some parent specula-

tive or non-speculative thread, it may be more beneficial for the thread to: 1) speculate as usual by simply loading the current value from the parent thread into a dependence buffer; 2) predict the value using a load value predictor; or 3) dynamically synchronize, by stalling and waiting for the dependence to be resolved [SCZM02]. In a similar study, Cintra and Torrellas employed a finite state machine for each dependence that begins by speculating, switches to value prediction if the number of misspeculations is high enough, and from value prediction switches to dynamic synchronization if the confidence of the predictor is low [CT02]. Each group found that for some benchmarks, a combination of optimizations outperformed any one technique in isolation.

Specific compiler optimizations have also been developed. Zhai *et al.* define and evaluate instruction scheduling optimizations based on the length of the *critical forwarding path*, or time between a definition in one thread and a dependent speculative use of the same variable in another [ZCSM02]. This can be quite effective at reducing stalls when variable dependencies between threads are enforced through synchronization. A flow analysis described in [KlOE+01] depends on a compiler to label memory references as *idempotent* if they need not be tracked in speculative storage and can instead access main memory directly. This reduces the overhead in terms of space and time of buffering the reads and writes of a speculative thread. For Java MLS no local variables are visible from other threads, and so can easily be designated idempotent without implementing an analysis. This means that values on a thread's stack need not be tracked in the speculative storage, an observation that we exploit throughout this thesis. Ooi *et al.* develop further optimizations that limit the size of speculative threads to fit within hardware buffering constraints [OKP+01]. Finally, a *probabilistic* points-to analysis can be used to predict the likelihood of violation in a speculative thread [CHH+03, SS06].

5.14 Non-Speculative Method Level Parallelism

Various software systems have been designed to support parallel execution in imperative languages at the method level without speculation. In general, these implementations exchange the complexity of speculative execution designs for the complexity of ensuring conservatively correct memory access orderings. We have focused on specifying and under-

standing the behaviour of MLS in terms of its interaction with the program call stack. Concurrent stack management is an important practical design concern, with many common aspects between various forms of method level parallelism, particularly for continuation-based approaches. Efficient models have been explored in general [HDB90], and also with respect to specific parallelization strategies.

The Cilk language is based on a sophisticated runtime environment for non-speculative method level parallelization with dynamic load balancing and scheduling, and is guided by explicit programmer specifications [FLR98]. The pure continuation passing style of Cilk simplifies implementation by ensuring stacks are empty upon method completion and hence do not overlap [BJK+95]. The zJava compiler and runtime system is a more recent and VM-related example. zJava depends on symbolic access paths computed at compile time to parallelize a program dynamically, without using programmer directives [CA04]. Method calls are executed in separate child threads, while the parent executes the method continuation until either a return value is consumed or a heap data dependence is encountered, at which point it blocks. A registry of running threads, methods, and heap regions is maintained to enforce sequential execution semantics. In another compiler-based study, Rul *et al.* parallelized `bzip2` at the function level by using profile-based knowledge of data dependences, a call graph, and an interprocedural data flow graph to identify function clusters operating on shared data structures and partition them into using pipeline-based parallelism [RVB07].

Goldstein *et al.* provide an efficient implementation of *parallel call* that also uses a thread creation model dual to ours, wherein the child task executes the method and the parent thread executes the continuation [GSC96]. They represent the concurrent stack by a coarsely linked structure of individually allocated *stacklets*, which are regions of contiguous memory that can store several frames. This eliminates the need to collect garbage frames, at the cost of occasionally allocating and linking new stacklets. Pillar is a new language that supports this abstraction [AGG+07]. Although parallel call was not designed to be speculative, a translation of the speculation rules and runtime behaviour patterns of our system could be used to extend it. Carlisle and Rogers use a similar approach in Olden, migrating futures to different processors and leaving their continuations on the stack [CR01]. The resulting concurrent stack management is minimized through the use of a simplified

spaghetti stack, a sequential interleaving of thread frames with potentially non-contiguous live stack segments [RCRH95]. Goldstein reviews the multithreaded stack design space for systems where the forking thread executes the continuation [Gol97], discussing the advantages and disadvantages of both stacklets and spaghetti stacks. Carlisle and Rogers also provide a detailed review of many non-speculative, imperative, parallel programming languages and runtime environments [CR01].

We compared our model to ones in which the parent thread executes the continuation in Section 1.2 in reference to Figure 1.1. The most significant challenge in adapting any of these models for speculation is in the support for stack frame buffering. Our understanding is that in non-speculative systems, a primary reason to execute the continuation in the forking thread is that the continuation stack frame does not need to be copied to a new thread. As such, none of these models have been designed with a mechanism for stack frame buffering in mind, and in fact are optimized for the case where it is unnecessary. Another concern that speculation presents is that non-speculative work must have a higher priority than speculative work, which is best supported by having the parent execute the method body, whereas in non-speculative systems it can be just as efficient to execute the continuation in the current thread and delay execution of the parent. We also feel that irrespective of buffering and speculation issues, our representation is easier to depict and understand visually, because each thread has its own stack and it scales cleanly to arbitrary numbers of threads and stack frames in 2D space. Nevertheless, our stack model is closest to Goldstein's stacklets in terms of implementation, and there may be specific low level optimizations therein that are transferable to ours.

As discussed in Section 4.5, futures often use a "get" or "touch" method that blocks the continuation before using the return value from the future. Although return value prediction would always allow MLS to proceed safely, in some cases get / touch is also used to control runtime parallelism, as in the case of level by level tree traversals that we illustrate. To provide this functionality, we suggest a "pause" keyword that only stops speculation until the immediate and possibly speculative parent has completed, rather than stopping continuation execution altogether until the non-speculative parent has committed the child. This appears sufficient to allow MLS to subsume futures, as well as safe futures, which are discussed in Section 5.3.

5.15 Functional Language Speculation

The focus on method calls as a means to achieve parallelism in MLS suggests an affinity for functional language contexts, where the relative lack and greater control of method side-effects reduces implementation complexity and suggests significant available parallelism. In fact, there has been much work on speculative evaluation in functional languages. However, due to a difference in nomenclature, speculation in a functional language context does not imply interaction with a dependence buffer or transactional memory subsystem as it does with MLS, but rather simply that function bodies can evaluate in parallel with function arguments. Thus data dependences require blocking, although parallel computation can resume once they are resolved.

Osborne developed speculative computation for Multilisp, wherein speculative evalation can be aborted, but again this differs from abortion under MLS: instead of aborting speculative computations because they are incorrect, the computations are aborted because they are unnecessary, and the abortion is a way to reclaim computation resources [Osb90]. Greiner and Blelloch attempt to unify existing work by defining a parallel speculative λ-calculus that helps model the performance and prove the time efficiency of lazy languages using speculative evaluation [GB99], including those supporting futures [Hal85, GG89], lenient languages wherein all subexpressions can evaluate speculatively [Tra88], and speculative approaches to parallel lazy graph reduction [TG95]. Baker-Finch *et al.* [BFKT00] subsequently showed that parallel lazy evaluation was equivalent to sequential. Finally, Sarkar and Hennessy argue that much parallelization work is too fine-grained, and describe a coarse-grained compiler based approach for partitioning parallel Lisp programs into subgraphs for macro dataflow [SH86].

In more practical work, Mattson, Jr. found that speculative evaluation in Haskell can be supported with low overhead [Mat93]. Ennals and Peyton Jones present a similar optimistic execution system that works together with the lazy evaluation model in Haskell [EJ03]. They provide an operational semantics, but do not model the stack explicitly nor use their semantics to visualize behaviour. Harris & Singh later extended this model to work with the Haskell thunk structure allocated in optimized programs [HS07]. In their feedback directed system, they use an offline profiling phase and subsequent work stealing at runtime

to extract implicit parallelism, obtaining significant speedup in pure software.

Given that these speculation models are inherently stack based, the stack abstraction in Section 4.3 could be modified to support them, clarifying their semantics in terms of low-level implementation behaviour, and then subsequently used as a tool for visualizing program behaviour. There is also the question of whether MLS supports lazy or eager evaluation. The uses of return values and heap and static variables under MLS are beyond eager, in that they are predicted or read and then used speculatively before even becoming available. However, the overall structure for parallelizing method calls and their continuations is quite similar to lazy evaluation. Consider `f(g())`: lazy evaluation allows calling `f()` without knowing the result of `g()`, whereas method level speculation executes `f()` as if `g()` has already returned.

5.16 Other Uses of Speculation

Techniques developed for speculation have also been employed in various other contexts. Eugster demonstrated a debugging environment for concurrent programs that is also based on the core idea of thread rollback [Eug03]. Here, saving state and rolling back execution allows for different scheduling choices to be replicated in debugging or exhaustively considered in testing. Persistent designs also require basic program state checkpointing to restore the system to a previous, interrupted execution [CV06, Tja99]. Concepts such as rollback and checkpointing are important to fault-tolerant schemes in general, allowing correctness to be ensured by saving state and replaying an execution if failure is detected. They also provides the basic mechanism for transactional execution, which can be used to give Java codelets ACID properties [RW02]. Finally, Oplinger and Lam show that speculation is particularly effective in the context of parallelizing monitoring functions in a monitor-and-recover design for software reliability [OL02].

Chapter 6
Conclusions & Future Work

This thesis presents the first comprehensive study of software method level speculation for Java. Our work was broken into three major milestones: building a working prototype, designing an advanced return value prediction system, and developing a model of nested speculation that could be used to identify structural fork heuristics. Our work ranges from the concrete to the abstract and can support many future research directions.

6.1 Conclusions

The first major milestone of this work was to design and implement a working prototype system. We introduced the SableSpMT framework as an extension of the SableVM Java bytecode interpreter. Supporting method level speculation in a language as rich as Java is non-trivial, particularly in a non-simulated software environment where problematic operations can neither be trivially elided by the simulation nor deferred to a speculative hardware subsystem, meaning that correctness is of paramount concern. We detailed all of the significant modifications and considerations related to the JVM environment, including mechanisms for preparing speculative methods for execution, actual steps in the execution process, and high level language concerns. We also designed SableSpMT to be suitable as an analysis framework. It is highly instrumented for a wide range of analyses, and easily extendable to include new approaches. It supports logging and trace generation, has a unique single-threaded debugging mode, and its source code is freely available. Although

we have focused on Java, we believe that our designs here are transferable to other languages, in particular to virtual machine environments, and that method level speculation can benefit well-structured but irregular programs in general.

Experimentally, we found that most language level concerns do not impact significantly on speculative performance, with notable exceptions being speculative object allocation, heap access, return value prediction, method entry and exit, and synchronization. Although we could show significant parallelism and thus relative speedup, actual speedup was precluded due to excessive speculation overheads. We identified three main sources of overhead during profiling and these informed our subsequent efforts: expensive return value prediction due to non-adaptivity, idle speculative processors due to a shortage of speculative threads, and an abundance of short threads due to naïve fork heuristics.

The second major milestone was to expand and optimize our return value prediction subsystem. We first introduced a new unification framework for specifying predictors, and brought together many known predictors from the hardware literature under it. We also introduced several predictors based on memoization, uniquely suitable for return value prediction. We then described a design for a novel hybrid predictor that identifies ideal sub-predictors on a per-callsite basis at runtime and specializes to them, eliminating calls to unused sub-predictors. Our framework is implemented in libspmt, a library for speculative multithreading that was created by refactoring the original SableSpMT implementation. New predictors are simple to add, and the RVP code is instrumented for easy data analysis.

Experimentally, we reconfirmed our result from the first milestone that while a naïve hybrid prediction strategy yields high accuracy, it also suffers from high speed and memory overhead, essentially incurring the summed cost of all sub-predictors. Our first adaptive hybrid was an offline predictor that required an initial profiling run to determine ideal predictors. A subsequent run that used these ideal predictors at startup achieved the same accuracy as the naïve design at a fraction of the overhead cost. This result indicates that ideal per-callsite predictors do in fact exist, at least for our benchmarks. We then developed an online hybrid predictor that uses a warmup period as well as specialization and despecialization thresholds to find ideal predictors at runtime. This predictor performed acceptably close to the performance of the naïve and offline strategies in terms of accuracy, and dramatically better than the naïve strategy in terms of overhead. We thus considered

our first optimization objective complete.

The third major milestone was composed of three steps. We first described a memory allocator for child threads. The first feature of this allocator was already present in the initial milestone, namely recycling of aggregate data structures in order to prevent excessive calls to `malloc` and `free`. The second feature, migrating blocks of child tasks between local and global heaps, was novel and was required to prevent a producer consumer problem that arises under in-order nesting, namely one where memory allocated in one thread is freed in another. Following development of our allocator, we then described an abstract model of arbitrarily nested speculation, which included previously missing support for in-order nesting. This model was based on stack operations and drawn from the implementation of speculative call stacks in libspmt, which was again refactored from the original SableSpMT implementation. Finally, we used this model to explore speculation behaviour via visualization of stack state evolutions. Our allocator is simple to reimplement, our stack model can be used for both abstract reasoning and concrete implementation, and our visualizations are straightforward to understand.

Experimentally, we found that the memory allocator was dramatically faster than the system allocator, and that it also solved a memory leak problem under in-order nesting. We also found that enabling in-order nesting had the desired effect of providing idle processors with extra work, satisfying our second optimization goal, but that now the work available using our initial “always fork” strategy was so much that meaningful analysis was precluded. Thus the real experimental analysis in this chapter was a search for better fork heuristics, in turn satisfying our third optimization goal. This search consisted of numerous on-paper explorations of speculation behaviour, starting from simple programming idioms and observing how the stack state evolved as speculative threads were forked according to various conditions. We were able to identify some patterns as better than others, thus deriving a form of structural fork heuristic. We also demonstrated how they could be combined with each other to yield higher level patterns. When we examined the source code for a set of benchmarks parallelized with futures, which are closely related to method level speculation, we found many instances of our patterns as well as several new ones.

6.2 Future Work

Over the course of this thesis research, many interesting avenues for future exploration appeared, as discussed at the end of Chapters 2, 3, and 4. The SableSpMT framework we developed provides a good scaffolding for such work. We now consider a variety of such projects immediately possible within our system.

6.2.1 Speculative Locking

As discussed in Section 5.7, speculative locking is an SpMT extension that allows for speculative execution of critical sections. Implementing speculative locking requires two main extensions to our system. The first allows non-speculative threads to enter contended critical sections without holding the locks protecting them and thus become speculative. This is similar to typical transactional memory systems [LR06]. The second allows threads that are already speculative to enter and exit critical sections. This can be achieved by recording lock and unlock operations and not sharing buffer entries across these barriers. In both cases, accounting for the constraints and idiosyncrasies of the Java memory model is a significant challenge. Given support for the first kind of speculative locking, it would be interesting to combine it with our previous work on lock allocation that decides which objects should protect the critical sections in a program [HPV07]. We could first allocate the locks statically and then apply speculative locking to improve performance. The result would be an efficient implementation of optimistic atomic sections.

6.2.2 Manual Parallelization

SableSpMT works safely with both in-order and out-of-order nesting for arbitrary Java programs with efficient return value prediction. The major barrier to speedup is that the system does not know where to create speculative threads. However, *manual* parallelization of certain benchmarks following the patterns identified in Section 4.5 should yield actual speedup. Although a focus on manual parallelization does not directly support the goal of automatic MLS, this work would have several benefits. First, it would demonstrate the actual viability of our system. Second, it would likely expose additional system optimization

opportunities. Third, it would serve as an oracle for automatic application of speculation patterns based on static analysis. Fourth, it would provide a language level mechanism for experimentation with MLS.

Initially, we would expect synthetic micro-benchmarks based on the examples in Section 4.5 to asymptotically approach the performance predicted by our speculation patterns as the `work` functions increased in running time. In terms of methodology, these benchmarks could be manually annotated with our suggested `spec`, `pause`, and `stop` keywords, and then read in by a modified version of Soot's Java source code front end. Soot could then communicate the annotations to SableSpMT via a combination of classfile attributes and the current method for inserting forks and joins. The next step would be to parallelize JOlden, a sequential version of the Olden benchmarks written in Java [CM01]. Given that Olden is known to be parallelizable without speculation on data dependences or return values, we would also expect an explicit manual annotation of fork points for speculation to yield actual speedup. Following this, it should be possible to manually parallelize the Java Grande and OO7 benchmarks used for safe futures [WJH05], also with comparable speedup. Finally, the Lonestar suite of benchmarks [KBPC09] is known to be parallelizable using the Galois approach to speculation [KPW+07]. Many of these benchmarks depend on a loop-based worklist algorithm for computing fixed points. It would be most interesting to convert these benchmarks to use head recursive fixed point algorithms for parallelization with out-of-order method level speculation.

6.2.3 Static Analysis

Static analyses for SpMT have the potential to provide significant information to the runtime system. In particular, a set of static fork heuristics could complement or at least prime our dynamic fork heuristic system. These would be based on static estimations of method invocation and continuation length, method invocation and continuation purity, and return value use and predictability. They could also search for the speculation patterns identified in Chapter 4. Static analyses can also be used to restructure code for more efficient speculation. First, by advancing non-speculative writes and delaying dependent speculative reads, the chances of dependence violations in child threads can be reduced. This technique is

known as critical forwarding path minimization [ZCSM04]. Second, by extracting loop bodies into methods, method level speculation can subsume loop-level speculation, albeit with extra method call overhead [CO98]. A good name for this technique is outlining, the opposite of the well-known inlining optimization technique. Third, generalized code restructuring to fit known speculation patterns combined with precise marking of speculation points could make the results in Chapter 4 applicable in non-obvious situations. Fourth, moving fork points forwards into the method or backwards into the pre-invoke instruction can change the balance between parent and child thread lengths. Fifth, this rebalancing can also be achieved by moving the instructions surrounding a callsite into the method body, or by moving instructions from the method body into the pre-invoke or continuation instructions.

6.2.4 Dynamic Purity Analysis for Speculation

The side effects of a pure method are either invisible or in some way contained. We previously developed a dynamic purity analysis for Java programs [XPV07]. Integrating online purity information with our SpMT system would provide two benefits. First, entirely pure methods could be simply memoized, thus eliminating unnecessary speculation overhead. Second, quantifying the extent of dynamic purity on a fork point basis could provide another input to our fork heuristics module.

6.2.5 JIT Compiler Speculation

SableSpMT was initially implemented around SableVM, a Java bytecode interpreter. After refactoring, the code specific to SableVM became a client of libspmt, a general purpose SpMT library. Other clients of libspmt would afford different research challenges and opportunities, chief among them a just-in-time (JIT) compiler. As part of this thesis research, we implemented initial support for speculation inside the IBM Testarossa JIT compiler for the IBM J9 VM, and this was the primary motivation for creating libspmt. A JIT compiler runs approximately 10x faster than SableVM, so relative overheads could reasonably be estimated to be 10x higher. However, the presence of method inlining changes the call graph significantly, in turn changing the behaviour of speculation, possibly for the better;

Hu *et al.* attributed the positive difference between their results [HBJ03] and those of Warg & Stenström [WS01] to the presence of inlining. A direct comparison of the effect of inlining alone could be made by exporting the inlined call graph from a JIT compiler client of libspmt and importing it into SableSpMT. A JIT compiler could also take advantage of specific hardware support for speculation, whether simulated or actually existing, by generating code for the target architecture, much like the Jrpm system [CO03a]. Finally, there are interesting engineering problems, such as unsafe inlining of known generic C library code into generated JIT code, switching between speculative and non-speculative code in the presence of register usage, and deciding which methods to compile for speculation.

6.2.6 New Systems

It would perhaps be the most interesting to build new systems for new languages based on this thesis. In our review of related work, loop level speculation was a prominent feature of many designs. Our work here was focused primarily on new directions, but including explicit support for loops is an important optimization in any real system. It may or may not be possible to provide this support efficiently with MLS. Non-speculative parallelization is similarly important and will often expose a significant amount of parallelism, at least for regular, numerical programs. As far as method level speculation proper is concerned, it seems reasonable to implement our stack model and memory manager to provide both in-order and out-of-order nesting. Our examination of speculation patterns showed that both are necessary to maximize parallelization opportunities. The dependence buffering subsystem would best be provided by 3rd-party transactional memory, whether in the form of a software library, compiler, or hardware implementation. Forwarding dependence buffer values from a sequentially earlier speculative thread to a sequentially later one is a useful technique for reducing misspeculations. Adaptive return value prediction is useful for method level speculation, and can be implemented straightforwardly and efficiently. Reusing the simpler predictors for software load value prediction is an obvious optimization. In truth, many of these components are provided by libspmt and are not tied explicitly to SableSpMT. Rather than rewrite MLS from scratch and contend with a host of new bugs, it might make the most sense to adapt libspmt to a new language, compiler, or VM environ-

ment, even if it involves aggressive refactoring. Given its object-oriented nature, it might make sense to convert libspmt from C to C++.

At the VM level, safety is a primary concern, and it is important to get the language semantics correct. In our experience, the best stress test for MLS was to attempt speculation at every method call for an industry standard benchmark suite. As far as optimizations are concerned, speculative object allocation and synchronization are key areas. Related work has demonstrated that it is possible build a JIT compiler for speculation without supporting speculation at the interpreter level. The most aggressive dynamic recompilations in a JIT compiler focus on frequently executed or "hot" methods, but in the case of MLS, it is also important to optimize continuation code for speculation. There is a great body of work on compiler analysis for speculation. It would be sensible for any new system to include either an AOT or JIT compiler that implements existing techniques. Profiling support, whether online or offline, is extremely valuable. As suggested above for specific enhancements to SableSpMT, simple language level keywords for controlling method level speculation and a compiler analysis that can insert them based on structural fork heuristics would be useful. As long as parallelizing compilers are not sufficiently advanced, there will be value in safe and simple manual parallelization constructs that mirror the safety and simplicity of atomic sections for transactional memory.

Bibliography

[AFG$^+$05] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, February 2005.

[AGG$^+$07] Todd Anderson, Neal Glew, Peng Guo, Brian T. Lewis, Wei Liu, Zhanglin Liu, Leaf Petersen, Mohan Rajagopalan, James M. Stichnoth, Gansha Wu, and Dan Zhang. Pillar: A parallel implementation language. In *LCPC'07: Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing*, volume 5234 of *LNCS: Lecture Notes in Computer Science*, pages 141–155, October 2007.

[AGMM00] Pedro V. Artigas, Manish Gupta, Samuel P. Midkiff, and José; E. Moreira. Automatic loop transformations and parallelization for Java. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 1–10, 2000.

[AR02] Matthew Arnold and Barbara G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*, volume 2374 of *LNCS: Lecture Notes in Computer Science*, pages 498–524, 2002.

[BCF+88] Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, Vivek Sarkar, and David Shields. Automatic discovery of parallelism: a tool and an experiment (extended abstract). In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 77–84, 1988.

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, February 1990.

[BDA00] Derek Bruening, Srikrishna Devabhaktuni, and Saman Amarasinghe. Softspec: Software-based speculative parallelism. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[BDH02] Martin Burtscher, Amer Diwan, and Matthias Hauswirth. Static load classification for improving the value predictability of data-cache misses. In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 222–233, June 2002.

[BF04] Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(8):713–724, August 2004.

[BFKT00] Clem Baker-Finch, David J. King, and Phil Trinder. An operational semantics for parallel lazy evaluation. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 162–173, 2000.

[BGH+06] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06:*

Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 169–190, October 2006.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *PPoPP'95: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, July 1995.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, November 2000.

[BS06] Saisanthosh Balakrishnan and Gurindar S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *ISCA'06: Proceedings of the 33rd International Symposium on Computer Architecture*, pages 302–313, June 2006.

[BSMF08] Ioana Burcea, Stephen Somogyi, Andreas Moshovos, and Babak Falsafi. Predictor virtualization. In *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–167, March 2008.

[BSWBR03] Chandrasekhar Boyapati, Alexandru Salcianu, Jr. William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI'03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 324–337, June 2003.

[Bur02] Martin Burtscher. An improved index function for (D)FCM predictors. *ACM SIGARCH Computer Architecture News*, 30(3):19–24, June 2002.

[BZ99a] Martin Burtscher and Benjamin G. Zorn. Exploring last n value prediction. In *PACT'99: Proceedings of the 8th International Conference on Parallel Architectures and Compilation Techniques*, pages 66–77, October 1999.

[BZ99b] Martin Burtscher and Benjamin G. Zorn. Prediction outcome history-based confidence estimation for load value prediction. *JILP: Journal of Instruction-Level Parallelism*, 1:1–25, May 1999.

[BZ00] Martin Burtscher and Benjamin G. Zorn. Hybridizing and coalescing load value predictors. In *ICCD'00: Proceedings of the 2000 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 81–92, September 2000.

[BZ02] Martin Burtscher and Benjamin G. Zorn. Hybrid load-value predictors. *TC: IEEE Transactions on Computers*, 51(7):759–774, July 2002.

[BZM02] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *OOPSLA'02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12, November 2002.

[CA04] Bryan Chan and Tarek S. Abdelrahman. Run-time support for the automatic parallelization of Java programs. *Journal of Supercomputing*, 28(1):91–117, April 2004.

[Car96] Martin C. Carlisle. *Olden: Parallelizing Programs With Dynamic Data Structures On Distributed-Memory Machines*. PhD thesis, Princeton University, Princeton, New Jersey, USA, June 1996.

[CBM+08] Călin Caşcaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, 6(5):46–58, September 2008.

[CFE97] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 259–269, December 1997.

[CG90] Nicholas Carriero and David Gelernter. *How to write parallel programs: a first course*. MIT Press, Cambridge, MA, USA, 1990.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, USA, 1999.

[CHH+03] Peng-Sheng Chen, Ming-Yu Hung, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *PPoPP'03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 25–36, June 2003.

[CL03] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*, pages 13–24, June 2003.

[CL05] Marcelo Cintra and Diego R. Llanos. Design space exploration of a software speculative parallelization scheme. *TPDS: IEEE Transactions on Parallel and Distributed Systems*, 16(6):562–576, June 2005.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[CLS96] Jyh-Herng Chow, Leonard E. Lyon, and Vivek Sarkar. Automatic parallelization for symmetric shared-memory multiprocessors. In *CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, page 5, 1996.

[CM01] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT'01: Proceedings of the*

2001 International Conference on Parallel Architectures and Compilation Techniques, pages 280–291, September 2001.

[CO98] Michael K. Chen and Kunle Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT'98: Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques*, pages 176–184, October 1998.

[CO03a] Michael K. Chen and Kunle Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA'03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 434–446, June 2003.

[CO03b] Michael K. Chen and Kunle Olukotun. TEST: A tracer for extracting speculative threads. In *Symposium on Code Generation and Optimization (CGO '03)*, March 2003.

[Coo02] Jonathan J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, 45(6):608–619, May 2002.

[Cos07] Gianmatteo Costanza. Speculative multithreading in a Java virtual machine. Master's thesis, Laboratory for Software Technology, Institute of Computer Systems, Department of Computer Science, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, March 2007.

[CR01] Martin C. Carlisle and Anne Rogers. Supporting dynamic data structures with olden. In *Compiler Optimizations for Scalable Parallel Systems*, volume 1808 of *LNCS: Lecture Notes in Computer Science*, pages 709–750, 2001.

[CRT99] Brad Calder, Glenn Reinman, and Dean M. Tullsen. Selective value prediction. In *ISCA'99: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 64–74, May 1999.

[CSM06] Carlos A. Cunha, João L. Sobral, and Miguel P. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *AOSD*

'06: Proceedings of the 5th international conference on Aspect-oriented software development, pages 134–145, 2006.

[CT02] Marcelo Cintra and Josep Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 43–54, February 2002.

[CV06] Antonio Cunei and Jan Vitek. A new approach to real-time checkpointing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*, pages 68–77, June 2006.

[CW00] Lucian Codrescu and D. Scott Wills. On dynamic speculative thread partitioning and the MEM-slicing algorithm. *Journal of Universal Computer Science*, 6(10):908–927, 2000.

[DC07] Jialin Dou and Marcelo Cintra. A compiler cost model for speculative parallelization. *ACM Transactions on Architecture and Code Optimization*, 4(2):12, 2007.

[DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *OOPSLA'03: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 149–168, October 2003.

[DG02] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *ISMM'02: Proceedings of the 3rd International Symposium on Memory Management*, pages 163–174, June 2002.

[DH99] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *ECOOP*, volume 1628 of *LNCS: Lecture Notes in Computer Science*, pages 92–115, 1999.

[DL04] Yonghua Ding and Zhiyuan Li. A compiler scheme for reusing intermediate computation results. In *CGO'04: Proceedings of the International Symposium on Code Generation and Optimization*, page 279, March 2004.

[DLL+04] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI'04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 71–81, June 2004.

[DSK+07] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–234, June 2007.

[Duf04] Bruno Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, Montréal, Québec, Canada, October 2004.

[EJ03] Robert Ennals and Simon Peyton Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP'03: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 287–298, August 2003.

[Eug03] Pascal Eugster. Java virtual machine with rollback procedure allowing systematic and exhaustive testing of multi-threaded Java programs. Master's thesis, ETH Zürich, Zürich, Switzerland, March 2003.

[Eva06] Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *BSDCan'06: The Technical BSD Conference*, April 2006.

[FF01] Renato J. Figueiredo and José A. B. Fortes. Hardware support for extracting coarse-grain speculative parallelism in distributed shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 214–226, September 2001.

[FK03] Roy Friedman and Alon Kama. Transparent fault-tolerant Java virtual machine. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS '03)*, pages 319–328, October 2003.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, June 1998.

[Fra93] Manoj Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin–Madison, Madison, Wisconsin, USA, 1993.

[Fra03] Matthew Frank. *SUDS: Automatic Parallelization for Raw Processors*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 2003.

[Fu01] Chao-Ying Fu. *Compiler-Driven Value Speculation Scheduling*. PhD thesis, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, North Carolina, USA, May 2001.

[Gab96] Freddy Gabbay. Speculative execution based on value prediction. Technical Report 1080, Electrical Engineering Department, Technion – Israel Institute of Technology, Haifa, Israel, November 1996.

[Gag02] Etienne M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, December 2002. `http://sablevm.org`.

[GB99] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 21(2):240–285, 1999.

[GG89] Ron Goldman and Richard P. Gabriel. Qlisp: Parallel processing in Lisp. In *HICSS'89: Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, volume 2, pages 751–760, January 1989.

[GHZ98] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. Detecting parallelism in C programs with recursive data structures. In *CC'98: Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *LNCS: Lecture Notes in Computer Science*, pages 159–173, March 1998.

[GK08] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP'08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, February 2008.

[GLXN09] Lin Gao, Lian Li, Jingling Xue, and Tin-Fook Ngai. Exploiting speculative tlp in recursive programs by dynamic thread prediction. In *CC'09: Proceedings of the 18th International Conference on Compiler Construction*, volume 5501 of *LNCS: Lecture Notes in Computer Science*, pages 78–93, March 2009.

[GN98] Manish Gupta and Rahul Nim. Techniques for speculative run-time parallelization of loops. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–12, 1998.

[Gol97] Seth Copen Goldstein. *Lazy Threads: Compiler and Runtime Structures for Fine-Grained Parallel Programming*. PhD thesis, University of California at Berkeley, Berkeley, California, USA, 1997.

[GPL+05] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(3):247–279, September 2005.

[GPZ00] Eladio Gutiérrez, Oscar Plata, and Emilio L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 78–87, 2000.

[GSC96] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *JPDC: Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[GVdB01] Bart Goeman, Hans Vandierendonck, and Koen de Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *HPCA'01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 207–216, January 2001.

[Hal85] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[HBJ03] Shiwen Hu, Ravi Bhargava, and Lizy Kurian John. The role of return value prediction in exploiting speculative method-level parallelism. *JILP: Journal of Instruction-Level Parallelism*, 5:1–21, November 2003.

[HDB90] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *PLDI'90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 66–77, 1990.

[HHDH02] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *ISMM'02: Proceedings of the 3rd International Symposium on Memory Management*, pages 36–49, June 2002.

[HHS+00] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, March 2000.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 48–60, 2005.

[HPV07] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 353–364, September 2007.

[HS07] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. In *ICFP'07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 251–264, October 2007.

[HSATH06] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *ISMM'06: Proceedings of the 2006 International Symposium on Memory Management*, pages 74–83, June 2006.

[HWO98] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[JB06] Sandra J. Jackson and Martin Burtscher. Self optimizing finite state machines for confidence estimators. In *WISA'06: Proceedings of First Workshop on Introspective Architecture*, February 2006.

[Jen97] Bob Jenkins. A hash function for hash table lookup. *Dr. Dobb's Journal*, September 1997.

[JEV04] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*, pages 59–70, June 2004.

[JVWH05] Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, 2005.

[Kaz00] Iffat H. Kazi. *A Dynamically Adaptive Parallelization Model Based on Speculative Multithreading*. PhD thesis, Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN, USA, September 2000.

[KBI+09] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *PPoPP'09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–14, February 2009.

[KBPC09] Milind Kulkarni, Martin Burtscher, Keshav Pingali, and Calin Cascaval. Lonestar: A suite of parallel irregular programs. In *ISPASS'09: Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76, April 2009.

[KE01] Seon Wook Kim and Rudolf Eigenmann. The structure of a compiler for explicit and implicit parallelism. In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 2624 of *LNCS: Lecture Notes in Computer Science*, pages 336–351, August 2001.

[KGT+10a] Arun Kejariwal, Milind Girkar, Xinmin Tian, Hideki Saito, Alexandru Nicolau, Alexander V. Veidenbaum, Utpal Banerjee, and Constantine D. Polychronopoulos. Exploitation of nested thread-level speculative parallelism on multi-core systems. In *CF'10: Proceedings of the 7th ACM International Conference on Computing Frontiers*, pages 99–100, May 2010. Short paper from poster session.

[KGT+10b] Arun Kejariwal, Milind Girkar, Xinmin Tian, Hideki Saito, Alexandru Nicolau, Alexander V. Veidenbaum, Utpal Banerjee, and Constantine D. Polychronopoulos. On the efficacy of call graph-level thread-level speculation. In *WOSP/SIPEW'10: Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 247–248, January 2010. Short paper from poster session.

[KL00] Iffat H. Kazi and David J. Lilja. JavaSpMT: A speculative thread pipelining parallelization model for Java programs. In *IPDPS'00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, pages 559–564, May 2000.

[KL01] Iffat H. Kazi and David J. Lilja. Coarse-grained thread pipelining: A speculative parallel execution model for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):952–966, September 2001.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. Bibl. Matematica. North-Holland Publishing Company, Amsterdam, 1952.

[KlOE+01] Seon Wook Kim, Chong liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Reference idempotency analysis: A framework for optimizing speculative execution. In *PPoPP'01: Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 2–11, June 2001.

[KN07] Arun Kejariwal and Alexandru Nicolau. Speculative execution reading list, 2007. `http://www.ics.uci.edu/~akejariw/SpeculativeExecutionReadingList.pdf`.

[KPW+07] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222, June 2007.

[KT98] Venkata Krishnan and Josep Torrellas. A direct-execution framework for fast and accurate simulation of superscalar processors. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 286–293, October 1998.

[KT99] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, September 1999.

[KT01] Venkata Krishnan and Josep Torrellas. The need for fast communication in hardware-based speculative chip multiprocessors. *International Journal of Parallel Programming*, 29(1):3–33, February 2001.

[KTC00] Barbara Kreaseck, Dean M. Tullsen, and Brad Calder. Limits of task-based parallelism in irregular applications. In *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*, pages 43–58, October 2000.

[KTG+07] Arun Kejariwal, Xinmin Tian, Milind Girkar, Wei Li, Sergey Kozhukhov, Utpal Banerjee, Alexander Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using SPEC CPU2006. In *PPoPP'07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 215–225, March 2007.

[KTL+06] Arun Kejariwal, Xinmin Tian, Wei Li, Milind Girkar, Sergey Kozhukhov, Hideki Saito, Utpal Banerjee, Alexandru Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *ICS'06: Proceedings of the 20th Annual International Conference on Supercomputing*, pages 24–35, June 2006. Updated digital library version.

[LA05] Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 129–142, June 2005.

[LCČ09] Roberto Lublinerman, Swarat Chaudhuri, and Pavol Černý. Parallel programming with object assemblies. In *OOPSLA'09: Proceeding of the 24th ACM*

SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, pages 61–80, October 2009.

[LDZN03] Xiao-Feng Li, Zhao-Hui Du, Qingyu Zhao, and Tin-Fook Ngai. Software value prediction for speculative parallel threaded computations. In *VPW1: Proceedings of the 1st Value-Prediction Workshop*, pages 18–25, San Diego, CA, June 2003.

[Lea00] Doug Lea. A memory allocator, April 2000. First published in 1996. `http://gee.cs.oswego.edu/dl/html/malloc`.

[Lea05a] Doug Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, December 2005.

[Lea05b] Doug Lea. The JSR-133 cookbook for compiler writers, May 2005. `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`.

[LG09] Shaoshan Liu and Jean-Luc Gaudiot. Potential impact of value prediction on communication in many-core architectures. *IEEE Transactions on Computers*, 58(6):759–769, June 2009.

[Lia99] Sheng Liang. *The Java Native Interface. Programmer's Guide and Specification*. Addison–Wesley, Reading, Massachusetts, 1st edition, June 1999.

[Loh03] Gabriel H. Loh. Width-partitioned load value predictors. *JILP: Journal of Instruction-Level Parallelism*, 5:1–23, November 2003.

[LR06] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, December 2006.

[LS96] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, December 1996.

[LSC+08] Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, Angelos D. Keromytis, and Salvatore J. Stolfo. Return value predictability profiles for

self-healing. In *IWSEC'08: Advances in Information and Computer Security: Third International Workshop on Security*, volume 5312 of *LNCS: Lecture Notes in Computer Science*, pages 152–166, November 2008.

[LTC+06] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS compiler that exploits program structure. In *PPoPP'06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, March 2006.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, 2nd edition, 1999.

[LYDN05] Xiao-Feng Li, Chen Yang, Zhao-Hui Du, and Tin-Fook Ngai. Exploiting thread-level speculative parallelism with software value prediction. In *ACSAC'05: Proceedings of the 10th Asia-Pacific Computer Systems Architecture Conference*, volume 3740 of *LNCS: Lecture Notes in Computer Science*, pages 367–388, October 2005.

[Mat93] James Stewart Mattson, Jr. *An effective speculative evaluation technique for parallel supercombinator graph reduction*. PhD thesis, University of California at San Diego, La Jolla, California, USA, 1993.

[McK94] Kathryn S. McKinley. Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors. In *ICS '94: Proceedings of the 8th international conference on Supercomputing*, pages 54–63, 1994.

[MG02] Pedro Marcuello and Antonio González. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA '02)*, pages 55–64, February 2002.

[MG08] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL'08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 51–62, January 2008.

[MGT04] Pedro Marcuello, Antonio González, and Jordi Tubella. Thread partitioning and value prediction for exploiting speculative thread-level parallelism. *TC: IEEE Transactions on Computers*, 53(2):114–125, February 2004.

[MHHM09] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *PLDI'09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 166–176, June 2009.

[Mic04] Maged M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI'04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 35–46, June 2004.

[Mit06] Nick Mitchell. The runtime structure of object ownership. In *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *LNCS: Lecture Notes in Computer Science*, pages 74–98, July 2006.

[MKP06] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Address-value delta (AVD) prediction: A hardware technique for efficiently parallelizing dependent cache misses. *TC: IEEE Transactions on Computers*, 55(12):1491–1508, December 2006.

[MLNP+10] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPoPP'10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, January 2010.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 378–391, January 2005.

[MT02] José F. Martínez and Josep Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, October 2002.

[MTG99] Pedro Marcuello, Jordi Tubella, and Antonio González. Value prediction for speculative multithreaded architectures. In *MICRO 32: Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 230–236, November 1999.

[NJ09] Armand Navabi and Suresh Jagannathan. Exceptionally safe futures. In *COORD'09: Proceedings of the 11th International Conference on Coordination Models and Languages*, volume 5521 of *LNCS: Lecture Notes in Computer Science*, pages 47–65, June 2009.

[OHL99] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *PACT'99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, October 1999.

[OHW99] Kunle Olukotun, Lance Hammond, and Mark Willey. Improving the performance of speculatively parallel applications on the hydra cmp. In *ICS'99: Proceedings of the 13th International Conference on Supercomputing*, pages 21–30, June 1999.

[OKP+01] Chong-Liang Ooi, Seon Wook Kim, Il Park, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*, pages 368–380, 2001.

[OL02] Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[OM08] Cosmin E. Oancea and Alan Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE'08: Proceedings of the 1st International Workshop on Multicore Software Engineering*, pages 23–32, May 2008.

[OMH09] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. A lightweight in-place implementation for software thread-level speculation. In *SPAA'09: Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, pages 223–232, August 2009.

[Osb90] Randy B. Osborne. Speculative computation in Multilisp. In *LFP'90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 198–208, June 1990.

[PB09] Salil Mohan Pant and Gregory T. Byrd. Extending concurrency of transactional memory programs by using value prediction. In *CF'09: Proceedings of the 6th ACM Conference on Computing Frontiers*, pages 11–20, May 2009.

[PM01] Spiros Papadimitriou and Todd C. Mowry. Exploring thread-level speculation in software: The effects of memory access tracking granularity. Technical Report CMU-CS-01-145, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, USA, July 2001.

[PO03] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, 2003.

[PO05] Manohar K. Prabhu and Kunle Olukotun. Exposing speculative thread parallelism in spec2000. In *Proceedings of the tenth ACM SIGPLAN symposium*

on Principles and practice of parallel programming (PPoPP '05), pages 142–152, June 2005.

[PQVR+01] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In Reinhard Wilhelm, editor, *Proceedings of the 10th International Conference on Compiler Construction (CC '01)*, volume 2027 of *LNCS: Lecture Notes in Computer Science*, pages 334–354, April 2001.

[PV04a] Christopher J. F. Pickett and Clark Verbrugge. Compiler analyses for improved return value prediction. Technical Report SABLE-TR-2004-6, Sable Research Group, School of Computer Science, McGill University, October 2004.

[PV04b] Christopher J. F. Pickett and Clark Verbrugge. Return value prediction in a Java virtual machine. In *VPW2: Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop*, pages 40–47, October 2004.

[PVK07] Christopher J. F. Pickett, Clark Verbrugge, and Allan Kielstra. libspmt: A library for speculative multithreading. Technical Report SABLE-TR-2007-1, Sable Research Group, School of Computer Science, McGill University, March 2007.

[PVK09] Christopher J. F. Pickett, Clark Verbrugge, and Allan Kielstra. Understanding method level speculation. Technical Report SABLE-TR-2009-2, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, September 2009.

[PW94] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 16(4):1248–1278, 1994.

[QMS+05] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure

for speculative threading based on pre-computation slices. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 269–279, June 2005.

[RCRH95] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, December 2001.

[RKM+10] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS'10: Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 65–76, March 2010.

[rLK98] Per Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *ISMM'98: Proceedings of the 1st International Symposium on Memory Management*, pages 176–185, October 1998.

[Rot99] Eric Rotenberg. *Trace Processors: Exploiting Hierarchy and Speculation*. PhD thesis, U. Wisconsin–Madison, August 1999.

[RP95] Lawrence Rauchwerger and David Padua. The LRPD test: Speculative runtime parallelization of loops with privatization and reduction parallelization. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation*, pages 218–232, 1995.

[RR95] Mukund Raghavachari and Anne Rogers. Understanding language support for irregular parallelism. In *PSLS'95: Proceedings of the International Work-*

shop Parallel Symbolic Languages and Systems, volume 1068 of *LNCS: Lecture Notes in Computer Science*, pages 174–189. Springer, October 1995.

[RS01] Peter Rundberg and Per Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *JILP: Journal of Instruction-Level Parallelism*, 3:1–28, October 2001.

[RS03] Peter Rundberg and Per Stenström. Speculative lock reordering: Optimistic out-of-order execution of critical sections. In *IPDPS'03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, pages 11–18, April 2003.

[RTL+05] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In *ICS'05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 179–188, June 2005.

[RVB07] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. Function level parallelism driven by data dependencies. *CAN – DASCMP'06: SIGARCH Computer Architecture News – Special Issue on the 2006 Workshop on Design, Analysis, and Simulation of Chip Multiprocessors*, 35(1):55–62, March 2007.

[RVRA08] Easwaran Raman, Neil Vachharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *CGO'08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 175–184, April 2008.

[RW02] Algis Rudys and Dan S. Wallach. Transactional rollback for language-based systems. In *Proceedings of the 3rd International Conference on Dependable Systems and Networks (DSN)*, pages 439–448, June 2002.

[SA00] Toshinori Sato and Itsujiro Arita. Table size reduction for data value predictors by exploiting narrow width values. In *ICS'00: Proceedings of the 14th International Conference on Supercomputing*, pages 196–205, May 2000.

[SAN06] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM'06: Proceedings of the 2006 International Symposium on Memory Management*, pages 84–94, June 2006.

[SB04] Nana B. Sam and Martin Burtscher. Exploiting type information in load-value predictors. In *VPW2: Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop*, pages 32–39, October 2004.

[SB05a] Nana B. Sam and Martin Burtscher. Complex load-value predictors: Why we need not bother. In *WDDD'05: Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, pages 16–24, June 2005.

[SB05b] Nana B. Sam and Martin Burtscher. Improving memory system performance with energy-efficient value speculation. *CAN: SIGARCH Computer Architecture News*, 33(4):121–127, September 2005.

[SB06] Jeremy Singer and Gavin Brown. Return value prediction meets information theory. In *QAPL'06: Proceedings of the 4th International Workshop on Quantitative Aspects of Programming Languages*, volume 164:3 of *ENTCS: Electronic Notes in Theoretical Computer Science*, pages 137–151, October 2006.

[Sch08] Georg Schätti. Hotspec - a speculative JVM. Master's thesis, Laboratory for Software Technology, Institute of Computer Systems, Department of Computer Science, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, January 2008.

[Sco06] Micheal L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN Workshop on Transactional Computing*, Jun 2006. Held in conjunction with PLDI 2006.

[SCZM00] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of*

the 27th Annual International Symposium on Computer Architecture (ISCA '00), pages 1–12, June 2000.

[SCZM02] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving value communication for thread-level speculation. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 65–75, 2002.

[SCZM05] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems (TOCS)*, 23(3):253–300, August 2005.

[SH86] Vivek Sarkar and John Hennessy. Partitioning parallel programs for macro-dataflow. In *LFP'86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 202–211, August 1986.

[SH04] John Seng and Greg Hamerly. Exploring perceptron-based register value prediction. In *Second Value-Prediction and Value-Based Optimization Workshop*, pages 10–16, Boston, MA, October 2004.

[SHSA01] Toshinori Sato, Akihiko Hamano, Kiichi Sugitani, and Itsujiro Arita. Influence of compiler optimizations on value prediction. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 312–321, 2001.

[SL98] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 283–294, October 1998.

[SMH98] Byoungro So, Sungdo Moon, and Mary W. Hall. Measuring the effectiveness of automatic parallelization in SUIF. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 212–219, 1998.

[SS97a] Yiannakis Sazeides and James E. Smith. Implementations of context-based value predictors. Technical Report TR ECE-97-8, University of Wisconsin–Madison, December 1997.

[SS97b] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–258, December 1997.

[SS06] Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, pages 416–425, 2006.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture*, volume 2: Patterns for Concurrent and Networked Objects. Wiley, 2000.

[Sta98] Standard Performance Evaluation Corporation. SPEC JVM Client98 benchmark suite, June 1998. `http://www.spec.org/jvm98/`.

[SYK+01] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. In *OOPSLA'01: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 180–195, October 2001.

[SYON08] Toshio Suganuma, Toshiaki Yasue, Tamiya Onodera, and Toshio Nakatani. Performance pitfalls in large-scale Java applications translated from COBOL. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 685–696, October 2008.

[SZ99] Nir Shavit and Asaph Zemach. Scalable concurrent priority queue algorithms. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 113–122, May 1999.

[TCC+00] Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, December 2000.

[TG95] Guy Tremblay and Guang R. Gao. The impact of laziness on parallelism and the limits of strictness analysis. In *Proceedings High Performance Functional Computing*, pages 119–133, 1995.

[THA+99] Jenn-Yaun Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, September 1999.

[THG09] Fuad Tabba, Andrew W. Hay, and James R. Goodman. Transactional value prediction. In *TRANSACT'09: Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, February 2009.

[Tja99] Stephen Jeremy Tjasink. *PLaVa: A Persistent, Lightweight Java Virtual Machine*. PhD thesis, Department of Computer Science, University of Cape Town, Rondebosch, South Africa, February 1999.

[TK04] Anita Thomas and David Kaeli. Value prediction with perceptrons. In *Second Value-Prediction and Value-Based Optimization Workshop*, pages 3–9, Boston, MA, October 2004.

[Tra88] Kenneth R. Traub. *Sequential implementation of lenient programming languages*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1988.

[TS99] Dean M. Tullsen and John S. Seng. Storageless value prediction using prior register values. In *ISCA'99: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 270–279, May 1999.

[TT05] Nathan Tuck and Dean M. Tullsen. Multithreaded value prediction. In *HPCA'05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 5–15, February 2005.

[vPCC07] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 79–89, March 2007.

[VR00] Raja Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Montréal, Québec, Canada, July 2000. `http://www.sable.mcgill.ca/soot/`.

[War06] Fredrik Warg. *Techniques to Reduce Thread-Level Speculation Overhead*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, May 2006.

[WF97] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *MICRO 30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 281–290, December 1997.

[Wil03] Barry Wilkins. *Meld: A Pattern Supported Methodology for Visualisation Design*. PhD thesis, School of Computer Science, University of Birmingham, Birmingham, West Midlands, England, March 2003.

[WJH05] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for Java. In *OOPSLA'05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 439–453, October 2005.

[WJH06] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Revocation techniques for Java concurrency. *CC:PE: Concurrency and Computation: Practice and Experience*, 18(12):1613–1656, October 2006.

[WK05] John Whaley and Christos Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *ICPP'05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 147–156, June 2005.

[Wol96] Michael J. Wolfe. *High Performance Compilers for Parallel Computing*. Pearson Eduction POD, 1st edition, January 1996.

[WS01] Fredrik Warg and Per Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *PACT'01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230, September 2001.

[XPV07] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for Java programs. In *PASTE'07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 75–82, June 2007.

[YMH98] Kazuki Yoshizoe, Takashi Matsumoto, and Kei Hiraki. Speculative parallel execution on JVM. In *Proceedings of the 1st UK Workshop on Java for High Performance Network Computing*, September 1998.

[YYX05] Xiao Yong, Yang Yanping, and Zhou Xingming. Revised stride data value predictor design. In *HPCASIA'05: Proceedings of the 8th International Conference on High-Performance Computing in Asia-Pacific Region*, pages 526–531, November 2005.

[ZC05] Huiyang Zhou and Thomas M. Conte. Enhancing memory-level parallelism via recovery-free value prediction. *TC: IEEE Transactions on Computers*, 54(7):897–912, July 2005.

[ZCSM02] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication. In *ASPLOS*, San Jose, CA, USA, October 2002.

[ZCSM04] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of memory-resident value communication between speculative threads. In *CGO'04: Proceedings of the 2004 International Symposium on Code Generation and Optimization*, page 39, March 2004.

[ZF02] Mohamed Zahran and Manoj Franklin. Return-address prediction in speculative multithreaded environments. In *HiPC'02: Proceedings of the 9th International Conference on High Performance Computing*, pages 609–619, December 2002.

[ZF03] Mohamed Zahran and Manoj Franklin. Dynamic thread resizing for speculative multithreaded processors. In *ICCD'03: Proceedings of the 21st International Conference on Computer Design*, pages 313–318, October 2003.

[ZFC03] Huiyang Zhou, Jill Flanagan, and Thomas M. Conte. Detecting global stride locality in value streams. In *ISCA'03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 324–335, June 2003.

[Zha05] Antonia Zhai. *Compiler optimization of value communication for thread-level speculation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, January 2005.

[ZKN07a] Lingli Zhang, Chandra Krintz, and Priya Nagpurkar. Language and virtual machine support for efficient fine-grained futures in Java. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 130–139, September 2007.

[ZKN07b] Lingli Zhang, Chandra Krintz, and Priya Nagpurkar. Supporting exception handling for futures in Java. In *PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, pages 175–184, September 2007.