Compiler and Runtime Techniques for Automatic Parallelization of Sequential Applications

by

Mojtaba Mehrara

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

Doctoral Committee:

Associate Professor Scott Mahlke, Chair
Professor Todd M. Austin
Associate Professor Robert Dick
Assistant Professor Satish Narayanasamy
Tim Harris, Microsoft Research

To my parents.

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Prof. Scott Mahlke, for his continuous guidance and support. His encouragement, enthusiasm, and energy have kept me going at many critical moments of frustration during graduate school. He is always ready to discuss ideas and come up with interesting ways to tackle research problems. This dissertation would not have been completed without his help, vision, and support.

I would also like to thank my dissertation committee, Prof. Todd Austin, Dr. Tim Harris, Prof. Robert Dick, and Prof. Satish Narayanasamy for their valuable feedback. I learned a lot from Todd about how to develop ideas and come up with ways to present them in writing and speech. Special gratitude goes to Tim for his enthusiasm and patience. I learned a lot from him during my internship at Microsoft Research. He has also been very kind to provide useful suggestions and feedback on all parts of this work, when we were submitting different pieces for publication. Ideas and presentations throughout the dissertation improved a lot as a result of incorporating his vision and suggestions. Prof. Robert Dick also provided many useful comments on this dissertation.

I should thank my colleagues, Jeff Hao, Po-chun Hsu, and Mehrzad Samadi for helping with infrastructures and experiments, and Ganesh Dasika, Shuguang Feng, Shantanu Gupta, and Amir Hormati for proof-reading and providing feedback on various parts of this dissertation. I

would also like to thank the Liberty Research group at Princeton University for providing us with the memory profiler used in Chapter 2 and the hardware used for experiments in Chapter 3.

I feel truly fortunate to be among my fellow comrades in the Compilers Creating Custom Processors (CCCP) group. You guys made my grad school experience much less boring, and in fact, quite fun and enjoyable. I'd like to thank all my lab-mates over the years: Ganesh Dasika, Amir Hormati, Shantanu Gupta, Shuguang Feng, Kevin Fan, Hyoun Kyo Cho, Hongtao Zhong, Kypros Constantinides, Andreas Moustakas, Mike Chu, Manjunath Kudlur, Nathan Clark, Po-Chun Hsu, Yongjun Park, Amin Ansari, Hyunchul Park, Jeff Hao, and Mark Woh.

Much gratitude goes to my friends outside office: Niloufar, Alireza, Mona, Amir, Danial, Mojtaba, Armin, Elnaz, Mehrzad, Parisa, and many others. I couldn't have made it through without you people. I'll never forget the fun times we spent together. I should also thank my friends from high school and college: Ali, Erfan, Reza, and Pedram. Thank you for always being there to talk, laugh, and discuss about very important and not-so-important things in life.

Finally and most importantly, I'd like to thank my family - my parents, Mohammad and Fatemeh, and my siblings, Mostafa, Zahra, and Tahereh. My parents have always provided me with unconditional love and support. Their encouragement and trust always makes me more confident about my decisions in life. My brother, Mostafa, has always been there for me as a good friend, ready to share stories, laugh, argue, and put up with my complaints about random things in school and life. Tahereh, my sister, has been a great friend over the years, always there to hang out, talk, and occasionally fight. Fights got much less brutal for some reason as we grew older, though, and somehow stopped eventually. I also thank my brother-in-law, Mostafa, for being such a good friend.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES viii

ABSTRACT xi

CHAPTER

1. Introduction 1

1.1 Static and Dynamic Parallelization: Challenges and Opportunities . . . 5
1.1.1 Static Speculative Parallelization 5
1.1.2 Dynamic Speculative Parallelization 6
1.2 Contributions 8

2. Parallelizing Sequential C/C++ Applications on Commodity Hardware using a Low-cost Software Transactional Memory . 11

2.1 Introduction 11
2.2 Motivation 14
2.2.1 Challenges in Software Transactional Memory Systems . . 14
2.2.2 Speculation Requirements for Loop Parallelization 16
2.3 STMlite 17
2.3.1 Overview 19
2.3.2 Transactional Loads and Stores 22
2.3.3 Transaction Commit Manager 22
2.3.4 Individual Transaction Commits 25
2.4 Loop Parallelization Using STMlite 27
2.4.1 Baseline Parallelization Framework 27
2.4.2 Interaction of Parallel Loops with STMlite 31
2.5 Results 32

2.5.1 STMlite on STAMP 33
2.5.2 STMlite on Parallelized Sequential Programs 36
2.5.3 Effects of static memory analysis and signature sizes 40
2.6 Related Work 42
2.7 Summary 45

3. Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism 47

3.1 Introduction 47
3.2 Dynamic Parallelization Challenges 52
3.3 Dynamic Analysis and Code Generation 55
3.3.1 Dynamic Parallel Loop Selection 57
3.3.2 Parallel Code Generation 61
3.4 Ultra-lightweight Speculation Support 63
3.4.1 Conflict Detection Using Reference Counting 64
3.4.2 Conflict Detection Using Range-based Checks 66
3.4.3 Instrumenting Array Functions 67
3.4.4 Checkpointing and Recovery 68
3.4.5 Checkpointing Optimizations 69
3.5 Evaluation 70
3.5.1 Experimental Setup 70
3.5.2 Parallelism Potential and Speculation Cost 71
3.5.3 Results 73
3.5.4 Discussion 75
3.6 Related Work 77
3.7 Summary 80

4. Dynamically Accelerating Client-side Web Applications through Decoupled Execution 81

4.1 Introduction 81
4.2 Background 83
4.3 ParaGuard: Concurrent Guard Execution 87
4.3.1 ParaGuard Generation 89
4.3.2 Recovering Interpreter State using Selective Snapshots . . . 95
4.4 Optimizations on ParaGuard 99
4.4.1 Guard Branch Aggregation 99
4.4.2 Profile-based Snapshot Elimination 101
4.5 Experimental Evaluation 102
4.5.1 Methodology 102
4.5.2 Results 104
4.6 Related Work 109
4.7 Summary 111

5. Summary and Conclusion 112

BIBLIOGRAPHY 115

LIST OF FIGURES

Figure

1.1 Performance of SPEC CPU benchmarks over the past two decades. The y-axis is in logarithmic scale. 2

1.2 Static parallelization framework. Analysis and code generation is done at compile time, while the runtime provides speculation support. 5

1.3 Dynamic parallelization framework. Analysis, code generation and speculation is done at runtime. 7

2.1 Single-threaded runtime breakdown of the Transactional Locking II STM system [35] on two STAMP transactional benchmarks. 15

2.2 STMlite execution model. Solid lines denote execution flow. Dashed lines denote passing messages by signals or memory polling. The dash-dot line shows an indirect write/read relation (each transaction writes to a precommit log entry which is later read by the TCM). 19

2.3 STMlite data structures. Each transaction has an individual header. The TCM has a single commit log and there is a precommit log for each execution core inside the TCM. 20

2.4 Pseudocode for transactional loads and stores. 21

2.5 Commit management in the TCM. 23

2.6 Overview of the parallelization framework (CS: chunk size, IS: iteration start, IE: iteration end, SS: step size, TC: thread count). 28

2.7 STMlite performance on STAMP benchmarks. The vertical axis shows the speedup compared to the sequential execution and horizontal axis is the number of cores. The number of cores in STMlite includes one core that is used for the TCM. 34

2.8 Small transactions in `Kmeans` working on global data and their equivalent lock-based implementation. 35

2.9 Profiled DOALL, provable DOALL and selected parallel loop coverage. The vertical axis shows fraction of sequential execution. 36

2.10 STMlite performance on automatically parallelized kernel benchmarks. The vertical axis shows the speedup compared to the sequential execution and horizontal axis is the number of cores. The number of cores in STMlite includes one core that is used for the TCM. 38

2.11 STMlite performance on automatically parallelized SPECfp benchmarks. The vertical axis shows the speed up compared to the sequential execution and horizontal axis is the number of cores. The number of cores in STMlite includes one core that is used for the TCM. 39
2.12 Effect of using static pointer analysis on speedup for `052.alvinn`. 41
2.13 Effect of varying signature size on speedup for `RLS` 42
3.1 Performance of JavaScript, compared to C, C++ and Java. 48
3.2 Static parallelization flow. 52
3.3 ParaScript dynamic parallelization flow. 55
3.4 Examples of dependence instances in various variable categories. Solid and dashed arrows depict control flow and data flow edges respectively. Question marks on dashed arrows show possible data flow dependences. Check-marks show operations that are safe for parallelization, while cross-marked operations are the ones that hinder parallelisms. Both of these operation categories are identified by the simple JIT-time data flow analysis with DU chains. Exclamation marks are instances of possible cross iteration dependences that could not be figured out at JIT-time. Final safety decisions on these are deferred to further runtime analysis and speculation (Section 3.4). 58
3.5 Dynamic code generation template (CS: chunk size, IS: iteration start, IE: iteration end, SS: step size, TC: thread count). 62
3.6 Extra steps needed at runtime for speculation support for loop parallelization in a traditional STM (TL2 [35]) and in ParaScript. 64
3.7 Parallelization coverage as ratio of parallelizable code runtime to total sequential program execution. 72
3.8 Speculation and checkpointing overhead as a fraction of total sequential program execution. 73
3.9 ParaScript performance improvement over sequential for 2, 4 and 8 threads. Black lines on top of the graphs show the performance of a system with hardware support for speculation (e.g., HTM). 74
4.1 Fraction of total instructions devoted to computing guards across four groups of benchmarks: SunSpider, V8, Pixastic image processing applications, and a set of JavaScript games. These bars include guards and portion of the backward slice only needed by guards and not used elsewhere. 82
4.2 JavaScript tracing and type specialization in TraceMonkey. This state machine describes how the trace monitor manages trace-based just-in-time compilation. 85
4.3 Breakdown of different types of guards in SunSpider, V8, Pixastic image processing and JavaScript games. 88
4.4 Offloading guard execution to the ParaGuard thread. 89
4.5 Sample JavaScript source code. 92
4.6 Original TraceMonkey's Low-level IR for the source code in Figure 4.5. Instructions marked with (*) are to be copied and the ones with (+) are to be moved to the ParaGuard trace. 93
4.7 Main Trace LIR after guard promotion. 94

4.8 ParaGuard trace LIR after guard promotion. 95
4.9 Extra code added after PG7 in Figure 4.8. label0 is inserted right before instruction 24’. updateMin and updateMax code segments are inserted after the label2 code segment. 97
4.10 ParaGuard Trace LIR after applying guard branch aggregation. GA1 is the final and only guard check in the ParaGuard trace. 100
4.11 Ratio of promoted guards to total number of guards. 104
4.12 Number of triggered guards in ParaGuard during every 100,000 instructions in the guard promotion technique without applying the profile-based snapshot elimination. The y-axis is in logarithmic scale. 105
4.13 ParaGuard speedup on 2 processors compared to the baseline tracing. The left bars show the speedup after guard promotion and the right bars show the speedup after applying the profile-based snapshot elimination optimization. . 107
4.14 Utilization of the ParaGuard thread relative to the main thread before and after applying guard branch aggregation optimization. 108

ABSTRACT

Compiler and Runtime Techniques for Automatic Parallelization of Sequential Applications

by

Mojtaba Mehrara

Chair: Scott Mahlke

Multicore designs have emerged as the mainstream design paradigm for the microprocessor industry. Unfortunately, providing multiple cores does not directly translate into performance for most applications. An attractive approach for exploiting multiple cores is to rely on tools, both compilers and runtime optimizers, to automatically extract threads from sequential applications. This dissertation tackles many challenges faced in automatic parallelization of sequential applications, including general-purpose applications written in C/C++ and client-side web applications written in JavaScript, with the goal of achieving speedup on commodity multicore systems. First, a complete parallelizing compiler system for C/C++ is introduced. This system successfully identifies parallelization opportunities in programs and transforms the code to a parallel version. A matching runtime system, STMlite, is proposed to monitor the parallelized program behavior and fix any misspeculations that might happen. We show that this system can generate and execute parallel programs that are upto 2.2x faster than their sequential counterparts, when executed on an 8-core commodity system.

The second piece of work focuses on a similar problem in a very different application domain, JavaScript programs running on the client's web browser. This dissertation is the first research work that proposes dynamic and automatic parallelization of JavaScript applications. The nature of the JavaScript language and its target execution environments impose a completely different set of challenges that we intend to solve. We first propose the ParaScript parallelizing engine, which identifies and speculatively parallelizes potentially parallel code segments while the code is running in the browser. A low-cost and highly customized speculation approach verifies the execution of the parallelized client-side code and rolls back in case of any misspeculation. Dynamic parallelization using ParaScript yields an average of 2.18x speedup over sequential JavaScript code on an 8-core commodity system. In addition, we introduce ParaGuard, a technique which executes the runtime checks required by trace-based JavaScript compilers in parallel with the main execution. ParaGuard is able to improve performance by 15% by using an additional core in multi-core systems.

CHAPTER 1

Introduction

For more than four decades, the semiconductor industry has depended on Moore's law to deliver consistent application performance gains through the multiplicative effects of increased transistor counts and higher clock frequencies. However, power dissipation and thermal constraints have emerged as dominant design issues and forced architects away from relying on increasing clock frequency to improve performance. Exponential growth in transistor counts still remains intact and a powerful tool to improve performance, though the paradigm through which performance is perceived has shifted. Performance is now based on throughput, utilizing multiple cores performing computation in parallel to complete a larger volume of work in a shorter period of time. These multicore systems have become the industry standard from high-end servers, down through desktops, gaming, and even mobile platforms. However, one of the most difficult challenges going forward is software: if the number of devices per chip continues to grow with Moore's law, can the available hardware resources be converted into meaningful application performance gains? Multiple cores readily help where threads are plentiful, such as web servers. However, they provide little or no gains for sequential applications. In fact, the performance of sequential applications may suffer due to the use of simpler cores.

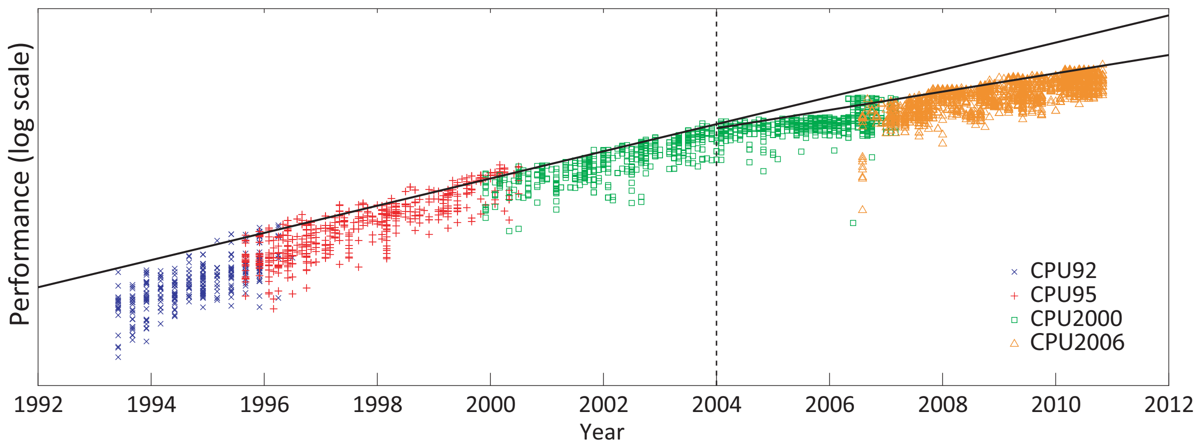


Figure 1.1: Performance of SPEC CPU benchmarks over the past two decades. The y-axis is in logarithmic scale.

This effect is clearly illustrated in Figure 1.1, which shows the performance of sequential SPEC CPU benchmarks over the past two decades. Up until 2004, sequential applications were enjoying exponential performance improvements as a result of various architectural techniques, larger caches, and scaling in the processor's operating frequency. However, with the industry-wide shift to multicore systems, this exponential growth slowed down mainly due to the adoption of simpler architectures, smaller caches per core, and lower frequencies.

One way to alleviate this performance deficiency is to develop explicitly parallel programs to make use of the additional resources in multicore processors. Many new languages have been proposed to ease the burden of writing these programs, including Atomos [27], Cilk [39], and StreamIt [85]. Despite these and many other languages, the effort involved in creating correct and efficient parallel programs is still far more substantial than writing the equivalent sequential version. Developers must be trained to program and debug their applications with the additional concerns of deadlock, livelock, and race conditions. Converting an existing sequential application is often more challenging, as it may not have been developed to be easily parallelized in the first place. The lack of necessary compiler technology is increasingly apparent as the push to run general-purpose software on multicore platforms is required [59].

An attractive approach for exploiting multiple cores is to rely on tools, both compilers and runtime systems, to automatically extract threads from sequential applications. However, despite decades of research on automatic parallelization, most techniques are only effective in the scientific and data parallel domains where array dominated codes can be precisely analyzed by the compiler. Thread-level speculation offers the opportunity to expand parallelization to general-purpose programs, but at the cost of expensive hardware support. In addition, these techniques provide means for the automatic parallelization of statically typed general-purpose languages such as C, C++ and Java, without providing solutions for the emerging and increasingly important web applications.

As the web becomes the platform of choice for the execution of more complex applications, a growing portion of computation is handed off by developers to the client side to reduce network traffic and improve application responsiveness. Therefore, the client-side component, often written in JavaScript, is becoming larger and more compute-intensive, increasing the demand for high performance JavaScript execution.

Historically, achieving high performance has been a difficult challenge for dynamically typed languages such as JavaScript. The types of variables and expressions may vary at runtime, thus the compiler must emit generic code that can handle all potential type combinations. The most straight forward technique for executing such a generic code is through interpretation, which is extremely slow in comparison to executing binaries generated for statically typed languages such as C or C++. There have been many recent and on-going efforts to tackle this performance bottleneck by web browser developers, with a great focus on improving the sequential performance of JavaScript programs [3, 40]. They mostly try to accelerate these programs by compiling them down to the binary code, while adding some flexibility to fix the execution flow in case types change dynamically. However, with the industry-wide move to multicore platforms, these techniques provide no solutions to scale the performance. On the other hand, due to the lack of concurrency support in the language and runtime systems, JavaScript applications have been written as sequential programs by default. Therefore, there is no immediate way of exploiting available hardware resources in multicores to improve performance. Straight forward application of static parallelization techniques is ineffective for a number of reasons, including the inability to perform whole program analysis, and expensive profiling and memory dependence analysis.

In this dissertation, the above challenges in exploiting parallelism are addressed by introducing novel techniques for automatically extracting parallelism from sequential, but *implicitly parallel*, applications. Such applications are written and debugged using a sequential language, such as C, C++ or JavaScript. However, they contain implicit parallelism that can be identified and exploited using advanced compilation and runtime mechanisms. We focus on automatic parallelization on commodity multicore systems in two very different domains, general-purpose applications written in C/C++ and client-side web applications written in JavaScript. In the following sections, we first look into the static versus dynamic parallelization paradigm, and then we detail our contributions in this work.

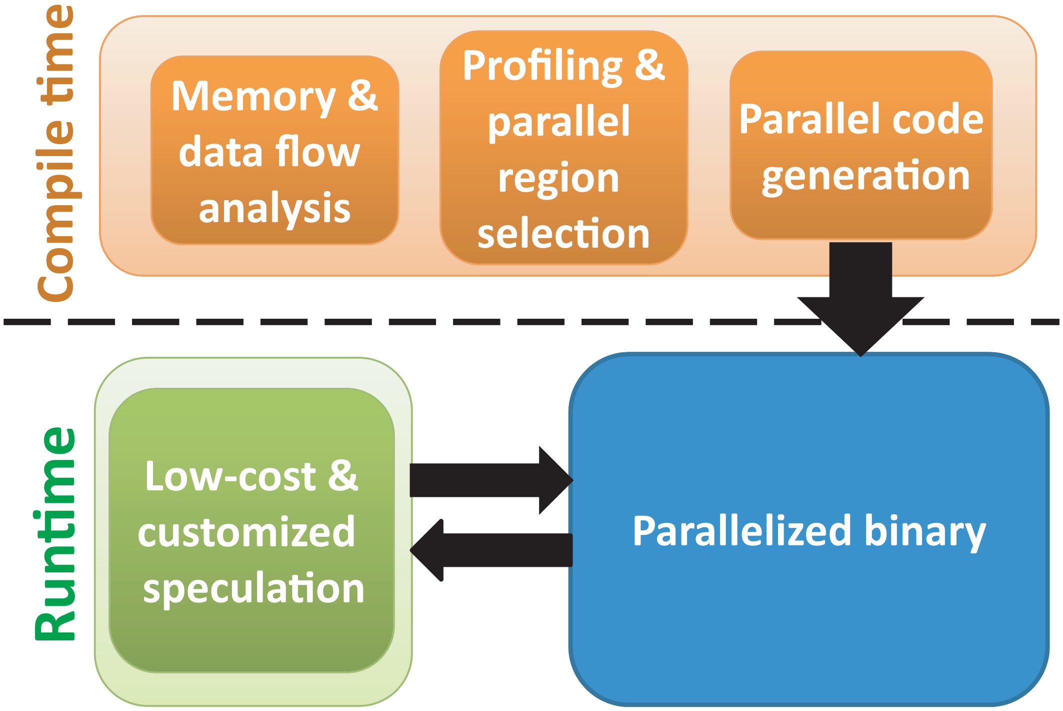

Compile time
Memory & data flow analysis
Profiling & parallel region selection
Parallel code generation
Runtime
Low-cost & customized speculation
Parallelized binary

Figure 1.2: Static parallelization framework. Analysis and code generation is done at compile time, while the runtime provides speculation support.

1.1 Static and Dynamic Parallelization: Challenges and Opportunities

1.1.1 Static Speculative Parallelization

There is a long history of static techniques for automatic and speculative parallelization of scientific and general purpose applications. As shown in Figure 1.2, these techniques usually make use of a combination of memory dependence analysis (pointer alias analysis), memory access profiling, execution graph profiling, and data flow analysis to identify potential loops for parallelization and generate parallel binary code using different speculative frameworks such as speculative DOALL [57, 93] and DSWP [87]. This binary is later executed along with a runtime speculation engine to monitor execution and roll-back in case of any misspeculations.

There have been various proposals for providing speculation support to programs both at the hardware and software level. In hardware, many research works introduce techniques for speculation at the cost of complicated and expensive hardware support [29, 46, 64]. However, with the exception of Sun's Rock processor [31], which had very limited hardware speculation support, none of these works have made their way into real products yet, mainly due to the high implementation costs and complex verification challenges.

Providing speculation support in software in the form of software transactional memory systems, has also been a very popular research topic. However, obviating the need for extra hardware support comes at a considerable performance cost. Many previous works [18, 35, 49] tackle various performance and correctness issues in software transactional memory models. Despite being effective in some applications, the kind of per-access logging and expensive software checks makes them unusable in many scenarios [28], especially sequential program parallelization where the performance improvements might get completely offset by the software speculation overhead. Therefore, we look into lowering the cost of software speculation by customizing it to the loop level parallelization target. As we describe in Chapter 2, by removing a number of requirements associated with conventional software transactional memories, and utilizing software bloom-filter based signatures to track memory accesses, we are able to achieve speedup by static parallelization of C/C++ applications on commodity multi-core hardware.

1.1.2 Dynamic Speculative Parallelization

While static parallelization techniques prove effective for languages such as C/C++ and Java, they fail to operate efficiently for highly dynamic environments like that of JavaScript, where the source code is shipped to the client right before execution and all the efforts of inter-

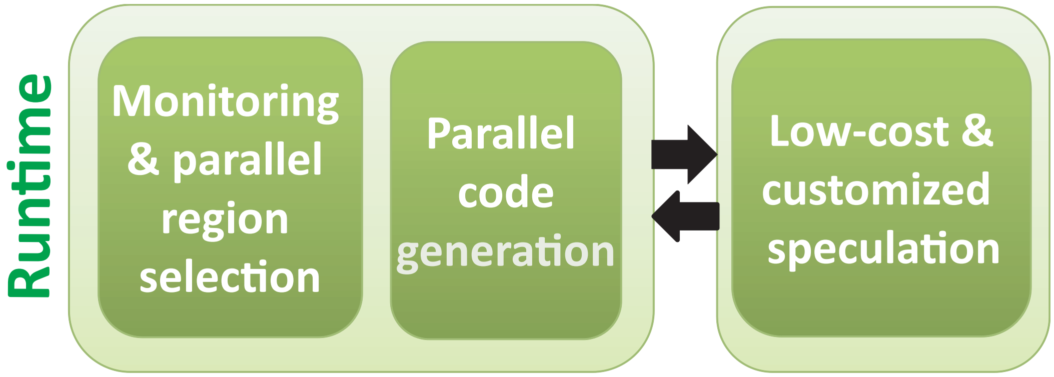

Runtime
Monitoring & parallel region selection
Parallel code generation
Low-cost & customized speculation

Figure 1.3: Dynamic parallelization framework. Analysis, code generation and speculation is done at runtime.

pretation, just-in-time compilation, and execution is performed on the client's web browser. In such a setting, as shown in Figure 1.3, all steps of the parallelization process including analysis, parallel region selection, and parallel code generation needs to be done at runtime, in addition to the runtime speculation. Therefore, many of the techniques used for static parallelization would be too expensive to be applied at runtime. For instance, authors in [42] recently proposed a static JavaScript pointer analysis for security purposes, which takes about 3 seconds to complete on a set of JavaScript programs with an average size of 200 lines of code. Performing this analysis at runtime is quite prohibitive, even for programs of that size. This overhead would be higher for larger applications that consist of hundreds of lines of code [73]. In addition to compiler analysis, the overhead of performing memory access profiling at runtime is also unacceptable. The work in [56] reported 100x to 500x increase in the execution time as a result of performing static memory profiling on C/C++ applications.

Therefore, with the exception of simple data flow analysis and code generation (both of which are already done at runtime in JavaScript engines), a dynamic parallelization framework cannot afford to utilize any of the static compile time steps in Figure 1.2. The lack of static information makes the use of C/C++ customized software speculation engines, as proposed in

Chapter 2 and also in [57, 69, 86], very expensive. As we show in Section 2.5, this overhead could make the parallelized version of the code up to 4x slower than the original sequential application.

Having these constraints in mind, we introduce low-cost and highly customized analysis and speculation techniques in Chapters 3 and 4 to alleviate most of the overheads associated with static parallelization, and achieve speedup for JavaScript applications on off-the-shelf multicore hardware.

1.2 Contributions

In this dissertation, with our goal of parallelization on commodity hardware, we look into reducing compilation and runtime overhead for both C/C++ and JavaScript platform using two main insights. First, instead of fine-grain runtime checking for conflict detection, we perform coarse-grain checks using software bloom filter signatures in Chapter 2, range-based and reference counting-based checks in Chapter 3, and decoupled parallel checks in Chapter 4. This gives us the opportunity to significantly reduce the checking overhead, which has been done on a fine-grain per-access fashion in prior art, and enable realization of automatic parallelization techniques on off-the-shelf multicore hardware. Second, by taking less risk and perform more pessimistic speculation, we are able to reduce a large portion of checking overhead in Chapters 2 and 3 and snapshot taking overhead in Chapter 4. In particular, the contributions offered in this dissertation are as follows:

- First, a complete parallelizing compiler system for C/C++ is presented. This system identifies parallelization opportunities in sequential programs and generates their parallel counterpart. In order to ensure the correctness of the parallelized code, a runtime speculation system, called STMlite, is proposed. STMlite [57] is a low-cost software transactional memory system which monitors and validates parallelized execution at runtime and rolls back execution in case of any misspeculations. STMlite eliminates a considerable amount of checking and locking overhead in conventional software transactional memory models by decoupling the commit phase from the main transaction execution, and using software-based bloom filter signatures to detect conflicts between transactions. By utilizing these signatures, the overhead of individual memory accesses within each speculative region is minimized. The centralized commit process enables both effective cross checking of signatures and in-order speculative region commits with minimal overhead.

- The second part of this dissertation focuses on extracting parallelism from JavaScript web applications. In order to exploit hardware concurrency while retaining the traditional sequential programming model, we propose ParaScript [58], an automatic runtime parallelization system for JavaScript applications. First, we introduce a runtime scheme for identifying parallelizable regions, generating the parallel code on-the-fly, and speculatively executing it. Second, we propose an ultra-lightweight software speculation mechanism to manage parallel execution. This speculation engine consists of a selective checkpointing scheme and a novel runtime dependence detection mechanism based on reference counting and range-based array conflict detection. Our system is able to achieve speedup over the Firefox web browser using multiple threads on commodity multi-core systems, while performing all required analyses and conflict detection dynamically at runtime.

- The final part of this dissertation, ParaGuard, exploits the extra processing power in multicore systems to further improve the performance of trace-based JavaScript executions. Trace-based just-in-time compilation has been proposed to address the sequential performance bottleneck in JavaScript applications. In trace-based engines, a considerable portion of execution time is spent on running *guards* which are operations inserted in the native code to check if the properties assumed by the compiled code actually hold during execution. We introduce ParaGuard [60] to off-load these guards along with their backward slices to another thread, while speculatively executing the main trace, and thereby, take the checking overhead off the execution's critical path. In a manner similar to what happens in current trace-based JITs, if a check fails, ParaGuard aborts the native trace execution and reverts back to interpreting the JavaScript bytecode. We also propose several optimizations including guard branch aggregation and profile-based snapshot elimination to further improve the performance of our technique.

The rest of this dissertation is organized as follows. Chapter 2 introduces the C/C++ parallelizing system and the *STMlite* software transactional memory system. The *ParaScript* system for automatically and speculatively parallelizing JavaScript execution is described in Chapter 3. Parallelizing runtime checks with the main execution in *ParaGuard* is presented in Chapter 4. Finally, Chapter 5 provides a summary and concludes the dissertation.

CHAPTER 2

Parallelizing Sequential C/C++ Applications on Commodity Hardware using a Low-cost Software Transactional Memory

2.1 Introduction

As the scaling of clock frequency and complexity of uniprocessors has reached physical limitations, the industry has turned to multicore designs. However, having multiple cores does not directly translate into performance for most applications. The industry has already fallen short of the decades-old performance trend of doubling performance every 18 months. While explicit parallel programming is one potential solution to the problem, it is not a panacea. These systems may burden the programmer with implementation details and can severely restrict productivity and creativity. In particular, getting performance for a parallel application on a heterogeneous hardware platform, such as the Cell architecture, often requires substantial tuning, a deep knowledge of the underlying hardware, and the use of special libraries. Further, there is a large body of legacy sequential code that cannot be parallelized at the source level.

Techniques for parallelizing Fortran programs [20, 25, 34, 44] usually target counted loops that manipulate array accesses with affine indices, where memory dependence analysis can be precisely performed. Unfortunately, these techniques do not often translate well to C and C++ applications. These applications, including those in the scientific and media processing domains, are much more difficult for compilers to analyze due to the extensive use of pointers, recursive data structures, and dynamic memory allocation. More sophisticated memory dependence analysis, such as points-to analysis [83], can help, but parallelization often fails due to unresolvable memory accesses.

Thread-level speculation (TLS) offers an opportunity for parallelizing C and C++ applications. With TLS, the architecture allows optimistic execution of code regions before all values are known [23, 45, 51, 65, 80, 84, 94]. Hardware and/or software structures track memory accesses to determine if any dependence violations occur. In such cases, register and memory state are rolled back to a previous correct state and sequential re-execution is initiated. With TLS, the programmer or compiler can delineate regions of code believed (but not provably) to be independent and amenable to parallelization [32, 37, 53, 55]. Profile data is often utilized during this process. The POSH compiler is an excellent example where TLS yielded approximately 1.3x speedup for a 4-way CMP on SPECint2000 benchmarks [53]. More recent work has shown that additional loop-level parallelism is covered up by a small number of register and control dependences, but can be unlocked with several dependence breaking transformations [93]. Outer-loop pipeline parallelism has also been identified as a key parallelization opportunity. Bridges *et al.* report a geometric mean of 5.5x gain on SPECint2000 (with variable number of threads up to 32) using decoupled software pipelining [26].

Proponents of TLS advocate hardware support for speculation generally in the form of transactional memory or similar techniques [45, 84]. Bulk tracking of memory dependences using signatures along with dedicated structures for managing speculative state provide an efficient environment for TLS [29]. However, the cost and complexity of implementing hardware or hybrid Rock processor [31], hardware support for TLS has not made it into mainstream multicore systems yet.

Alternatively, software TMs, or STMs, offer the opportunity for TLS support without any dedicated hardware. The first STM by Shavit *et al.* maintained read and write access locations in order to roll back in case of a transaction abort [77]. Many other works [35, 47, 50, 54, 74] proposed different forms of STM to tackle various performance and correctness issues involved in these systems. However, these STM implementations are far too expensive in terms of runtime overhead. For parallel applications, STMs typically result in visible slowdowns of 2x or more. The problem is even worse for compiler parallelized sequential applications where all the gains and more are typically wiped out by the STM.

STMs generally focus on flexibility to support a wide variety of transactions and scalability to enable many concurrent threads. STM control is fully distributed to the running threads. In this chapter, we take the opposite approach by introducing *STMlite*, a lean and efficient STM specifically customized for automatic parallelization. With our focus on compiler parallelization, the goal is managing a modest number of speculative threads (2-8) that a compiler can realistically expect to find in C and C++ applications. Further, we focus on tightly integrating the STM with the compiler parallelization framework to ensure low overhead. Some requirements of more generic STMs such as strong atomicity [78] and special handling of local variables are not needed in this setting. Locks are removed by centralizing the TM bookkeeping on a single, perhaps idle, core. In this manner, bookkeeping tasks occur in parallel with transaction execution and the overhead on each work thread is minimized. Most importantly, centralized control obviates the need for locks and their associated overhead. The obvious downside of centralized control is the lack of scalability, but for a modest number of threads, large increases in efficiency are possible for both parallelized and multithreaded applications.

This chapter is organized as follows. In Section 2.2, we discuss challenges in STM systems and customization opportunities based on our main goal – exploiting loop-level parallelism. Section 2.3 describes STMlite, our proposed STM model. We discuss our parallelization framework and the interaction between the compiler-generated code and STMlite in Section 2.4. In Section 2.5, we present our experimental results. Finally, Section 2.6 discusses related work and Section 2.7 provides a summary of the work presented in this chapter.

2.2 Motivation

2.2.1 Challenges in Software Transactional Memory Systems

STMs have the advantage of requiring no additional hardware. However, since it is implemented entirely in software, it entails a large runtime overhead in maintaining transactional state. The high overheads of an STM are due to several reasons. The largest bottleneck in STMs is the maintenance and validation of read sets in read-write transactions. These sets keep track of every address read by a transaction, and are used to maintain coherence between transactions. For each load, the STM has to execute at least one transactional load and revalidate its timestamp when the transaction commits. As transactions read larger amounts of data, this overhead

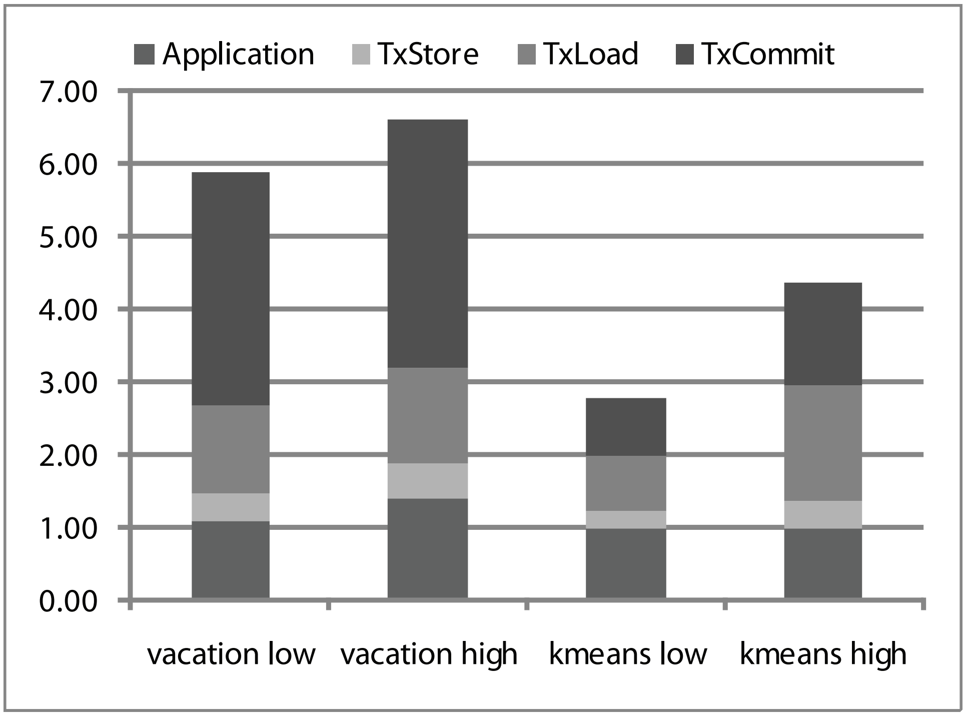


Figure 2.1: Single-threaded runtime breakdown of the Transactional Locking II STM system [35] on two STAMP transactional benchmarks.

becomes substantial.

Secondly, global locks are necessary for transactions to write back their final "correct" data. During a transactional store, the address and value are stored into a write set, deferring any change in memory until commit. This allows transactions to remain coherent with each other, but adds a considerable overhead during commit time for obtaining the locks on these addresses and writing them back to their final location. The use of locks in the data write back is expensive as it involves atomic instructions.

In order to get a better understanding of what the major sources of overhead are in an advanced STM system, we performed an experiment on two STAMP benchmarks [62] using a state-of-the-art STM system - Sun's Transactional Locking 2 (TL2) [36]. We measured the time spent in each TM component of a single threaded transactional execution of these benchmarks using the TL2 library. A similar analysis has also been done in [63]. Figure 2.1 shows the result of this experiment. The vertical is the execution time normalized to the sequential runtime. Vertical bars show the fractions of runtime spent in the main application, transactional commits (TxCommit), transactional stores (TxStore), and transactional loads (TxLoad).

The chart clearly shows the large overhead of read set maintenance in the `Vacation` benchmark, which has large transactions with many transactional reads. Keeping track of the read set causes considerable overhead, as depicted by the TxLoad portion of each bar. Additionally, the checks required during commit to maintain read set coherence are extremely costly [81], representing over half the runtime in `Vacation` with high contention. For the `Kmeans` benchmark, the overheads are not as severe because its read sets are smaller, but it still exhibits similar behavior.

2.2.2 Speculation Requirements for Loop Parallelization

There are several aspects of STM models that are crucial for correctness in general. However, we can loosen some of these limitations and requirements in the loop parallelization domain to make the software-based speculation more efficient.

1. One of the shortcomings in STM models is the lack of strong atomicity guarantees, which raises correctness issues in parallel programs. Previous works [17, 78, 75] have addressed the issue of strong atomicity in STMs. While being effective, these approaches impose a non-trivial amount of complexity or performance overhead on the system. However, using STM for speculation in loop parallelization obviates the need for strong atomicity, because the execution consists of at most a single in-flight parallel loop at each point. Since all the code in the loop is running inside transactions, there can be no non-transactional

code running at the same time as transactional code.

2. Special handling of local variables in a STM is not required for loop parallelization, because the loop iterations are not supposed to share any local variables on stack. Otherwise, they cause unresolvable cross iteration dependences, which prevent loop parallelization to begin with. Therefore, there is no need to have specialized transactional loads and stores for local variables.

3. Zombie transactions are transactions that have read a stale value or pointer from memory and have taken an incorrect code path which might lead to an infinite loop. One of the main sources of zombie transactions are loops with complicated linked-list operations. These loops are generally not parallelizable and therefore, we do not need to provide efficient and complicated ways for handling zombies in a STM for loop parallelization. However, to ensure correctness in other cases, we provide a mechanism for handling zombies in later sections that does not affect normal execution of transactions.

With these challenges in mind, we aim to tackle the two main sources of STM overhead: read-set maintenance and lock-based writeback mechanism. In addition, based on the specific speculation requirements in loop-level parallelism, we make simplifications to STMlite that makes it even more efficient.

2.3 STMlite

In this section, we describe our proposed STM model, STMlite. As was mentioned in Section 2.2, in traditional STM models, a considerable part of the execution time is spent in maintaining auxiliary data structures needed for providing correctness guarantees. In particular, one of the major bottlenecks is construction, maintenance, and frequent checking of read logs. The read log structure keeps track of the addresses (or objects in object-based implementations) read by each transaction. At transaction commit, each address in these logs is checked for consistency. In addition, although the programmer does not have to deal with the subtleties of lock-based programming, thanks to the usage of atomic blocks and TM primitives, the performance of the underlying runtime system still suffers from the downsides of using locks in many implementations. In order to address these problems, and as a step towards stylized customization for speculation used in loop-level parallelization, we developed a new software-based model that eliminates the need for read log maintenance during transaction execution and explicit locking during memory writebacks.

We assign a dedicated software thread for managing the execution of the transactions involved in the main computation. This thread, which runs on an individual core, is referred to as the Transaction Commit Manager (TCM). Having a central commit manager provides an environment in which the manager is responsible for ensuring that, at any given time, at most one transaction is writing to a particular memory location. With higher numbers of transactions, there can be several coordinating TCMs with each TCM managing a group of execution transactions (TCM virtualization). In this way, we can avoid having a single point of serialization in highly parallel applications.

The STMlite model essentially consists of several execution cores for running individual transactions and a TCM core for maintaining transactional consistency in the system. In the following subsections, we explain in more detail how each step works.

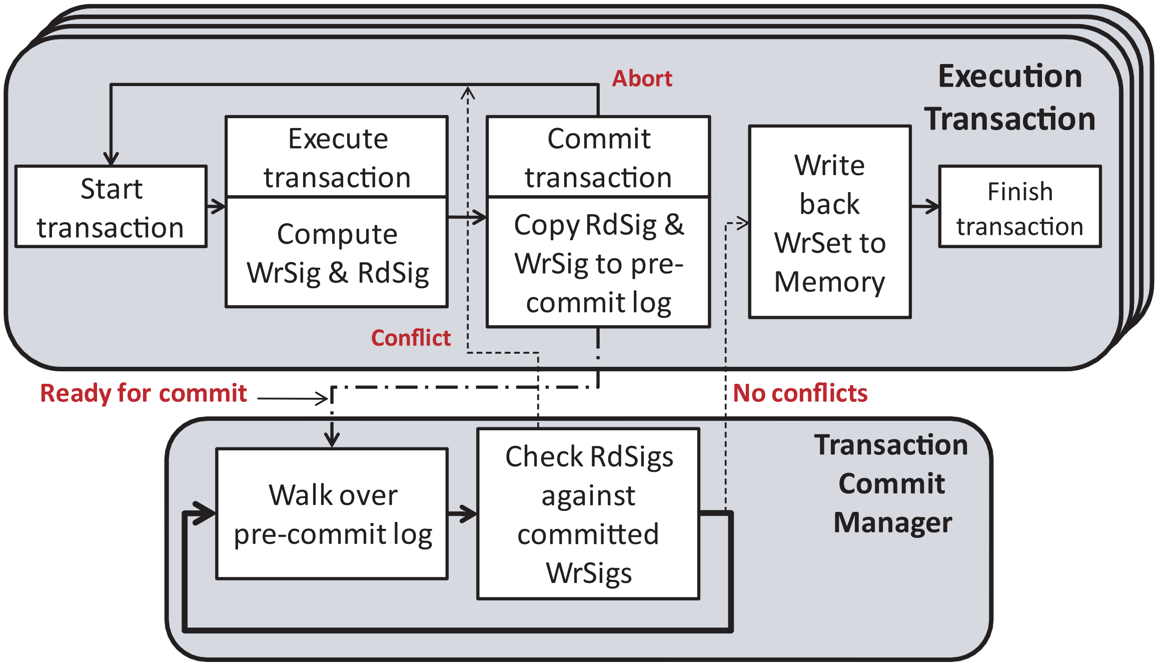


Figure 2.2: STMlite execution model. Solid lines denote execution flow. Dashed lines denote passing messages by signals or memory polling. The dash-dot line shows an indirect write/read relation (each transaction writes to a precommit log entry which is later read by the TCM).

2.3.1 Overview

Figure 2.2 summarizes the operation of STMlite. The top rectangle shows the execution flow inside each transaction. The bottom part is a summary of what happens inside the TCM.

Centralized management of individual transactions is made possible by using transactional read and write signatures, which are essentially hash-based representations of all reads and writes performed during execution. Using signatures in hardware was first proposed in [29]. However, unlike hardware, hash-based computations can become quite expensive in software. Therefore, choosing the right set of hash functions and the proper size for signatures is crucial in software systems to ensure minimal overhead and few false positives at the same time. In [49], hashing schemes are used to remove duplicates in the read-log and undo-logs of the "same" transaction. However, in order to use signatures for conflict detection between different trans-

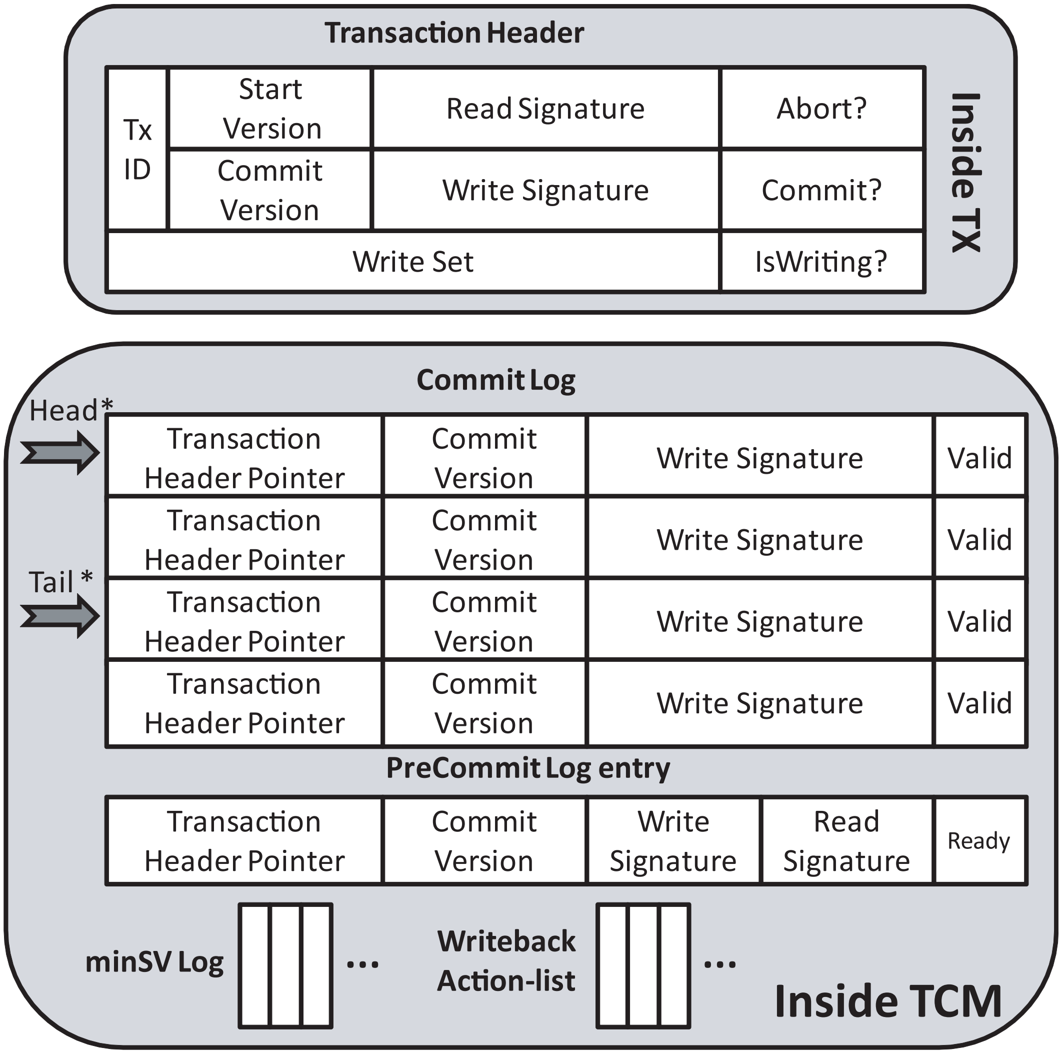


Figure 2.3: STMlite data structures. Each transaction has an individual header. The TCM has a single commit log and there is a precommit log for each execution core inside the TCM.

actions, a central manager is needed to check signatures against each other. In signature-based HTMs [29, 92], this is done in the coherence protocol. Here, it is done by the TCM.

Each transaction maintains a transaction header, which is shown in Figure 2.3. The transaction header contains some information gathered during transaction execution and is used during the commit process. The main idea behind STMlite is that all transactions compute read and write signatures during their execution. At commit, they copy these signatures to a list called the precommit log (Figure 2.3). This log is basically a single-reader/single-writer buffer that is

```
TxLoad(Addr){
  if SignatureFind(Addr, Self->wrSig)
    Load the correct value from the wrSet
  else
    Load from memory
    SignatureInsert(Addr, Self->rdSig)
}
TxStore(Addr, Data){
  Store Data to the WriteSet
  SignatureInsert(Addr, Self->wrSig)
 }
```

Figure 2.4: Pseudocode for transactional loads and stores.

read by the TCM and written by transactions. Its operation is inspired by the reservation station in traditional out-of-order processors. Committed transactions reside in another data structure called commit log (Figure 2.3). The commit log is only updated and read by the TCM.

The TCM goes through precommit log entries and checks whether their read signatures have conflicts with the write signatures of overlapping already-committed transactions in the commit log. If there is no hash collision, the transaction is notified to start writing back its write set. Otherwise, the transaction aborts and restarts its execution. During the write back process, the TCM is responsible for preventing concurrent writes to the same addresses in memory. TCM operation is detailed in Section 2.3.3.

In order to keep track of the relative start and commit times of transactions, we use a global clock mechanism similar to [35]. The TCM increments the global clock value whenever a writing transaction commits. We define the start version for each transaction as the value of the global clock at transaction start. Likewise, the commit version is the value of the global clock at commit time.

2.3.2 Transactional Loads and Stores

Figure 2.4 shows the pseudocode for STMlite's transactional load and store functions. `TxLoad` first checks the transaction's write signature (`wrSig`) to see if this transaction has previously written to `Addr`. If so, it reads the data from the write set (`wrSet`) and returns. In order to avoid walking through the entire write set when the number of store-to-load forwarding instances is high, we added a hash map to each transaction that caches the latest stored addresses and values for quick retrieval. Therefore, if `Addr` is found in the write signature, this hash table is checked before walking through the write set. This helps to lower transactional load overhead in many cases. If the transaction hasn't written to `Addr`, data is loaded from memory and `Addr` is inserted into the read signature (`rdSig`). `TxStore` stores `Data` to the write set and inserts `Addr` to the write signature.

As can be seen, the only major extra overhead in transactional loads and stores is due to the signature insert and find operations, though they remain low-cost for moderately sized signatures. Furthermore, the signature operations can be inserted in a decomposed fashion, separate form transactional loads and stores, enabling more aggressive compiler optimizations such as hoisting the signature calculations out of the loops with the aid of pointer alias analysis.

2.3.3 Transaction Commit Manager

As mentioned before, the TCM has two main data structures: the precommit log and the commit log (Figure 2.3). The commit log keeps track of committed transactions, and the precommit log contains transactions waiting to be served by the TCM. In order to reduce contention among transactions, a separate precommit log is assigned to each core. Figure 2.5 provides a

```
TCM() {
  for entry precommitTX in PrecommitLogs
    if (precommitTX.Ready)
      if (ConflictCheck(precommitTX))
        Grant commit permission to precommitTX
      else
        Abort precommitTX
}
ConflictCheck(precommitTX) {
  for entry committedTX in CommitLog {
    if (precommitTX.startVersion < committedTX.commitVersion)
      if HashCollision(precommitTX.rdSig, committedTX.wrSig)
        return 0;
  }
  if !(precommitTX.readOnly){
    Go through WBActionList
    wait for concurrent conflicting
    WBs to finish
  }
  return 1;
}
```

Figure 2.5: Commit management in the TCM.

summary of what happens in the TCM during runtime.

The TCM constantly polls `Ready` flags of precommit log entries (first `for` loop in the figure). When it detects a `Ready` is set, it reads the transaction's start version and checks it against the commit versions of commit log entries (in the ConflictCheck function). If the start version of the committing transaction is less than the commit version of a commit log entry, we know that their execution has overlapped at some point in time. Therefore, they should be checked for possible conflicts (in the `HashCollision` function). In case of a hash collision between the signatures, the committing transaction is instructed to abort by setting the *Abort* flag in its header. If the committing transaction passes the check against all overlapping commit log entries, it is safe to be committed. This is all that needs to be done for read-only transactions. Therefore, the TCM sets the *Commit* flag in the transaction header. It is not necessary to copy any information about read-only transactions to the commit log.

However, the mechanism is more subtle for writing transactions. Since we want to avoid having individual locks for writing back the write set to memory, the TCM needs to make sure no concurrent writes are happening to the same address during writeback. The TCM uses a secondary structure called the writeback action-list (`WBActionList`) for this purpose. The action-list has the same number of entries as the active threads in the system. At any given time, it contains the write signatures of the transactions that have passed the commit check in the TCM and are writing back their write set to the memory. When a transaction is ready to commit, the commit manager checks its write signature against all write signatures in the writeback action-list. If there is no collision, the commit manager sets the *Commit* flag in the transaction header and writes the transaction's write signature to the action-list. Otherwise, it keeps checking the list until the colliding entry has finished writing back. An extra bit is added to the list to make sure that TCM does not repeatedly keep checking the signatures that have passed the collision test with the current committing transaction before. These checks could potentially become the TCM's bottleneck, though we did not notice any considerable busy waiting in our experiments. Subsequently, the TCM writes the necessary information about the committed transaction to the commit log, and moves on to checking the next entry in the precommit log.

Since commit log entries are no longer needed after all overlapping transactions have finished, a clean up mechanism is required to remove unnecessary entries. For this purpose, we maintain a minimum start version (minSV) log which contains the start versions of all in-flight transactions. Each transaction adds an entry to this log at start time and removes it at commit or abort. After each transaction commit or abort, the TCM starts from the commit log head entry and checks it against the start versions in the minSV log. If there are no overlapping in-flight transactions with the commit log head entry, that entry is removed and the head pointer is incremented. We keep doing this until the head entry in the commit log has an overlapping in-flight transaction. The reason we decided to use a circular buffer for the commit log (as opposed to a linked-list buffer) is to avoid the extra overhead of maintaining a linked list. Our commit log model only allows us to remove entries from the head of the log and add entries to the tail.

2.3.4 Individual Transaction Commits

When a transaction reaches the commit point, it fills up an entry in its precommit log with a pointer to its transaction header and sets the entry's `Ready` flag. Subsequently, it keeps polling *Commit* and *Abort* fields, waiting for them to be filled by the TCM. In order to avoid busy waiting at this point, we can relinquish the core[1] which is particularly useful when we have a larger number of threads than cores.

After a transaction receives commit permission from the TCM, it walks through its write set and writes back the actual values to memory. Because the TCM has already made sure that there are no concurrent transactions writing to the same locations, the committing transaction does not need to lock any memory locations. We chose to use a lazy version management strategy, because an eager version management system without locks introduces many complications in rolling back updates to memory locations after a conflict.

To minimize the overhead of individual transactional loads, a lazy conflict detection scheme

[1] In Linux, this can be done using `sched_yield` function.

is employed. This works particularly well for speculation support in loop parallelism, because minimum transactional load overhead is important for gaining performance from parallelizing loops. Furthermore, conflicts are rare due to the smart loop selection, and trying to detect conflicts eagerly at each transactional load provides no extra benefit. In eager conflict detection mechanism, since transactions are checked for conflicts at each load and store, the possibility of having zombie transactions is really low. However, eager conflict detection incurs substantial overhead on individual transactional operations.

Lazy conflict detection makes STMlite vulnerable to zombie transactions. These transactions may never reach the commit point and the commit manager normally does not get the chance to force them to abort. As a matter of fact, zombie transactions are particularly bad for our implementation because their corresponding entries remain valid within the minSV log and prevent the other commit log entries from being cleaned up. However, we can exploit the minSV log to resolve the zombie transaction issue. Each time we go through the commit log reading the minSV entries, if the difference between the start version of a particular transaction and the global clock is more than a threshold, the TCM identifies the corresponding transaction as a potential zombie. Subsequently, the TCM checks the suspicious transaction's read signature against write signatures of the commit log entries (although it has not reached the commit point yet). If there is a conflict, the TCM forcibly aborts the zombie transaction by sending an abort signal. We have a signal handler in each transaction that calls the abort function whenever it receives the TCM's abort signal. Otherwise, the TCM concludes that the suspicious zombie was just a long running transaction and avoids aborting it. In this work, since we do not parallelize loops with complicated linked list operations (which are the main sources of zombies transactions), the possibility of having zombies is quite low in our framework.

2.4 Loop Parallelization Using STMlite

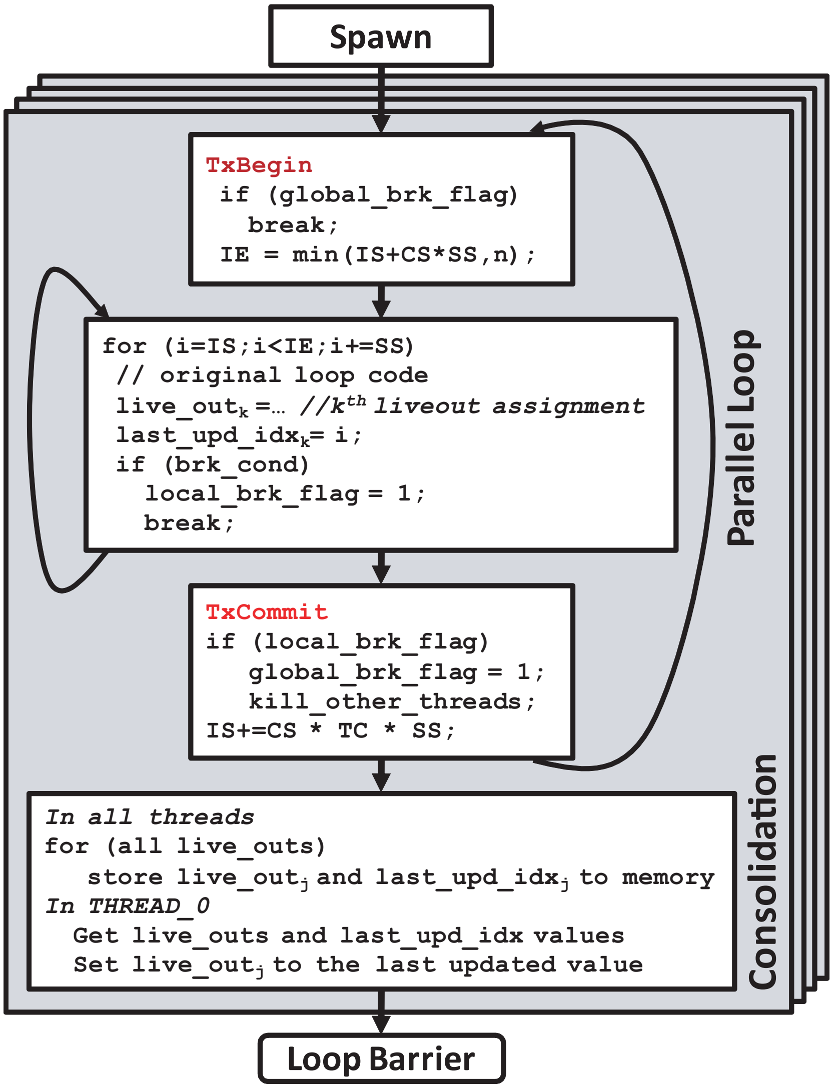
In this section, we introduce our loop parallelization framework and customizations made to STMlite for parallelizing speculative DOALL loops. Our framework successfully handles loops with cross iteration control dependences (e.g., while loops) as well as normal counted loops.

The general structure of our parallelization framework follows the code generation schema used in [93]. However, using that framework without the extra hardware support imposes a large overhead on the execution time. At the same time, STMlite gives us the opportunity to simplify the parallelization framework by exploiting some of its underlying features that are already used for providing transactional correctness.

2.4.1 Baseline Parallelization Framework

The purpose of the parallelization framework is to distribute loop execution across multiple cores. In this framework, DOALL loops are categorized into DOALL-counted and DOALL-uncounted types. In DOALL-counted loops, the number of iterations is known at runtime, whereas in DOALL-uncounted loops, this number is dependent on the loop execution (e.g., while loops). In these cases, starting every iteration is dependent on the outcome of exit branches in previous iterations (cross iteration control dependence).

Figure 2.6 shows the detailed implementation of the framework. In this scheme, loop iterations are divided into chunks. The operating system passes the number of available cores to the application and the framework is flexible enough to use any number of cores for loop execution.



```
TxBegin
 if (global_brk_flag)
   break;
 IE = min(IS+CS*SS,n);
```

```
for (i=IS;i<IE;i+=SS)
 // original loop code
 live_out$_k$ =… //$k^{th}$ liveout assignment
 last_upd_idx$_k$= i;
 if (brk_cond)
   local_brk_flag = 1;
   break;
```

```
TxCommit
if (local_brk_flag)
   global_brk_flag = 1;
   kill_other_threads;
IS+=CS * TC * SS;
```

```
In all threads
for (all live_outs)
   store live_out$_j$ and last_upd_idx$_j$ to memory
In THREAD_0
  Get live_outs and last_upd_idx values
  Set live_out$_j$ to the last updated value
```

Figure 2.6: Overview of the parallelization framework (CS: chunk size, IS: iteration start, IE: iteration end, SS: step size, TC: thread count).

An outer loop is inserted around the original loop body to manage parallel execution between different chunks. The main thread (THREAD_0), which runs the sequential parts of the program, spawns the required number of threads at the start of the application. When a parallel loop is reached, a function pointer containing the proper loop chunk along with necessary parameters is sent to each spawned thread and they start the execution of loop chunks.

In order to capture the correct live-out registers after parallel loop execution, we use a set of registers called last_upd_idx, one for each conditional live-out (i.e., updated in an if-statement). When a conditional live-out register is updated, the corresponding `last_upd_idx` is set to the current iteration number to keep track of the latest modifications to the live-out values. If the live-out register is unconditional (i.e., updated in every iteration), the final live-out value can be retrieved from the last iteration and no tracking by `last_upd_idx` is needed. It should be noted that loop chunks in the framework do not share any local memory variables on stack. Otherwise, the loop would have unresolvable cross iteration dependences and would be unparallelizable. This leads to one of the simplifications we made in STMlite which is the elimination of the handling mechanism needed for speculative local memory variables. Following is a description of the functionality of each segment in Figure 2.6.

Spawn: `THREAD_0`, the main thread, sends the function pointer pointing to the start of loop chunks to the in-flight threads through memory. It also sends along the necessary parameters (chunk size, thread count, etc.) and live-in values.

Parallel Loop: The program stays in the parallel loop segment as long as there are some iterations to run and no break has happened. In this segment, each thread executes a set of chunks. Each chunk consists of several iterations starting from IS (iteration start) and ending at IE (iteration end). The value of IS and IE are updated after each chunk using the chunk size (CS), thread count (TC), and step size (SS). Each chunk is enclosed in a transaction using `TxBegin` and `TxCommit` function calls. In order to ensure correctness, an abort signal is sent to transactions running higher iterations if a conflict is detected.

One important requirement for parallelizing loop chunks is to force in-order chunk commit. This is necessary for maintaining correct execution and enabling partial loop rollback and recovery. The TCM in STMlite already provides the means to enforce ordering among transactions in the commit log. The same infrastructure can be used for in-order chunk execution as well. Therefore, there is no need for send/receive instructions and a scalar operand network as was used in [93]. However, some extra book-keeping data is required both for STMlite and the parallelization framework. Since this is mostly done in STMlite and it is almost transparent to the generated code, we explain these necessary steps in the next subsection detailing the interaction between STMlite and loop parallelization.

For uncounted loops, if a break happens in any thread, higher transactions are not aborted immediately because thread execution is speculative and the break could be false. Instead, the `local_brk_flag` variable in each thread is used to keep track of breaks in individual chunks. If a transaction commits successfully with its `local_brk_flag` set, the break is no longer speculative, and a transaction abort signal is sent to all threads. In addition, a `global_brk_flag` is set, so that all threads break out of the outer loop after restarting the transaction as a result of the abort signal. The reason for explicitly aborting higher iterations is that, if an iteration is started by misspeculation after the loop breaks, it could produce an illegal state. The execution of this iteration might cause unwanted exceptions or might never finish if it contains inner loops. This procedure of explicit handling of breaks has the benefit of avoiding zombie transactions, and although STMlite can handle zombies, this explicit handling has much lower cost.

Consolidation: After all cores are done with the execution of iteration chunks, they enter the consolidation phase. Each core sends its live-outs and `last_upd_idx` array to `THREAD_0` through memory. `THREAD_0` picks the last updated live-out values. All other threads keep waiting for chunks from other parallel loops later in the program.

Since the goal is to provide a low-cost software-based parallelization mechanism, most of the extra code is kept outside the main loop body, and is executed only once per chunk.

2.4.2 Interaction of Parallel Loops with STMlite

As mentioned in the previous subsection, in-order commit of individual loop chunks is crucial for correct parallel execution. In order to enforce that requirement, we add another data structure, called the loop chunk commit log (LCCL), to the TCM. This log contains the loop ID of the last committing parallel loop and the chunk ID of the last committed chunk in that loop. The loop ID is assigned to each loop statically at compile time. It should be noted that our model allows only one in-flight parallel loop at a time by including a lightweight barrier at end of each chunk. Thus, there will be no problem if a parallel loop is invoked twice, because there is guaranteed to be no previous instances of this loop running. This is important, because if two in-flight loops have the same loop ID, they can completely distort each other's execution. The only problem is the case of loops in recursive functions. In this work, we do not parallelize loops with recursion. However, even in that case, a hash value based on the call site trace of the loop can be used to uniquely identify individual loops [76].

We reuse the initial value of IS (iteration start) which is computed at the beginning of each loop chunk as the chunk ID. When a loop chunk reaches the commit instruction, it writes its loop ID, chunk ID, chunk size, and the loop's first chunk ID to the precommit log. After the TCM reads in an entry from the precommit log, it performs one of the following two operations:

1. If the loop ID in the precommit log does not match the LCCL's committing loop ID, it infers that a new loop has started committing. Subsequently, it writes the new loop ID and the loop's first chunk ID to the LCCL. If the committing chunk is the first chunk

in the loop, the TCM proceeds with the commit process. Otherwise, it just moves on to checking the next precommit log entry. This is because a chunk's commit process should not be started until all earlier chunks have been committed (i.e, have got commit permission from the TCM and started the writeback process).

2. If the loop ID of the committing chunk matches the entry in the LCCL, the TCM checks to see if the current chunk is right after the last committed chunk. If so, it proceeds with the chunk's commit process. Otherwise, it starts checking the next precommit log entry.

The above mechanism provides low-cost commit ordering by adding minimal complexity to the STMlite library. This integration of loop parallelization with STMlite leads to an efficient parallel loop execution platform.

2.5 Results

We set up two sets of experiments. First, we evaluated how STMlite performs in a typical transactional environment using the STAMP transactional benchmarks [62]. In the second set of experiments, we implemented the code generation part of the parallelization framework in the LLVM compiler [52]. Using this framework, a set of SPECfp benchmarks and several kernel benchmarks are parallelized. All benchmarks were written in C or converted from Fortran to C[2]. While the original Fortran applications can be parallelized using compilers such as SUIF [44], Fortran to C conversion introduces a large number of pointer variables, thus compiler analysis alone was insufficient to parallelize all applications. For SPECint benchmarks, as previous works have shown [93, 53], the level of loop-level parallelism is quite low, thus the overhead

[2] Fortran to C conversion was done using the *f2c* tool with *-a* flag.

of using an all-software parallelization approach is too large to yield meaningful performance gains. More sophisticated parallelization techniques for integer applications are possible, such as those proposed by [26], and can lead to substantial gains. However, we have not implemented these transformations within our compiler system, yet they are orthogonal to what we are doing here.

2.5.1 STMlite on STAMP

We measured the performance of the STAMP benchmarks on a SunFire T2000 with an 8-core UltraSPARC T1 processor, running Solaris 10. We compare our performance with an implementation of the Transactional Locking 2 (TL2) software transactional memory [35]. Figure 2.7 shows the benchmark speedups on STMlite and TL2, both normalized to sequential execution. The number of cores in the STMlite results include the one extra core used for the TCM. For example, the 8 core results in STMlite have 7 computation cores and one TCM core. Thus, STMlite results start from two cores on the horizontal axis.

As can be seen, STMlite noticeably outperforms TL2 in both high and low contention executions of the `Vacation` benchmark. This is mainly because this benchmark has long transactions with a large number of loads. Therefore, the traditional STM performs poorly due to the high overhead of transactional loads and it can hardly achieve speedup over sequential even with 8 cores. However, using STMlite is particularly beneficial in these types of benchmarks. The overhead of transactional loads in our model is minimal due to the complete elimination of the read set. Furthermore, long length transactions and relatively low contention amortize the slight serialization effect that happens at commit time. Therefore, our model achieves about 2.5x and

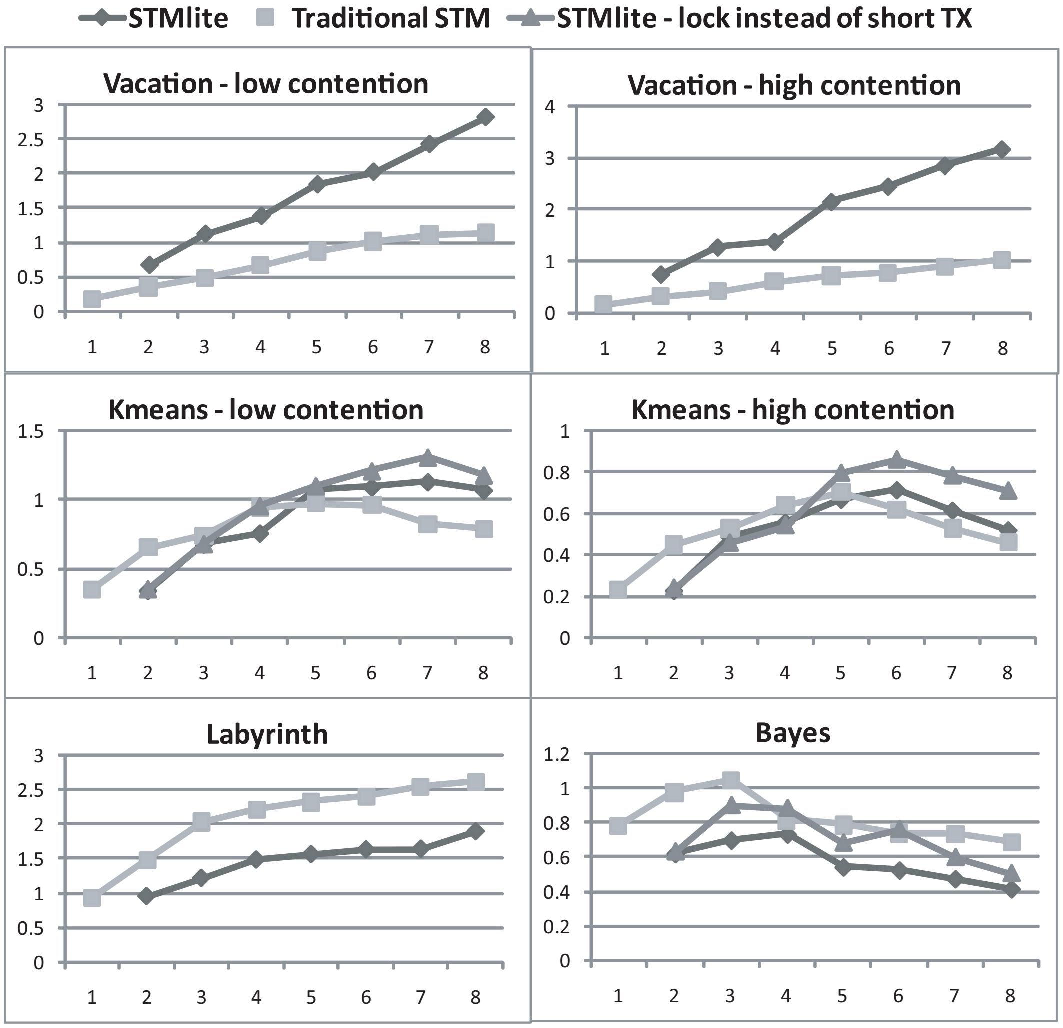


Figure 2.7: STMlite performance on STAMP benchmarks. The vertical axis shows the speedup compared to the sequential execution and horizontal axis is the number of cores. The number of cores in STMlite includes one core that is used for the TCM.

3.1x speedup over TL2 with 8 cores, which is quite close to the speedup achieved by previous hybrid schemes [63].

STMlite follows the performance of TL2 in `Kmeans`, `Labyrinth` and `Bayes`. First, it should be noted that poor scalability from 4 to 8 cores in `Kmeans` and from 2 to 8 cores in `Bayes` is mainly due to the fact that these benchmarks contain heavy floating point computations. Since the UltraSPARC processor only has a single floating point unit that is shared by all proces-

```
/* Original Kmeans Code*/           |  /* Lock-based Kmeans Code */
TxBegin;                            |  pthread_mutex_lock(&mutex1);
start = TxLoad(global_i);           |  start = global_i;
TxStore(global_i, (start + CHUNK)); |  global_i = start + CHUNK;
TxCommit();                         |  pthread_mutex_unlock(&mutex1);


TxBegin();                          |  pthread_mutex_lock(&mutex2);
TxStore_f(global_delta,             |  global_delta += delta;
  TxLoad_f(global_delta) + delta);  |
TxCommit();                         |  pthread_mutex_unlock(&mutex2);
```

Figure 2.8: Small transactions in Kmeans working on global data and their equivalent lock-based implementation.

sors, these floating point computations become the sequential bottleneck of parallel execution, especially with higher number of threads.

The main reason STMlite performs similarly to TL2 in these benchmarks is the short length of transactions in Kmeans and relatively high rate of contention in Bayes and Labyrinth. Therefore, the savings STMlite gets in transactional loads, transactional stores, and writebacks gets offset by the extra overhead of communications between execution transactions and the TCM. However, STMlite is still about 15% to 30% faster than TL2 in Kmeans for 4 and 8 threads. An interesting issue we found while looking through the performance bottlenecks of STMlite in Kmeans, is that there is a small transaction in the source code towards the end of the program that increments a global variable in all transactions (Figure 2.8). This part of the code causes a large number of transaction aborts in STMlite, which incurs a high cost considering the short transaction lengths. Whereas in TL2, since the library is acquiring locks for each address during writeback and uses a back-off mechanism if the lock is not free, there are fewer transaction aborts. In order to validate this observation, we placed a global lock around the transaction

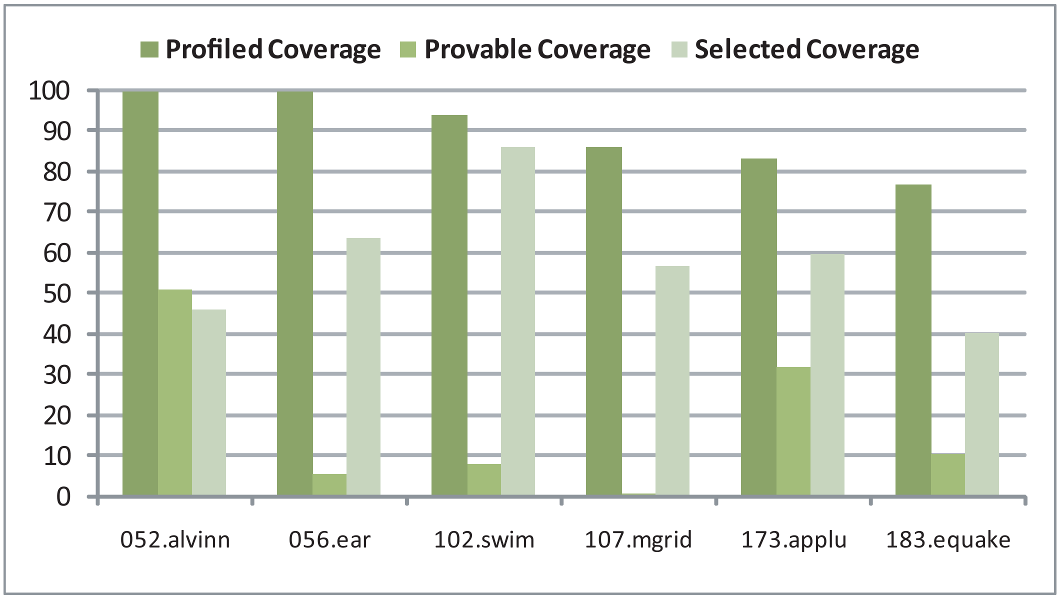


Figure 2.9: Profiled DOALL, provable DOALL and selected parallel loop coverage. The vertical axis shows fraction of sequential execution.

in Figure 2.8 and changed the transactional loads and stores to normal ones. The performance of the resulting execution is also shown in Figure 2.7. This change in the benchmark did not affect the runtime for TL2 – since TL2 essentially does the same thing in short transactions. As can be seen in the figure, although STMlite still suffers from lack of enough floating point units, it performs better after replacing the small transaction with locks in `Kmeans` and `Bayes`.

2.5.2 STMlite on Parallelized Sequential Programs

Figure 2.9 shows the fraction of dynamic sequential execution that can be parallelized in several SPECfp benchmarks.[3] The first bar, profiled coverage, shows the fraction of sequential execution in loops identified as DOALL after profiling. The second bar, provable coverage, is the fraction of sequential execution spent in loops that could be statically identified as DOALL at compile time using LLVM's memory dependence analysis. As can be seen, a non-trivial

[3] These applications are a subset of SPECfp92/95/2000 that had moderate to high amount of loop level parallelism.

percentage of DOALL coverage is obtained only after profiling, Finally the third bar, selected coverage, shows fraction of loops that were eventually parallelized.

It should be noted that not all the loops included in the coverage numbers are suitable for parallelization. There are many DOALL loops in these applications that do not contain any computation, or the computation is not substantial. For instance, parallelizing a loop which initializes an array's elements to zero or increments all elements in an array, can not provide much benefit, since the overhead of parallelization would be more than the actual work in these loops. Therefore, we added a loop selection heuristic in our compiler which, according to the profile data, computes a "parallelizability" metric based on the total number of dynamic operations in the loop, number of iterations and total number of loop invocations in the program. The last bars in Figure 2.9 shows the total coverage of DOALLs that passed this metric.

We have parallelized all these loops using the framework introduced in Section 2.4.1. During the code generation pass, according to the static memory dependence analysis data, we performed a selective replacement of the loops' loads and stores with `TxLoad` and `TxStore` function calls. We essentially avoid changing loads and stores that can be proved to cause no cross iteration dependences.

As a step towards showing the effectiveness of our approach, we first tried the parallelization framework and STMlite on four kernel benchmarks: `RLS`, `FMradio`, `DCT`, and `beamformer`. `RLS` is an implementation of recursive least squares filter which is used in system identification problems and time series analysis. `DCT` performs a discrete cosine transform and is used in image processing applications. `FMradio` and `beamformer` are two streaming applications from the StreamIt benchmark suite [85]. All these benchmarks have very high profiled DOALL

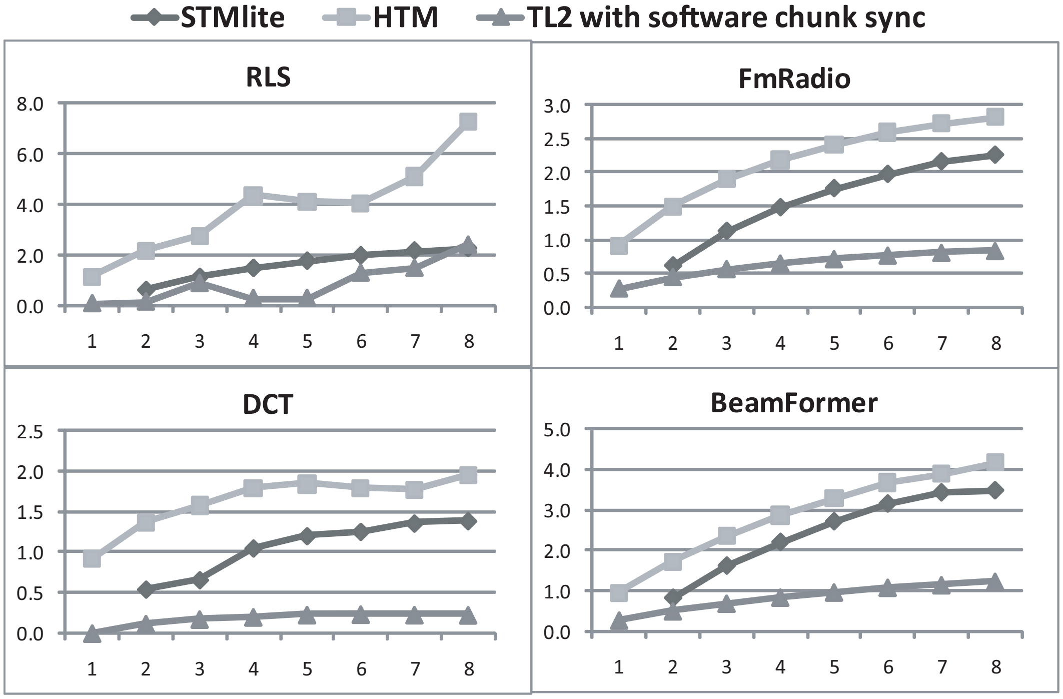


Figure 2.10: STMlite performance on automatically parallelized kernel benchmarks. The vertical axis shows the speedup compared to the sequential execution and horizontal axis is the number of cores. The number of cores in STMlite includes one core that is used for the TCM.

coverage. Figure 2.10 shows the achieved speedup using STMlite and TL2. The STMlite results include the resource used for the TCM (1 extra core). Furthermore, since TL2 does not have any primitives for supporting chunk commit serialization, we implemented a software-based send/recv mechanism similar to [93]. Lastly, we estimated the results on a similar system with HTM support by replacing all transactional loads and stores with normal ones. This would represent a best-case HTM, and since we are only doing this for performance measurement, we ignore the possibility of incorrect execution due to the lack of proper speculation and we only take into account the performance numbers for executions that complete successfully. As can be seen, STMlite outperforms TL2 with software based chunk synchronization by as much as a factor of 3x in `FMradio`. In `beamformer` and `DCT`, STMlite follows the HTM results quite closely. For `RLS`, STMlite performs poorly compared to HTM results due to high number of

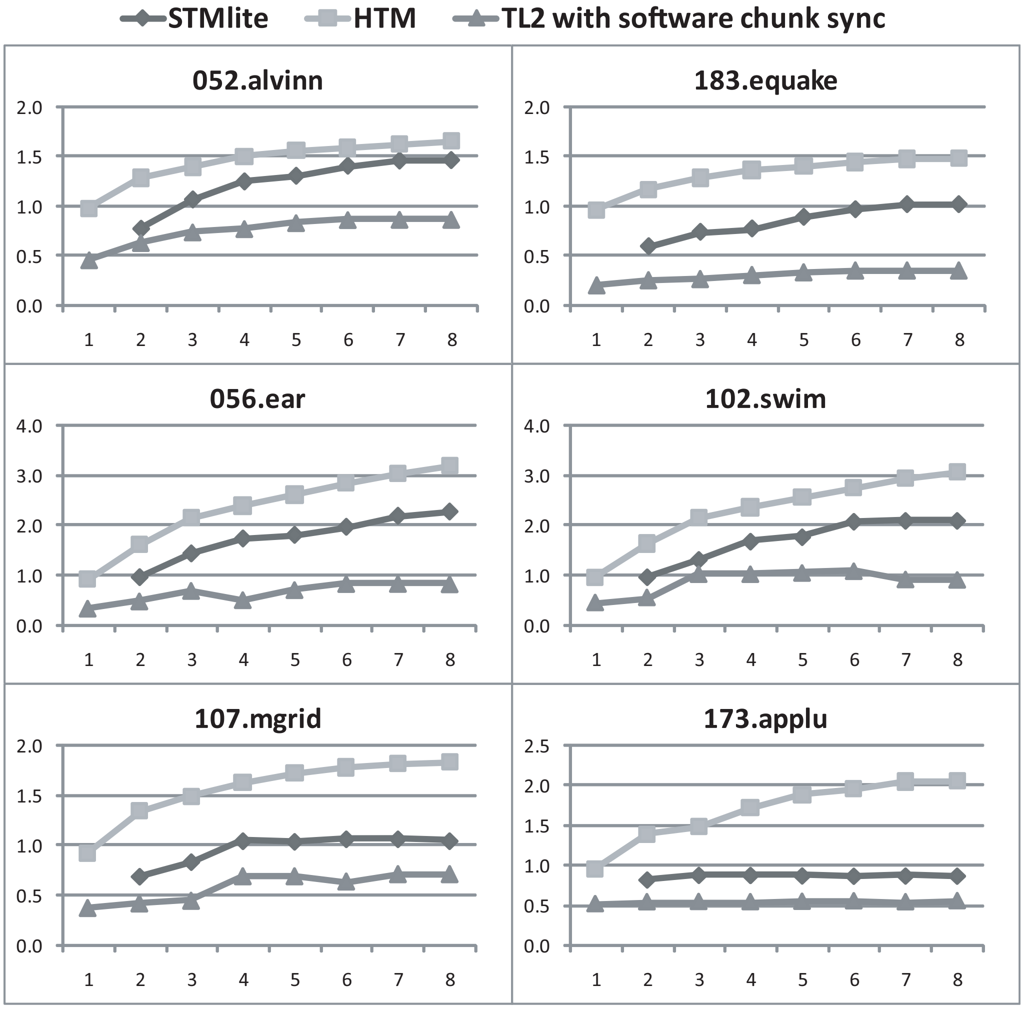


Figure 2.11: STMlite performance on automatically parallelized SPECfp benchmarks. The vertical axis shows the speed up compared to the sequential execution and horizontal axis is the number of cores. The number of cores in STMlite includes one core that is used for the TCM.

transactional operations, yet it still achieves 2x speedup over sequential for 8 threads.

Returning to SPECfp, Figure 2.11 shows the speedup for these benchmarks. Runtime values are normalized to the sequential execution of the program. The figure shows that we achieve 0.6x to 2.2x speedup compared to sequential by going from two to eight cores.

One of the reasons for performance degradation in TL2 with software synchronization is the lack of library support for enforcing commit ordering in TL2. Adding this explicit software synchronization has a noticeably negative impact on the performance. Performance degradation would be even more in traditional TM systems with eager conflict detection, such as [82]. As previous works have also suggested [81], workloads with transactions that have large read sets and low contention (similar to our parallelized sequential workloads), perform poorly with eager conflict detection. This is because eager conflict detection adds extra overhead to transactional loads and stores, but since conflicts are rare, it does not help improving the performance.

STMlite achieves decent speedup compared to HTM results and outperforms TL2 with software chunk synchronization in `052.alvinn`, `056.ear`, and `102.swim`. This is due to the lower overhead of transactional operations in STMlite which makes it quite efficient with moderate number of these operations. However, the relative STMlite achieved speedup, while being noticeably higher than TL2 with software synchronization, is quite low compared to HTM in other benchmarks. In SPECfp benchmarks, the parallelized loops contain a large number of memory operations that may cause cross iteration dependences based on the static analysis and therefore need to be transactified. Changing these operations to transactional versions causes the parallelized versions to become slow in some cases. Software-based speculation mechanisms are useful for parallelization in cases that the number of speculative variables is low, otherwise, the speculation mechanism amortizes the benefit caused by parallelization.

2.5.3 Effects of static memory analysis and signature sizes

To better understand the tradeoffs involved in compilation and execution parameters, we ran two other experiments. In the first experiment, we measured the achieved speedup with and without selective replacement of loads and stores with transactional versions. As mentioned before, LLVM's memory dependence analysis is used to avoid transactifying memory instruc-

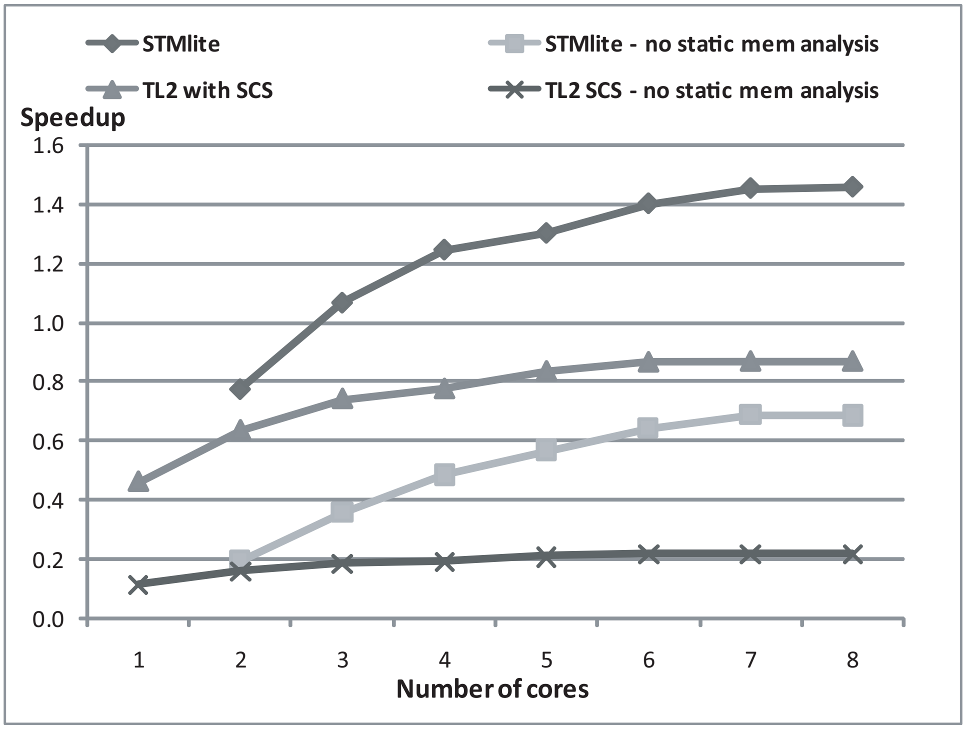


Figure 2.12: Effect of using static pointer analysis on speedup for `052.alvinn`.

tions that provably do not cause cross iteration dependence. Figure 2.12 shows the result of this experiment on the `052.alvinn` benchmark. As can be seen, filtering out unnecessary transactional operations, while keeping the necessary ones, has a great impact on performance in both STMlite and TL2. This result further proves that software speculation systems are best suited for applications in which speculation is applied to a limited number of memory variables.

Our second experiment involves changing the signature size and studying the resulting performance impact. The effect of changing signature sizes on STMlite's performance is interesting. There is a subtle tradeoff involved in determining the right signature size. Larger sizes reduce the number of false positives and thereby reduce re-execution of correct transactions. However, at the same time, they lead to more time consuming signature operations. Since STMlite is dependent on these operations in several parts of the implementation, this can

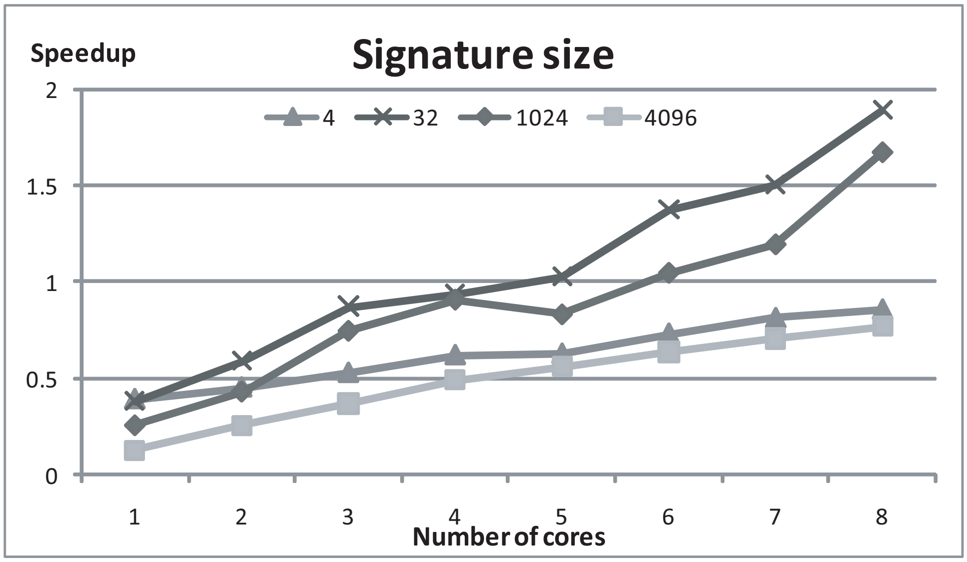


Figure 2.13: Effect of varying signature size on speedup for RLS
.

cause a noticeable performance degradation. Figure 2.13 illustrates this effect on the RLS kernel benchmark. Speedup values keep increasing up to signature sizes of 32, after which they start going down.

2.6 Related Work

There is a significant amount of previous efforts in the area of transactional memory. Harris *et al.* go through a detailed survey of different transactional memory techniques in [48].

In particular, Shavit *et al.* proposed the first implementation of software transactional memory in [77]. Several other works such as DSTM [50] and OSTM [47] proposed non-blocking STM implementations. A major part of non-blocking STMs is maintenance of publicly shared transaction structures which contain the undo information. In our implementation, the transaction structures only need to be visible to the TCM and individual executing transactions, keeping contention on those structures to a minimum. The authors in [49, 18] proposed a lock-based approach where write locks are acquired when an address is written. Also, they maintain a read set which needs to be validated before commit. In our STMlite design, no locks are required and correctness is guaranteed by the commit manager. Furthermore, we eliminate the need for the read set, which reduces the overhead of transactional loads and transaction commits. [36] proposes the Transactional Locking implementation which maintains a read set and a write set during transaction execution. Subsequently, at commit time, it acquires locks for each individual write set entry and writes back the data after the lock is secured. Also, the read set is checked during commit to ensure consistency.

There is also a large body of work in parallelization of sequential applications. Hydra [45] and Stampede [84] were two of the earlier efforts in the area of general purpose program parallelization. The POSH compiler [53] uses loop-level parallelization with TLS hardware support. The authors in [93] proposed compiler transformation to extract more loop level parallelism from sequential programs. The compiler transformation part of that work is orthogonal to what we are doing and can be applied simultaneously here. Speculative decoupled software pipelining [87] is another approach that focuses on extracting parallelism from loops with cross iteration dependencies. In that work, they distribute a single iteration of the loop over several cores. The SUDS framework [38] performs automatic speculative parallelization of applications for the RAW processor. This system relies on the special architectural features in RAW to accomplish efficient speculative state management and synchronization, such as the scalar operand network. However in all these works, hardware TLS or transactional memory support and additional hardware mechanisms for synchronization are required. Whereas in this work, we are looking at a software-only solution and although our achieved speed up in some cases is lower than these works, we have the advantage of running our system on commodity hardware.

Ceze *et al.* [29] proposed the idea of using Bloom filters to represent read and write sets for transactions. They showed how, with specialized hardware, transaction state can be maintained through signatures with less overhead. This technique was extended in LogTM-SE [92] and SigTM [63], which are hybrid TM systems requiring no modifications to hardware caches. Our work uses the idea of storing Bloom filter-based read and write sets in software data structures, alleviating the need for the extra hardware. Authors in [49] use software hashing to remove duplicates in the read-log and undo-log of the same transaction, whereas in STMlite, it is used for conflict detection between different transactions.

The most similar speculation management mechanism to ours is RingSTM [82] that uses a global ring structure to organize committing transactions. They use Bloom filters to represent read and write sets for transactions. However, because the ring is global, all threads face contention for ownership of the ring during commit, and prioritization is required to prevent starvation. Meanwhile, STMlite has thread local precommit logs and can relinquish the cores while the corresponding transaction is waiting for the commit manager to validate the transaction. Our commit log works in a round-robin fashion, ensuring all threads waiting to commit are serviced equally. Furthermore, in [82], the read signature is checked against several write signatures at each transactional load (eager conflict detection), which adds considerable overhead. However, in STMlite, transactional load overhead is minimal because the only extra operation added is insertion of the address in the read signature. This makes our model more prone to zombie transactions, but as mentioned in Section 2.3.4, the possibility of having zombies in parallelized loops is quite low, though STMlite can still handle them successfully.

Furthermore, we have customized STMlite to work for loop parallelization. This customization would be more complicated in RingSTM. The reason is that transaction commit is done by individual transactions after checking against the write signatures of ring elements. Therefore, if a loop chunk does not get a chance to commit in the first try (due to an unfinished previous chunk), there would be no efficient way of checking again later in the execution. The only way would be to use a back off mechanism and check back from time to time, which is inefficient. Whereas in STMlite, since the TCM is in charge of ordering loop chunks for commit, even if a chunk misses its chance, the TCM makes sure that it would be checked again in a timely manner.

FlexTM [79] adds mechanisms in hardware to coordinate read and write signature checking, speculative updates to caches and eager notifications to transactions about coherence events. They propose software mechanisms for deciding how to manage conflicts and for choosing appropriate conflict management and commit protocols.

2.7 Summary

As we move further into the multicore era, a major challenge in both hardware and software communities is exploiting the abundant computing resources made available by technology advancements. Automatic parallelization of applications is an appealing solution for utilizing these resources; however, parallelization efforts are commonly dependent on complex hardware changes such as adding speculation support. These changes are not yet popular among hardware manufacturers. On the other hand, software-based speculation support is still quite expensive in terms of performance to be widely used in parallel and parallelized applications. In this work, we strived to tackle these issues from two closely related angles. First, we minimize the overheads of software based transactional memory models by decoupling and centralizing the commit stage in STMlite. We also eliminate the need for maintaining a read set during loads and checking them during commit. Secondly, we are able to lower the overhead of loop parallelization by reusing some of the underlying structures of STMlite. We have shown that our work outperforms current transactional memory implementations on transactional benchmarks with large transactions while achieving similar performance in smaller transactions. Furthermore, we show that achieving real speculative speedup on sequential applications is possible without extra hardware support.

CHAPTER 3

Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism

3.1 Introduction

JavaScript was developed at Netscape in the early 1990's under the name Mocha to provide enhanced user interfaces and dynamic content on websites. It was released as a part of Netscape Navigator 2.0 in 1996, and since then it has become standard with more than 95% of web surfers using browsers with enabled JavaScript capabilities.Dynamically downloaded JavaScript programs combine a rich and responsive client-side experience with centralized access to shared data and services provided by data centers. The uses of JavaScript range from simple scripts utilized for creating menus on a web page to sophisticated applications that consist of many thousands of lines of code executing in the user's browser. Some of the most visible applications, such as Gmail and Facebook, enjoy widespread use by millions of users. Other applications, such as image editing applications and games, are also becoming more commonplace due to the ease of software distribution.

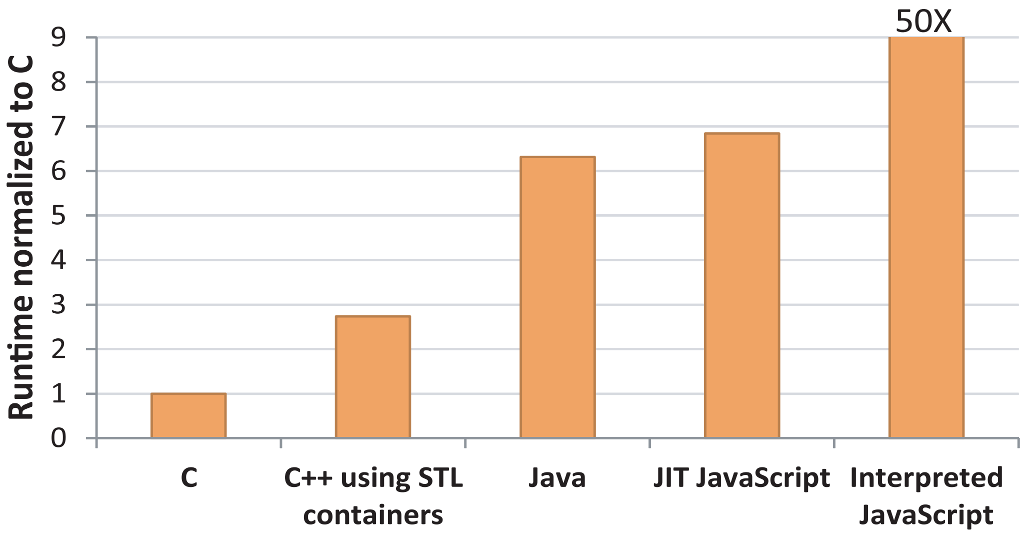


Figure 3.1: Performance of JavaScript, compared to C, C++ and Java.

As the complexity of these applications grows, the need for higher performance will be essential. However, poor performance is the biggest problem with JavaScript. Figure 3.1 presents the performance of an image edge detection algorithm implemented in a variety of programming languages on an Intel Core i7 processor. Not surprisingly, the C implementation is the fastest. Following are the C++ implementation with 2.8x slowdown and the Java version with 6.4x slowdown. The default JavaScript implementation is 50x slower than the C implementation. The performance gap occurs because JavaScript is a dynamically typed language and is traditionally executed using bytecode interpretation. Interpretation makes JavaScript execution much slower in comparison to the native code generated for statically typed languages such as C or C++. Using the trace-based optimizer in Firefox, TraceMonkey [40], that identifies hot execution traces in the code and compiles them into native code, the JavaScript execution is brought down in line with the Java implementation which is still 6.9x slower.

Bridging this performance gap requires an understanding of the characteristics of JavaScript programs themselves. However, there is disagreement in the community about the forms of

JavaScript applications that will dominate and thus the best strategy for optimizing performance. JSMeter [71] characterizes the behavior of JavaScript applications from commercial websites and argues that long-running loops and functions with many repeated bytecode instructions are uncommon. Rather, they are mostly event-driven with many thousands of events being handled based on user preferences. We term this class of applications as *interaction-intensive*.

While this characterization reflects the current dominance of applications such as Gmail and Facebook, it may not reflect the future. More recently, Richards *et al.* [73] performed similar analyses on a fairly large number of commercial websites and concluded that in many websites, execution time is, in fact, dominated by hot loops, but less so than Java and C/C++. Furthermore, an emerging class of online games and client-side image editing applications are becoming more and more popular. There are already successful examples of image editing applications written in ActionScript for Adobe Flash available online [8, 10]. There are also many efforts in developing online games and gaming engines in JavaScript [1, 14]. These applications are dominated by frequently executed loops and functions, and are termed *compute-intensive*.

The main obstacle preventing wider adoption of JavaScript for compute-intensive applications is historical performance deficiencies. These applications must be distributed as native binaries because consumers would not accept excessively poor performance. A circular dependence has developed where poor performance discourages developers from using JavaScript for compute-intensive applications, but there is little need to improve JavaScript performance because it is not used for heavy computation. This circular dependence is being broken through the development of new dynamic compilers for JavaScript. Mozilla Firefox's TraceMonkey [40] and Google Chrome's V8 [3] are two examples of such efforts. While these engines address a large portion of inefficiencies in JavaScript execution, they do not provide solutions to scale performance.

With multicore architectures becoming the standard for desktops, servers, and smart phones, it is apparent that parallelism must be exploited in JavaScript applications to sustain their ever increasing performance requirements. However, these applications are generally single-threaded because the language and run-time system provide little concurrency support. The primary problem is that the document object model (DOM), used by JavaScript to interact with web pages and browser state, does not have a model for shared access. So, for example, since JavaScript has no internal locking mechanism, two threads may invoke multiple event handlers that change the same DOM object and create either race conditions or correctness bugs. Considering this limitation, developing parallel JavaScript applications manually would be cumbersome and error-prone.

To exploit hardware concurrency, while retaining the traditional sequential programming model, this work focuses on dynamic parallelization of compute-intensive JavaScript applications. Our parallelization technique, called *ParaScript*, performs automatic speculative parallelization of loops and generates multiple threads to concurrently execute iterations in these applications. A novel and low-cost software speculation system detects misspeculations occurred during parallel execution and rolls back the browser state to a previously correct checkpoint. Our techniques are built on top of the TraceMonkey engine [40], thus we retain the benefits of trace-based optimizations.

While speculative loop parallelization has an extensive body of prior work [26, 32, 37, 45, 53, 70, 84, 93], the key new challenge here is performing parallelization efficiently at runtime, while ensuring correct execution through runtime speculation without any extra hardware support. Recent efforts for speculative parallelization of C/C++ applications on commodity hardware ([57, 69, 86]) make extensive use of static analysis along with static code instrumentation with speculation constructs. Straight-forward application of these static parallelization techniques is ineffective for a number of reasons including the inability to perform whole program analysis, and expensive memory dependence analysis and profiling at runtime. In addition, without effective use of static analysis, considerable overhead of per-access logging and commit time operations involved in software speculation systems such as software transactional memories [18, 49], makes immediate use of these systems impractical in a dynamic setting. Previous proposals for dynamic parallelization [33] are also not suitable for our target due to high dependency on expensive extra hardware support. In this work, we show that practical speedups can be achieved on commodity multi-core hardware with effective runtime analysis and efficient speculation management. These techniques let us avoid high-cost static analyses and expensive hardware support for dynamic analysis [33] in traditional parallelization systems and the kind of per-access logging and commit time operations which contribute to high overheads in software speculation systems, such as STMs [18, 49].

In particular, the contributions offered in this chapter of the thesis are:

- An technique for automatically identifying parallelizable loops dynamically at runtime and a code generation scheme to create parallelized versions of JavaScript applications.
- An ultra-lightweight software speculation system consisting of selective checkpointing of the system state and a novel runtime dependence detection mechanism based on conflict detection using reference counting and range-based checks for arrays.

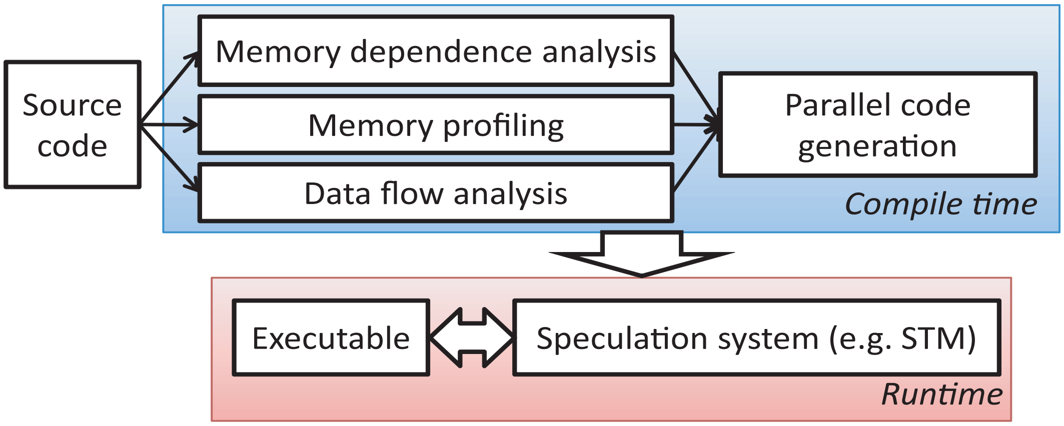


Figure 3.2: Static parallelization flow.

The rest of this chapter is organized as follows. Section 3.2 explores some of the challenges involved in dynamic parallelization and provide hints to potential solutions. Section 3.3 introduces novel analysis technique employed in ParaScript for low-cost parallelization detection at runtime and presents the code generation framework. ParaScript's ultra-lightweight speculation mechanism is proposed in Section 3.4, while Section 3.5 presents evaluation framework and experimental results. Section 3.6 reviews some of the related work, and finally, Section 3.7 provides a summary and concludes the chapter.

3.2 Dynamic Parallelization Challenges

There is a long history of static techniques for automatic and speculative parallelization of scientific and general purpose applications. A typical static parallelization framework is depicted in Figure 3.2. This framework uses memory dependence analysis (pointer analysis), memory access profiling and data flow analysis to find candidate loops for parallelization, after which generates the parallel code. This code is later run speculatively on the target system using a software or hardware memory speculation mechanism.

Static frameworks are usually optimized to achieve the best runtime performance by employing complex and often time-consuming compile-time analysis. However, in a highly dynamic setting like that of JavaScript execution, where the code is run via interpretation or dynamic just-in-time (JIT) compilation on the client's browser, applying these analyses at compilation time would be too expensive and will offset all potential parallelization benefits. It might even lead to slowdown in some cases. For instance, [42] recently proposed a static JavaScript pointer analysis for security purposes. They show that their analysis takes about 3 seconds to complete on a set of JavaScript programs with an average of 200 lines of code. While being an effective and fairly fast analysis for offline purposes, if the analysis were to be used at runtime, it could become quite prohibitive even for programs of that size, let alone larger applications that consist of hundreds of lines of code [73]. In addition to compiler analysis, the overhead of runtime memory profiling would be unacceptable – the work in [56] reported 100x to 500x increase in the execution time as a result of performing static memory profiling on C/C++ applications. Therefore, with the exception of simple data flow analysis and code generation (both of which are already done at runtime in JavaScript engines), a dynamic parallelization framework can not afford to utilize any of the static compile time steps in Figure 3.2.

Furthermore, by targeting commodity systems, an efficient and low-cost software speculation mechanism is required. Traditional software speculation mechanisms (e.g, STMs) focus on flexibility and scalability to support a wide variety of speculative execution scenarios and thereby increase the runtime by 2-5x [28]. Large overheads of memory operation instrumentation along with expensive checking mechanisms are two main sources of overhead. Recent proposals [57, 69, 86] introduce customized speculation systems for parallelization of C/C++

applications. However, as was reported in [57], without the aid of static memory analysis and profiling, their overhead could go up to 4x compared to sequential as well.

With these challenges in mind, we approach dynamic parallelization of JavaScript applications by proposing the following techniques:

Light-weight dynamic analysis: Limited information is carefully collected early on at runtime, based on which initial parallelization decisions are made. A low-cost and effective initial assessment is designed to provide optimistic, yet surprisingly accurate decisions to the speculation engine. Loops that require complicated transformations, high-cost runtime considerations or have high misspeculation risk are not chosen to be parallelized.

Low-cost customized speculation mechanism: A lean and efficient software speculation mechanism is proposed, which is highly customized for providing loop-level speculation support in JavaScript programs. A low-cost conflict detection mechanism is introduced based on tracking scalar array access ranges and reporting conflict if two threads access overlapping ranges. Parallelism-inhibiting heap references are also detected at runtime to guarantee correct execution. A reference counting scheme is introduced to detect potential conflicts in arrays of objects. Finally, an efficient checkpointing mechanism is also designed that is specialized for the JavaScript runtime setting. Two checkpointing optimizations, selective variable checkpointing and array clone elimination, are introduced to further reduce speculation overhead.

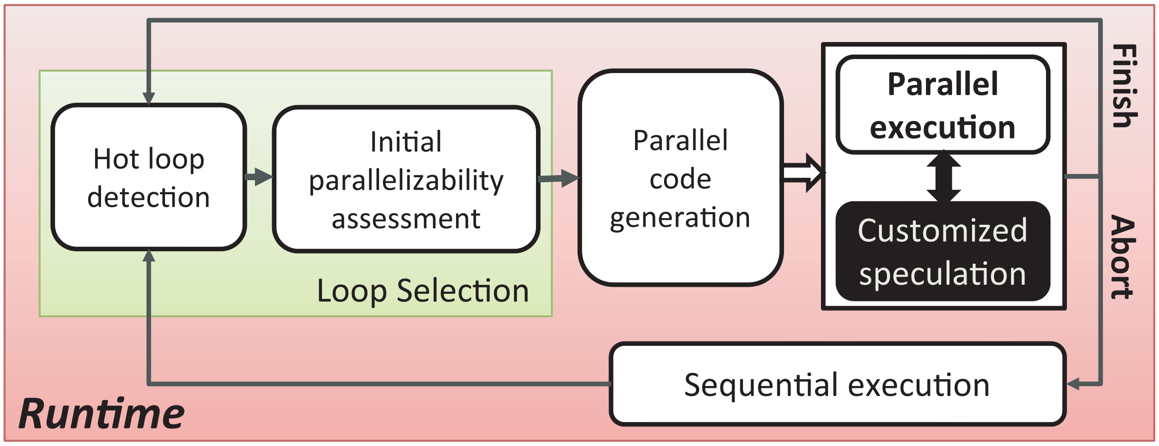


Figure 3.3: ParaScript dynamic parallelization flow.

3.3 Dynamic Analysis and Code Generation

In this work, we propose a dynamic mechanism, called ParaScript, for exploiting parallelism in JavaScript applications. Figure 3.3 shows the ParaScript execution flow at runtime. It first selects candidate hot loops for parallelization. Subsequently, in the dependence assessment step, using a mixture of data flow analysis and runtime tests, it detects immediately identifiable cross-iteration memory dependences during the execution of the first several loop iterations. In case of no immediate memory dependence, ParaScript dynamically generates a speculatively parallel version of the loop and inserts necessary checkpointing and conflict detection constructs. The detailed process of instrumentation with these constructs as well as their implementation is presented in Section 3.4.

The runtime system continues by executing the parallel loop speculatively. In case a cross-iteration dependence is found, the parallel loop is aborted, the system reverts back to the previous correct checkpoint and runs the loop sequentially. The failed loop is also blacklisted to avoid parallelization in further loop invocations in the same run. Furthermore, a cookie is stored in the browser's cache to blacklist the loop in future application executions as well.

To keep the framework and speculation rollback overhead low, only loops whose number of iterations is known at runtime (DOALL-counted, e.g., `for` loops) are considered for dynamic speculative parallelization. Implementing a general framework similar to [93] which handles DOALL-uncounted loops (e.g., `while` loops) needs a more elaborate framework and more frequent checkpointing, imposing negative impact on our goal of having a low-cost near-certain speculation and infrequent checkpointing mechanism.

Loops that contain document object model (DOM) interactions, HTTP request functions and several JavaScript constructs such as the `eval` keyword, `Function` object constructor, and `setTimeout` and `setInterval` keywords are not parallelized. Parallelizing loops with DOM interactions requires locking mechanisms on DOM elements inside the browser and also a heavy-weight checkpointing mechanism to take snapshots of DOM state at various times. Furthermore, most DOM accesses inside loops are to the same DOM element and inherently sequential. Therefore, parallelization will not yield much performance benefits. Accesses to different elements also require locking DOM tree structures which eventually makes the accesses sequential. Speculative execution of HTTP requests requires speculation and rollback support at the server side [88]. In this work, we focus on the client-side speculation and do not have this support. The `eval` keyword, `Function` constructor, `setTimeout` and `setInterval` take strings as arguments and execute them like normal JavaScript code at runtime. For instance, a call to `eval` or `Function` constructor could be the following:

```
var addFunction = new  Function("a" , "b", "return a+b;");
eval("a = 7; b = 13; document.write(a+b)");
```

The problem with using these constructs is that they introduce new code at runtime and the compiler has no access to the string contents and thereby, no analysis can be done on it before execution. In the following subsections, ParaScript stages are explained in more detail.

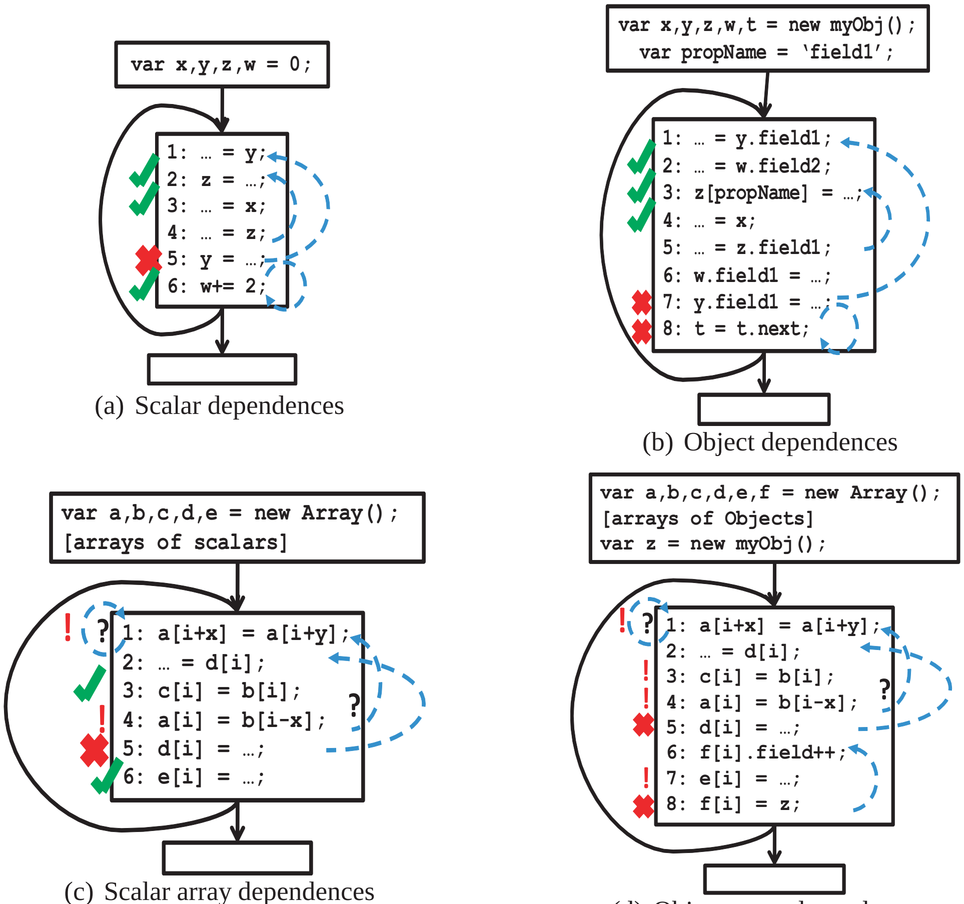
3.3.1 Dynamic Parallel Loop Selection

The loop selection process essentially consists of two steps: detecting hot loops and performing initial analyses to determine parallelization potential. During runtime, after a loop execution is detected, it is marked as hot if it passes the initial selection heuristic. This runtime heuristic takes into account iteration count and number of instructions per iteration.

In nested loops, if the outer loop's number of iterations is higher than a threshold, it is given the priority for being marked as hot by the system. Therefore, even if an inner loop is detected as hot, the system holds off on passing it to the next stage until a decision about the outer loop is made. However, if the outer loop turns out not to be hot or fails the dependence assessment, the inner loop is marked as hot. This mechanism is based on the intuitive assumption that parallelizing outer loops is always more beneficial and avoids premature parallelization of inner loops.

After a hot loop is detected, the system proceeds with the initial dynamic dependence assessment. There have been recent proposals for static pointer analysis in JavaScript [42]. There are also array analysis techniques such as the Omega Test [66] which could be employed for analyzing JavaScript code. However, due to the tight performance and responsiveness constraints on JavaScript execution, the overhead of using these traditional static analysis techniques at runtime in ParaScript is quite prohibitive.

In order to compensate for lack of static information and prior memory profiles, several tests



```
var x,y,z,w = 0;
```

```
1: … = y;
2: z = …;
3: … = x;
4: … = z;
5: y = …;
6: w+= 2;
```

(a) Scalar dependences

```
var x,y,z,w,t = new myObj();
var propName = ‘field1’;
```

```
1: … = y.field1;
2: … = w.field2;
3: z[propName] = …;
4: … = x;
5: … = z.field1;
6: w.field1 = …;
7: y.field1 = …;
8: t = t.next;
```

(b) Object dependences

```
var a,b,c,d,e = new Array();
[arrays of scalars]
```

```
1: a[i+x] = a[i+y];
2: … = d[i];
3: c[i] = b[i];
4: a[i] = b[i-x];
5: d[i] = …;
6: e[i] = …;
```

(c) Scalar array dependences

```
var a,b,c,d,e,f = new Array();
[arrays of Objects]
var z = new myObj();
```

```
1: a[i+x] = a[i+y];
2: … = d[i];
3: c[i] = b[i];
4: a[i] = b[i-x];
5: d[i] = …;
6: f[i].field++;
7: e[i] = …;
8: f[i] = z;
```

(d) Object array dependences

Figure 3.4: Examples of dependence instances in various variable categories. Solid and dashed arrows depict control flow and data flow edges respectively. Question marks on dashed arrows show possible data flow dependences. Check-marks show operations that are safe for parallelization, while cross-marked operations are the ones that hinder parallelisms. Both of these operation categories are identified by the simple JIT-time data flow analysis with DU chains. Exclamation marks are instances of possible cross iteration dependences that could not be figured out at JIT-time. Final safety decisions on these are deferred to further runtime analysis and speculation (Section 3.4).

are performed during the dependence assessment stage at runtime, in addition to the simple data flow analysis. Analysis is categorized based on four variable types: scalars, objects, scalar arrays and object arrays.

Scalars: Figure 3.4(a) shows an example of a loop with various forms of scalar dependences. ParaScript avoids parallelizing loops that contain any cross-iteration scalar dependences other than reduction and temporary variables. Using basic data flow analysis (with use-def chains), all instances of cross-iteration scalar dependences can be found [90].

One example of such dependence is variable `y` in Figure 3.4(a) which is both a live-in and is written inside the loop. Reduction variables such as accumulator (`w` in Figure 3.4(a)) and min/max variables are also instances of such dependences and can be found by simple pattern matching on the code. Furthermore, global temporary variables that are used inside each iteration and do not carry values past the iteration scopes are also identified (variable `z` in the example). A scalar which is used in the loop, but is not live-in belongs to the mentioned temporary variable category. These variables can be safely privatized, allowing the loop iterations to be executed in parallel. Scalar dependences caused by temporary locals are ignored, because they will automatically become private to each parallel thread.

Object references: Since JavaScript has no pointer arithmetic, objects can also be dealt with using data flow analysis. Figure 3.4(b) is an example of a loop with object dependences. Initially, statically resolvable object field accesses are identified and resolved. In JavaScript, object properties and methods can be accessed both using the "." operation and also by passing the field name as string to the object name as in line 3 of the code in Figure 3.4(b). In this example, `propName` is a constant string and can be resolved using a local constant propagation analysis [90] and converted to an access using the "." operation (`z.field1 = ...;`).

After this initial step, data flow analysis is used to find cross-iteration dependences caused by object references. Similar to scalars, a live-in reference which has been written inside the loop is an example of such dependences (`y.field1` in 3.4(b)). Likewise, global temporary objects are also handled by privatization (`z.field1`). Furthermore, this analysis successfully finds instances of parallelism-inhibiting linked list or tree operations in the loop (e.g., variable

t) and avoids loop selection.

Scalar arrays: Many cross-iteration dependences in scalar arrays can also be identified using data flow analysis. Array d in Figure 3.4(c) is an example of these cases. However, the data flow analysis can not determine the self-dependence on operation 1 and also the dependence between operations 4 and 1. In order to make parallelization decisions in these cases, we introduce an initial dependence assessment along with a set of range-based runtime checks. During the initial assessment, a copy of the loop code is made and all array accesses are instrumented in this copy to record accessed element indexes during each iteration in a dependence log. After a few iterations, these logs are checked to detect any immediate cross iteration array dependences. If no dependences are found, the system optimistically assumes that the loop is DOALL and selects it for parallelization. However, since this decision is based on only a few iterations, it might be wrong. ParaScript uses a range-based checking mechanism to detect possible conflicts at runtime. More details about this mechanism is discussed in Section 3.4.2.

Object arrays: Dynamic analysis for arrays of objects is more challenging. Similar to scalar arrays, data flow analysis can determine dependences caused by the array as a whole (assuming all elements point to the same location). For instance, array d in the example of Figure 3.4(d) causes a cross iteration dependence which can be identified by data flow analysis. Assuming that object z is loop-invariant, data flow analysis can also find the cross-iteration dependence caused through the assignment of z to elements of array f.

In addition to the data flow edges in Figure 3.4(d), any two array pairs in the loop can be dependent on each other through referencing to the same object during execution. Therefore, if there are writes to array elements inside the loop and the data flow analysis does not detect dependences, ParaScript optimistically assumes that the loop is parallelizable and employs a novel runtime mechanism to ensure correctness. During the checkpointing stage, before parallel loop execution, this runtime technique, which is based on object reference counting is applied to object arrays to avoid indirect writes to the same object through individual array element writes in different parallel threads. This mechanism, explained in more detail in Section 3.4.1, is effective in dynamic resolution of dependences that might be caused by arrays `a`, `c`, and `e` in Figure 3.4(d).

As we explained in this section, ParaScript is able to make optimistic parallelization decisions using only a single data flow analysis pass, while deferring complete correctness checks to the runtime speculation engine.

3.3.2 Parallel Code Generation

After the loop is chosen to be parallelized, the parallel loop's bytecode is generated at runtime according to the template in Figure 3.5. The chunk size determines the number of iterations after which speculative loop segments are checked for any possible conflicts (`conflictcheck()`). The value of chunk size is determined at runtime based on the number of threads, total number of iterations, iteration size and number of memory accesses. As the execution makes progress without conflicts, the runtime system increases the chunk size. A barrier is inserted at the end of each chunk (`chunkbarrier()`) and is released when all threads are done with their chunk execution after which the conflict checking routine (Section 3.4.2) is invoked.

To capture the correct value of live-out registers, if the live-out is unconditional and is up-

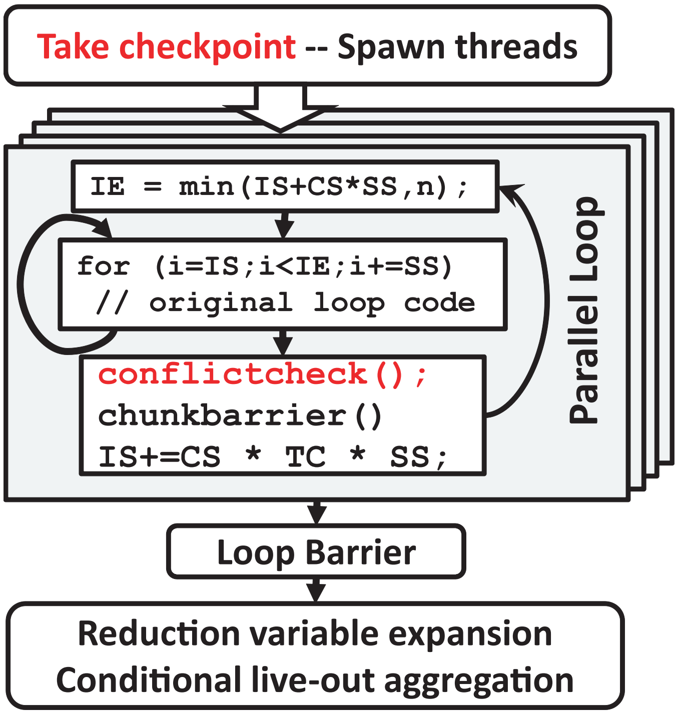


Figure 3.5: Dynamic code generation template (CS: chunk size, IS: iteration start, IE: iteration end, SS: step size, TC: thread count).

dated in every iteration, the final live-out value can simply be retrieved from the last iteration of the loop. For conditional live-outs (those updated in an if-statement), a set of registers called `update_index` are used. Whenever the live-out value is updated, the corresponding `update_index` is set to the current iteration number to keep track of the latest modifications to live-out values. After passing the loop barriers, the `update_index` values are checked to capture the correct final value for the live-out variable.

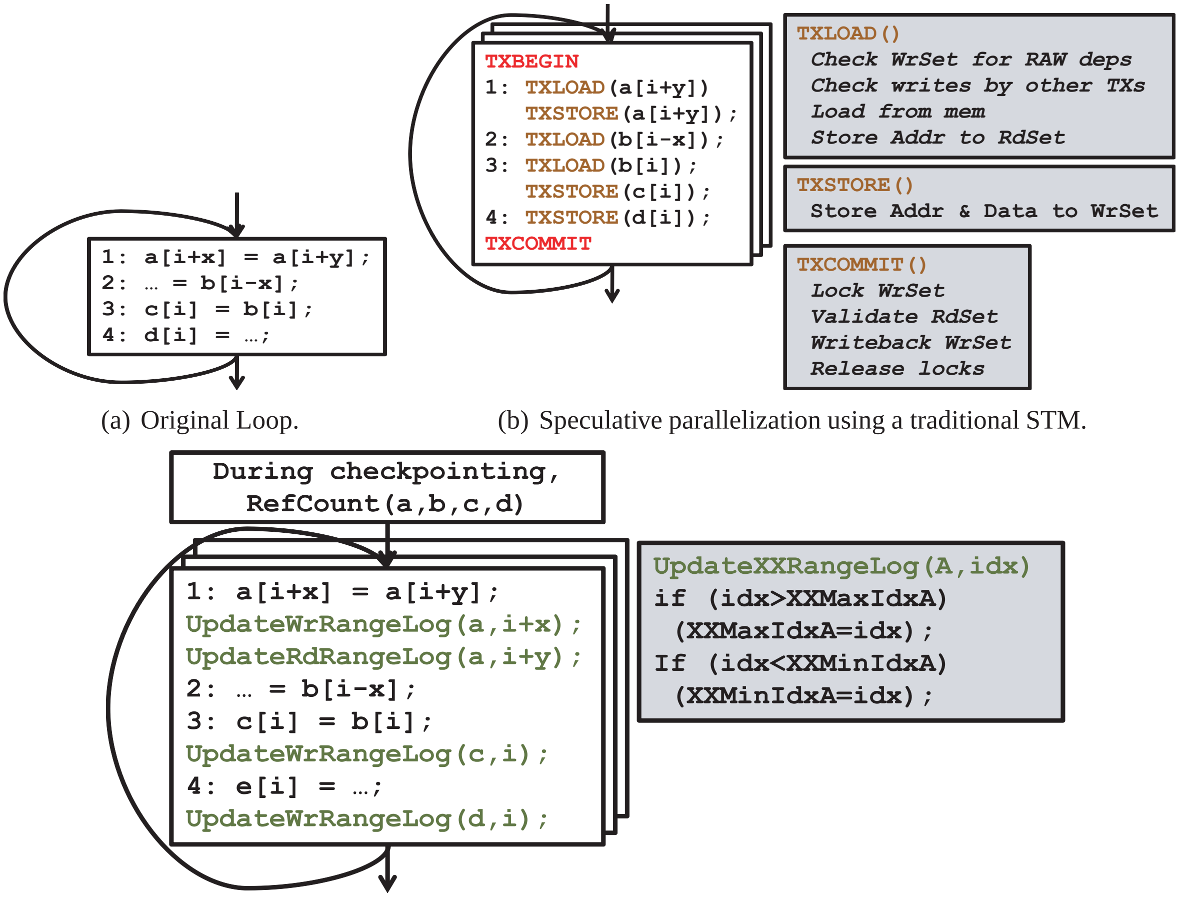
Another piece of code inserted after the last loop barrier is the code performing reduction variable expansion. Reduction variables (e.g., accumulators and min/max values) normally cause cross-iteration register dependences. The ParaScript framework resolves these dependences by creating a local accumulator or min/max value, and privately accumulating the totals for each individual trace. After all chunks are finished, local accumulators are summed up and global min/max values are determined amongst the local min/max values.

In this work, in order to minimize the checkpoining overhead, a single checkpoint (Section 3.4.4) is taken at the beginning of the parallelized loop, and if any of the conflict checks fail, the loop is executed sequentially from the beginning. Due to the high accuracy of our initial dependence assessment and stability of array access trends in the JavaScript programs we investigated (both ones with and without parallelism potential), this approach proved quite effective.

3.4 Ultra-lightweight Speculation Support

In this section, we introduce our approach for misspeculation detection, checkpointing and recovery. Furthermore, we describe two optimizations to further lower the checkpointing overhead. Figure 3.6(a) shows an example target loop for parallelization. Using a traditional speculation mechanism such as an STM for speculative parallelization (Figure 3.6(b)) requires the code generator to instrument all memory accesses with the corresponding speculative versions – `TXLOAD` and `TXSTORE` – and enclose loop chunks with `TXBEGIN` and `TXCOMMIT`. Replacing all memory accesses with transactional versions causes the runtime to spend substantial time tracking each individual memory access. Furthermore, at commit time, considerable overhead is incurred for locking target memory locations and performing consistency checks with reads. These overheads are the main reasons that STM systems usually incur up to 5x slow down over sequential versions of applications [28].

However, in ParaScript, the data flow analysis done during the JIT compilation, obviates the need for speculation on scalar values and individual object variables. Furthermore, a reference counting based analysis is performed during checkpointing on the object arrays, while



```
1: a[i+x] = a[i+y];
2: … = b[i-x];
3: c[i] = b[i];
4: d[i] = …;
```

(a) Original Loop.

```
TXBEGIN
1: TXLOAD(a[i+y])
   TXSTORE(a[i+y]);
2: TXLOAD(b[i-x]);
3: TXLOAD(b[i]);
   TXSTORE(c[i]);
4: TXSTORE(d[i]);
TXCOMMIT
```

```
TXLOAD()
 Check WrSet for RAW deps
 Check writes by other TXs
 Load from mem
 Store Addr to RdSet
```

```
TXSTORE()
 Store Addr & Data to WrSet
```

```
TXCOMMIT()
 Lock WrSet
 Validate RdSet
 Writeback WrSet
 Release locks
```

(b) Speculative parallelization using a traditional STM.

```
During checkpointing,
RefCount(a,b,c,d)
```

```
1: a[i+x] = a[i+y];
UpdateWrRangeLog(a,i+x);
UpdateRdRangeLog(a,i+y);
2: … = b[i-x];
3: c[i] = b[i];
UpdateWrRangeLog(c,i);
4: e[i] = …;
UpdateWrRangeLog(d,i);
```

```
UpdateXXRangeLog(A,idx)
if (idx>XXMaxIdxA)
 (XXMaxIdxA=idx);
If (idx<XXMinIdxA)
 (XXMinIdxA=idx);
```

(c) Speculative parallelization using ParaScript.

Figure 3.6: Extra steps needed at runtime for speculation support for loop parallelization in a traditional STM (TL2 [35]) and in ParaScript.

a minimal set of operations consisting mostly of array index comparisons is added to each array access to enable range-based dependence checking. These techniques are detailed in the following subsections.

3.4.1 Conflict Detection Using Reference Counting

As mentioned in Section 3.3.1, manipulating object arrays inside the loop potentially creates dependence cases that are not resolvable using simple data-flow analysis. For example, if the same object reference is assigned to two different array elements, since those elements might be written to during different iterations of the loop, the loop has potential cross iteration dependences, and can not be parallelized.

In order to identify these cases, we employ a technique similar to what reference counting garbage collectors use. At runtime, a header is added to all objects involved with loop arrays. This header, which is only used during array checkpointing, includes a reference count and entries for storing array IDs (e.g., starting address). The checkpointing mechanism already goes through all live-in array elements and stores them in the state checkpoint (Section 3.4.4). During this process which happens before starting parallel loop execution, if an array element refers to an object, the object's reference count and array ID entries are queried. If it had been referenced by the same array before, the system will know that there are two elements in the same array that refer to the same object, and thereby, the loop is disqualified from the parallelization process.

If the object has been referred to by another array and any elements of the two arrays are dependent through def-use chains, the loop is disqualified as well (this condition might disqualify loops that are actually parallelizable, but in order to eliminate the need for more in-depth and expensive speculation, this pessimistic assumption is made to ensure correct execution in all cases).

If, according to the header, the object has no array element references so far, the reference count is incremented and the array's starting address is added to array ID list in the object header. If all object arrays in the loop pass this reference counting phase, parallel loop execution is started and range-based checks as described next are applied to the arrays.

3.4.2 Conflict Detection Using Range-based Checks

A mechanism is needed to detect cross-iteration conflicts for scalar arrays and for object arrays that have passed the reference counting checks. In order to avoid instrumenting every array access with speculative operations, a range-based dependence checking is employed. In this method, during the speculative loop execution, each array maintains four local variables called `rdMinIndex`, `rdMaxIndex`, `wrMinIndex`, and `wrMaxIndex`. These values keep track of the minimum and maximum read and write indexes for the array. During parallel code generation, at each array read or write access, necessary comparisons are inserted in the bytecode to update these values if needed. After several iterations of the loop, accessed array references and their minimum and maximum access index values are written to the array write and read sets (`arrayWrSet` and `arrayRdSet`) in the memory. Each thread of execution maintains its own array read and write sets. The number of iterations after which write and read sets are updated is determined based on a runtime heuristic depending on the number of arrays in the loop, number of array accesses and the difference between minimum and maximum accessed index values. As was shown in Figure 3.5, after a predefined number of iterations, all threads stop at a barrier and the conflict checking routine checks the read and write access ranges of each array in each thread against write access ranges of the same array in all other threads. If any conflict is found, the state is rolled back to the previous checkpoint and the loop is run sequentially.

One downside of this mechanism for array conflict checking is that strided accesses to arrays are always detected as conflicts. However, this is a minor cost compared to the large overhead reduction as a result of efficient range-based conflict detection in other cases.

Function	Description	new Min	new Max
`pop()`	Remove last array element	`oldMin`	`oldArray.length()`
`push()`	Add elements to end of array	`oldMin`	`oldArray.length()+1`
`reverse()`	Reverse array elements	`0`	`oldArray.length()`
`shift()`	Remove first array element	`0`	`oldArray.length()`
`sort()`	Sorts array elements	`0`	`oldArray.length()`
`splice(i,n,e1,...,eN)`	Removes n and/or add N elements from position i	`min(oldMin,i)`	`max(oldMax,i+N,i+n)`
`unshift(e1,...,eN)`	Add N elements to beginning of array	`0`	`oldArray.length + N`

Table 3.1: JavaScript Array manipulation functions' effect on minimum and maximum access index values. `oldArray.length()` is the length of the array before applying the function. Likewise, `oldMin` and `oldMax` are old values of minimum and maximum read and write access index values.

3.4.3 Instrumenting Array Functions

As mentioned before, all array read and write accesses have to be instrumented for updating minimum and maximum access index values. In addition to normal array access using indexes, JavaScript provides a wide range of array manipulation functions which are implemented inside the JavaScript engine and read or write various points in arrays. In order to correctly capture their effect, their uses in the JavaScript application are also instrumented. Table 3.1 shows these functions and their effect on min/max access index values.

In JavaScript, it is possible to add custom functions to built-in object types such as `Array`, `String`, or `Object` using the `prototype` keyword inside the program source code. These custom functions are able to change values or properties of their respective object. This could pose complications in determining the values of minimum and maximum access index values. Therefore, any additional properties of functions added by the `prototype` keyword are detected and instrumented to correctly update access index values.

3.4.4 Checkpointing and Recovery

At any point during runtime, the *global object* contains references to all application objects in the global namespace. Inside the JavaScript source code, the global object can always be accessed using the `this` keyword in the global scope and is equivalent to the `window` object in web pages. Pointers to local objects, variables and function arguments reside in the corresponding function's *call object*, and can be accessed from within the engine.

Checkpointing: When a checkpoint is needed in the global scope, only a global checkpoint is taken, whereas in case the checkpoint request is issued inside a function, both global and local checkpoints need to be taken. The stack and frame pointers are also checkpointed in the latter case. While going through variable references in the global or local namespace, they are cloned and stored in the heap. Since there will be no references to these checkpoints from within the JavaScript application, the garbage collector needs to be asked explicitly not to touch them until the next checkpoint is taken. During checkpointing, to ensure proper roll-back, objects are deep-copied. However, based on a runtime threshold, ParaScript stops the deep-copying process and avoids parallelizing the loop if checkpointing becomes too expensive. Although a function's source code can be changed at runtime in JavaScript, since ParaScript avoids parallelizing loops containing code injection constructs, the checkpointing mechanism does not need to clone the source code. All other function properties are cloned.

Recovery: In case of a rollback request, the checkpoint is used to revert back to the original state and the execution starts from the original checkpointing location. Similar to checkpointing, recovery is also done at two levels of global and local namespaces. Due to complications of handling different stack frames at runtime, cross-function speculation is not supported. This means that speculative code segments should start and end in the global scope or the same function scope (obviously, different functions could be called and returned in between). During local recovery, stack and frame pointers are over-written by the checkpointed values. This makes sure that when an exception happens in the speculative segment and the exception handler is called from within some child functions, if a rollback is triggered, the stack and frame pointers have the correct values after recovery.

3.4.5 Checkpointing Optimizations

Using the speculation mechanism inside the ParaScript framework provides several optimization opportunities to further reduce the checkpointing overhead.

Selective variable cloning: The original checkpointing mechanism takes checkpoints of the whole global or local state at any given time. However, in ParaScript, only a checkpoint of variables written during speculative code segment execution is needed. In the selective variable cloning optimization, these written variables are identified using the data flow information. If any variable is passed as an argument to a function, the function's data flow information is used to track down the variable accesses and determine if there are any writes to that variable. This information is passed to the checkpointing mechanism, and a selective rather than a full variable cloning is performed.

Array clone elimination: In some applications, there are large arrays holding return values from calls to functions implemented inside the browser. One example of these functions is `getImageData` from the `canvas` HTML5 element implementation inside the browser. This function returns some information about the image inside the `canvas` element. Since speculative code segments in ParaScript do not change anything inside DOM, the original image remains intact during the speculative execution. Therefore, instead of cloning the array containing the image data information, ParaScript calls the original function again with the same input at rollback time (e.g., `getImageData` is called again on the same image).

3.5 Evaluation

3.5.1 Experimental Setup

ParaScript is implemented in Mozilla Firefox's JavaScript engine, SpiderMonkey [12] distributed with Firefox 3.7a1pre. All experiments are done with the tracing optimization [40] enabled. In SpiderMonkey, at a high-level, each engine instance has a *Runtime* object, which is mainly responsible for memory management and handling global data. Each JavaScript program in the browser has only one *Runtime* object, in which references to all other objects in the program reside. Objects cannot move or be shared between different *Runtime*s. *Context* objects, on the other hand, are per-thread objects that are responsible for tasks such as exception handling. Each *Runtime* can have multiple *Context*s and other objects can be shared among different *Context*s. Multithreading support in SpiderMonkey is made possible by the serialization of several data structures belonging to the *Runtime* object and special handling of garbage collection and object property accesses [12].

The proposed techniques in this work are evaluated on the SunSpider [13] benchmark suite and a set of filters from the Pixastic JavaScript Image processing Library [11].

SunSpider has 26 JavaScript programs. All these were run through the ParaScript engine and 11 were found to benefit from parallelization. Lack of parallelization opportunities in the rest of them, however, was found early on in the dependence assessment stage (Section 3.3) without any noticeable performance overhead (an average of 2% slow down due to the initial analysis across the 15 excluded benchmarks). From these 11 parallelizable benchmarks, three `bitops` benchmarks do not perform useful tasks and are only in the suite to benchmark bitwise operations performance. Therefore, they are also excluded from the final results (These 3 benchmarks gained an average of 2.93x speedup on 8 threads after running through ParaScript).

The Pixastic library has 28 filters and effects, out of which the 11 most compute-intensive filters were selected. The rest of the filters perform simple tasks such as inverting colors, so their loops did not even pass the initial loop selection heuristic based on the loop size. Out of the 11 selected filters, 3 benchmarks (namely `blur`, `mosaic` and `pointilize`) were detected by ParaScript to be not parallelizable, due to cross iteration array dependence between all iterations in `blur`, and DOM accesses inside loops in `mosaic` and `pointilize`. All benchmarks were run 10 times, and the average execution time is reported. The target platform is an 8-processor system with 2 Intel Xeon Quad-core processors, running Ubuntu 9.10.

3.5.2 Parallelism Potential and Speculation Cost

Figure 3.7 shows the fraction of sequential execution time selected by ParaScript for parallelization in our subset of the SunSpider and Pixastic suites. These ratios show the upper bound on parallelization benefits. In the image processing benchmarks, a high fraction of sequential execution outside parallelized loops is spent in the `getImageData` function implementation in-

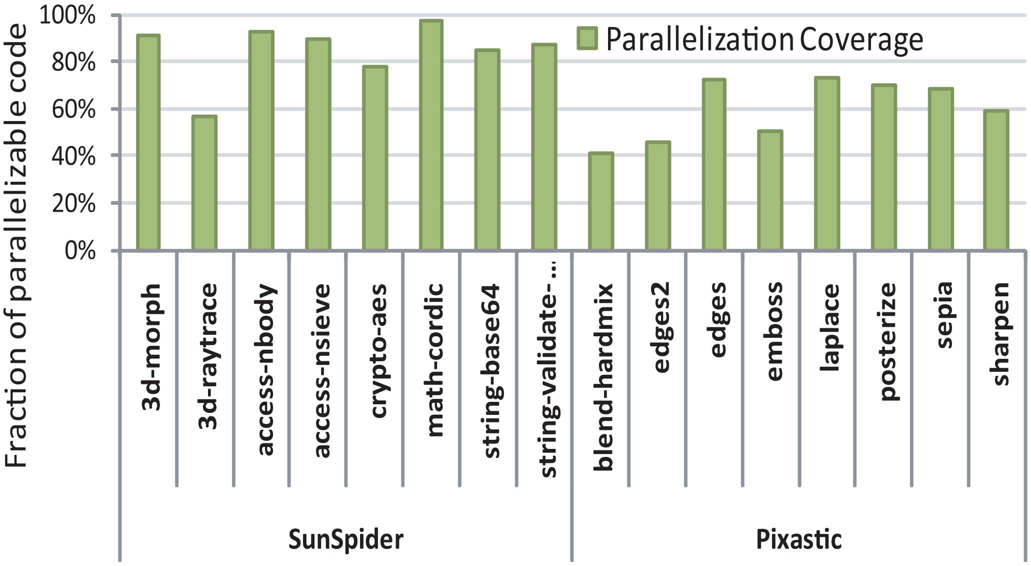


Figure 3.7: Parallelization coverage as ratio of parallelizable code runtime to total sequential program execution.

side the browser. This function takes the image inside an HTML `canvas` element as the input and returns a 2D array containing the RGB and alpha value information for all pixels in the image. The function's implementation inside the Firefox browser mainly consists of a DOALL loop that walks over all pixels in the image. We exploited this parallelization opportunity and hand-parallelized this DOALL loop inside the browser. The effect of this parallelization is discussed in the next subsection.

Checkpointing and speculation overheads are presented in Figure 3.8. This study has been done on single-threaded versions of the applications. The experiments are set up such that the sequential program is passed through the ParaScript system and parallelization and speculation constructs required for proper execution of the parallelized version are inserted into the sequential benchmark. The benchmark is then executed with one thread. The overhead is measured after applying both optimizations in Section 3.4.5. Array clone elimination proved to be quite effective in Pixastic benchmarks, due to cloning elimination in the return array of the

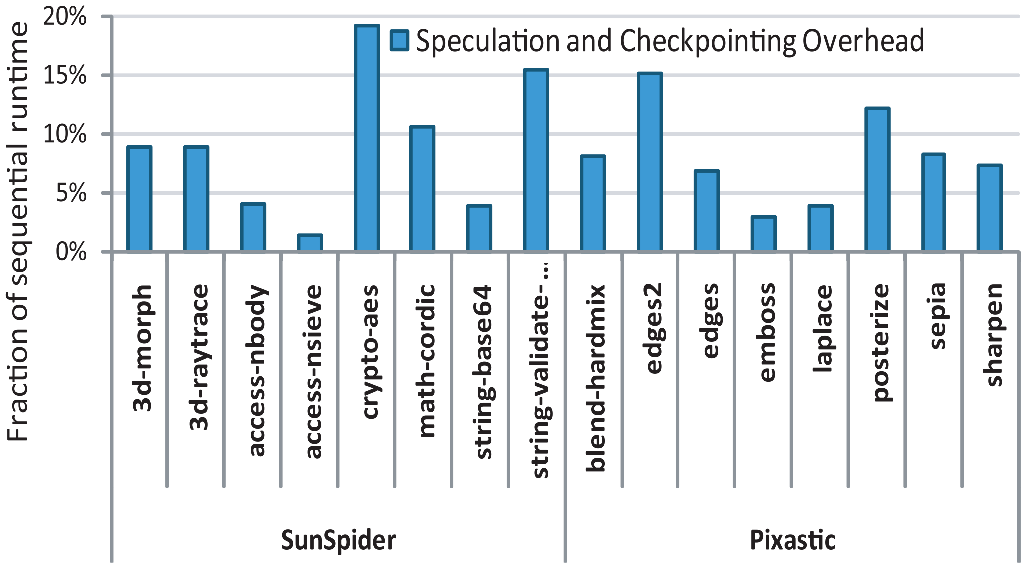
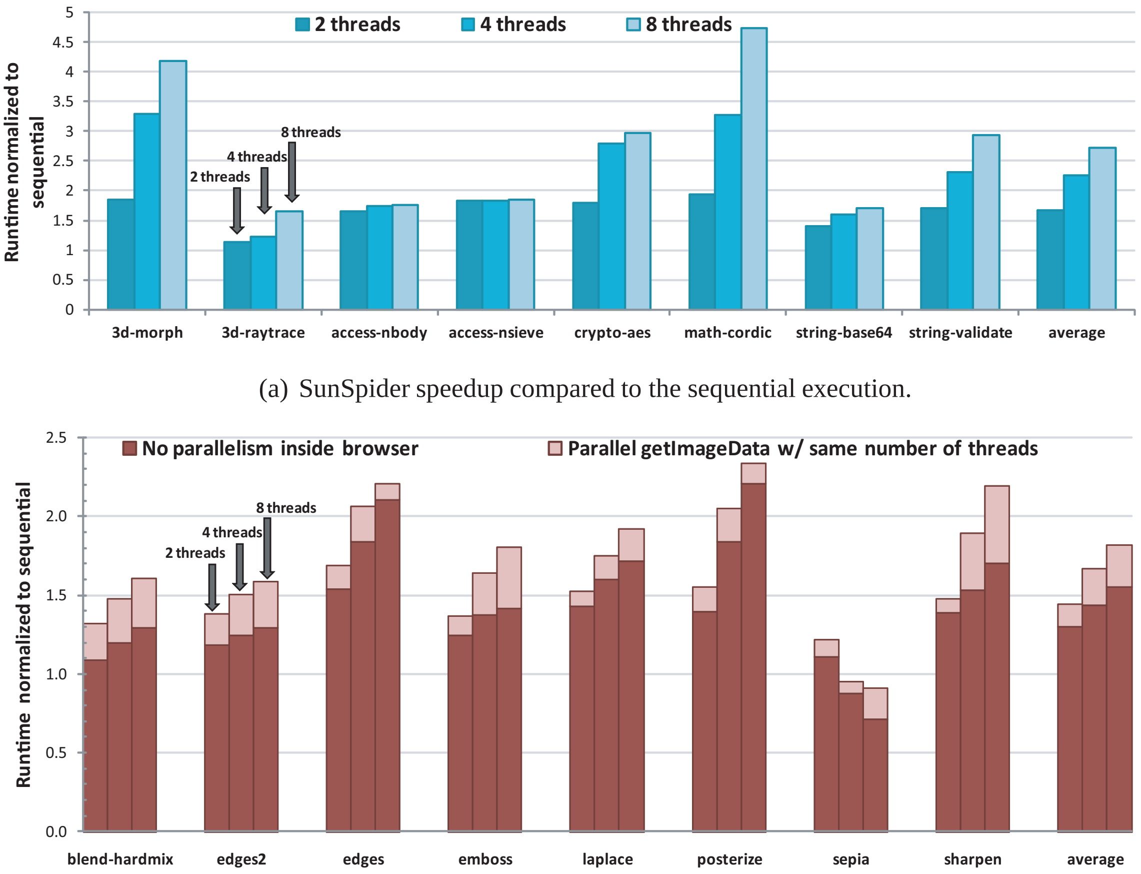


Figure 3.8: Speculation and checkpointing overhead as a fraction of total sequential program execution.

getImageData function. The only overhead missing in these bars is the checking overhead at the end of each parallel chunk. Due to the efficient storage of accessed array ranges, this checking overhead turned out to be negligible in the experiments. On average, checkpointing and speculation overhead is around 17% of the sequential execution runtime of the whole application. The main reasons for this low overhead are accurate dependence prediction, light-weight array access instrumentation, efficient conflict checking mechanism due to the use of array access bound checks instead of individual element checks, and the proposed optimizations on checkpointing. Furthermore, due to the high prediction success rate of the parallelization assessment step, ParaScript is able to take checkpoints only once at the beginning of the loop, which in turn lowers the overhead.

3.5.3 Results

Figure 3.9 shows performance improvements as a result of speculative parallelization using the ParaScript framework, across subsets of SunSpider suite and the Pixastic image processing



(a) SunSpider speedup compared to the sequential execution.

(b) Pixastic speedup compared to the sequential execution. The top part of each graph shows the extra performance gain obtained by parallelizing the `getImageData` implementation inside Firefox.

Figure 3.9: ParaScript performance improvement over sequential for 2, 4 and 8 threads. Black lines on top of the graphs show the performance of a system with hardware support for speculation (e.g., HTM).

applications. The Y-axis shows the speedup versus sequential. These performance numbers include all overheads from the JIT compilation time analysis, runtime analysis and runtime speculation. The black line on top of each bar presents the effect of replacing our light-weight speculation mechanism with hardware speculation support such as a hardware transactional memory system. Speculation costs of such a hardware system is assumed to be zero.

Overall, across the subset of SunSpider benchmarks, the experiments show and average of 1.51x, 2.14x and 2.55x improvement over sequential for 2, 4, and 8 threads of execution (Figure 3.9(a))[1]. In case of using hardware speculation support, the speedups would have increased to an average of 1.66x, 2.27x and 2.72x over sequential for 2, 4, and 8 threads.

Figure 3.9(b) presents the results for Pixastic applications. The lower part of each bar shows the performance improvement as a result of speculatively parallelizing the application itself. The top part of each bar is the extra performance improvement gained by parallelizing the `getImageData` function inside the browser with the same number of threads as the parallelized application. Finally, the black line on top of each bar shows the performance improvement, assuming hardware speculation support. On average, speedups of 1.29x, 1.43x, and 1.55x are gained after parallelization for 2, 4, and 8 threads of execution. Additional speedups of 14%, 23%, and 27% are gained after parallelizing the Firefox implementation of the `getImageData` function which increases the performance improvement to an average of 1.43x, 1.66x, and 1.82x respectively. Using hardware speculation support, speedups go upto 1.58x, 1.79x and 1.92x for 2, 4 and 8 threads.

3.5.4 Discussion

In SunSpider benchmarks (Figure 3.9(a)), `access-nbody`, `access-nsieve` and `string-base64` do not show much scaling past 2 threads. The reason is that the parallelized loops in these benchmarks are unbalanced and the final runtimes are always bounded by the longest running iteration, irrespective of the number of threads used for parallelization. Comparatively low overall speedup in `raytrace` (1.65x for 8 threads) is due to the low coverage (less than 60%) of parallelizable loops.

The loop selection mechanism discovered many loops with cross-iteration dependences in

[1] Average speedup for all 26 SunSpider benchmarks is 1.69x for 8 threads.

these programs and avoided parallelizing them, so the abort rate was zero in almost all the benchmarks. The only case of abort was from an inner loop in 3d-raytrace, which contributed to less than 5% of the total execution time.

In Pixastic applications, other than the inherent parallelization limitation due to relatively low coverage of parallelizable sections (an average of 60% coverage), the large ratio of memory to computation operations in these benchmarks is a limiting factor to their parallelism potential. This causes the program to become limited by the memory system latency rather than the amount of computation.

The negative effect of this factor has the most impact on the `sepia` benchmark where it causes slowdown rather than speedup for 4 and 8 threads. Each iteration in the inner loop of this benchmark has 3 multiply operations, 7 adds, and 6 memory operations. This causes the program to be quite memory intensive and therefore be bound by the performance of the memory system. Although the computations and memory requests are parallelized in the 4 and 8 threaded versions, the program is sequentialized at the memory system bus, waiting for its memory requests to be serviced and returned. Based on this observation, the loop selection heuristic was updated to account for the memory to computation operation ratio when making parallelization decision to avoid such slowdowns.

The same effect limits the performance improvements from parallelizing the `getImageData` function inside the browser. Although, this function can be parallelized without any speculation, the innermost loop has 8 memory operations versus 8 add, 3 multiply and 3 division operations. Therefore, becoming sequential at the memory system bus has negative impacts on the parallelization gains of this function as well.

3.6 Related Work

There is a long history of work in the area of parallelizing sequential C/C++ applications. Hydra [45] and Stampede [84] are two of the first efforts in the area of general purpose program parallelization. The POSH compiler [53] uses loop-level parallelization with TLS hardware support. The authors in [93] proposed compiler transformations to extract more loop level parallelism from sequential programs. Speculative decoupled software pipelining [87] is another approach that focuses on extracting parallelism from loops with cross iteration dependences. In that work, they distribute a single iteration of the loop over several cores. All these works use static analysis and profiling for parallelization and assume hardware support for speculation, while in this work, we do not rely on any static memory analysis and we perform the speculation completely in software.

There have been previous proposals for dynamic parallelization techniques. The JPRM system [33] focuses on dynamic parallelization of Java applications, but it is dependent on complex and costly hardware support for online memory profiling that does not exist in commodity processor systems. The LRPD test [72], performs speculative dynamic parallelization of Fortran applications. This framework is dependent on static analysis and transformation of loops, and runtime checkpointing and speculative execution. In this work, no static analysis is done on the code and the code is generated on the fly. Furthermore, LRPD's array dependence testing is based on tracking individual array accesses, while ParaScript only tracks array bounds which significantly reduces tracking and checking overheads.

The work in [24] employs range tests based on static symbolic analysis, whereas ParaScript performs these tests dynamically and at a lower cost at runtime. The Ninja project [21] uses array reference comparison for linear arrays, which does not cover arrays of objects or arrays of arrays. Their tests on higher order arrays involves introducing a custom array package and changing the source code.

The work in [19] proposes a C++ library which is dependent on the programmer identification of parallel regions and performs dynamic orchestration of these regions. Being dependent on the programmers, they do not have any speculation support. Authors in [91] and [89] investigate dynamic parallelization of binary executables. They use binary rewritting to transform sequential to parallel code. However, [89] while exploiting slice-based parallelization, assumes hardware speculation and hardware parallel slice support neither of which exist in current systems. The work in [91] only performs control speculation and does not speculate on data. Therefore, after identifying the parallel loop, they have to perform data dependence analysis to prove lack of cross iteration memory dependence, which would be quite costly (Section 3.2). Furthermore, none of these dynamic parallelization techniques work on dynamic languages such as JavaScript.

There is a significant amount of previous efforts in the area of memory speculation. Harris *et al.* go through a detailed survey of different transactional memory techniques in [48]. In particular, Shavit *et al.* proposed the first implementation of software transactional memory in [77]. Several other works such as DSTM [50] and OSTM [47] proposed non-blocking STM implementations. The authors in [49, 18] proposed a lock-based approach where write locks are acquired when an address is written. Also, they maintain a read set which needs to be validated before commit. Our speculation mechanism is not a full-featured software transactional memory. ParaScript introduces a customized speculation system which is tailored towards efficient speculative loop level parallelism in JavaScript. By excluding many features required by general-purpose TM models, speculation in ParaScript has become highly lean and efficient.

Due to the lack of concurrency support in the JavaScript language, there have not been many previous efforts for exploiting multicore systems to improve the execution performance of client-side web applications. There has been a recent proposal to add a limited form of concurrency called *web workers* to the language. There are already standardization efforts being undertaken on this proposal [16] and almost all major browsers are starting to support these constructs. However, due to separate memory spaces among the workers, the only means of communication between them is through message passing, which makes developing multithreaded applications very difficult. Crom [61], is a recent effort in employing speculative execution to accelerate web browsing. Crom runs speculative versions of event handlers based on user behavior speculation in a shadow context of the browser, and if the user generates a speculated-upon event, the precomputed result and the shadow context are committed to the main browser context. Crom exploits a different parallelization layer and is orthogonal to our work. One could use both approaches at the same time to enjoy parallelization benefits at multiple levels.

We implemented our framework on top of Mozilla's TraceMonkey, the trace-based JIT compiler described in [40] and released as a part of recent versions of Firefox [7]. TraceMonkey is able to achieve more than 10x speedup on some programs in the SunSpider suite compared to previous versions of SpiderMonkey on Firefox (which is an interpreter-only JavaScript engine). All this performance is achieved by intelligent type specialization and the tracing mechanism. Trace-based compilation approach is orthogonal to our techniques. As a matter of fact, all reported results in our work (including baseline sequential results), have been generated with the tracing support enabled inside our modified Firefox browser.

3.7 Summary

JavaScript is the dominant language in the client-side web application domain due to its flexibility, ease of prototyping, and portability. Furthermore, as larger and more applications are deployed on the web, more computation is moved to the client side to reduce network traffic and provide the user with a more responsive browsing experience. However, JavaScript engines fall short of providing required performance when it comes to large and compute-intensive applications. At the same time, multicore systems are the standard computing platform in the laptop and desktop markets and are making their way into cell phones. Therefore, in addition to the efforts underway by browser developers to improve engines' performance, parallel execution of JavaScript applications is inevitable to sustain required performance in the web application market. In this chapter, we proposed ParaScript, a fully dynamic parallelizing engine along with a highly customized software speculation engine to automatically exploit speculative loop level parallelism in client-side web applications. We show that our technique can efficiently identify and exploit implicit parallelism in JavaScript applications. The prototype parallelization system achieves an average of 2.55x speedup on a subset of the SunSpider benchmark suite and 1.82x speedup a set of image processing filters, using 8 threads on a commodity multicore system.

CHAPTER 4

Dynamically Accelerating Client-side Web Applications through Decoupled Execution

4.1 Introduction

TraceMonkey, a trace-based JavaScript engine, was developed for the Firefox web browser to remove some of the inefficiencies associated with dynamic typing [40]. TraceMonkey identifies hot bytecode sequences and compiles them to native machine code with statically assumed types. As long as these sequences (traces) remain type-stable, execution remains in the type-specialized machine code. TraceMonkey works at the granularity of individual loops, and therefore, is very well suited for compute-intensive web applications.

While compiling hot traces to the native code, TraceMonkey inserts runtime checks, called *guard* instructions, into the trace to check for type, control flow, and other assumptions made during the just-in-time compilation process. These checks are heavily biased not to fire as the vast majority of the time, types do not vary and a single control flow path is dominant [40]. However, these guards comprise a significant fraction of total executed instructions. Figure 4.1

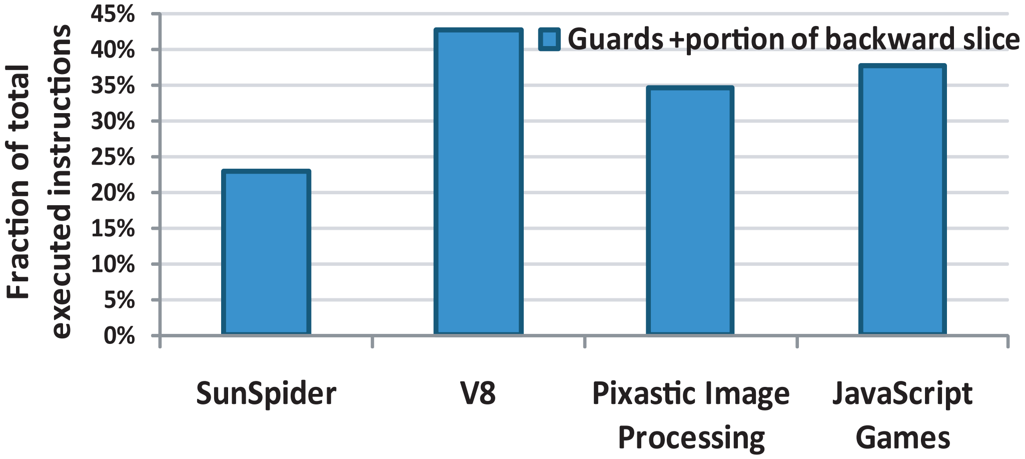


Figure 4.1: Fraction of total instructions devoted to computing guards across four groups of benchmarks: SunSpider, V8, Pixastic image processing applications, and a set of JavaScript games. These bars include guards and portion of the backward slice only needed by guards and not used elsewhere.

presents the overhead of guards consisting of the guard instructions themselves as well as the dependent computation used by the them. These are the instructions only used by guards and are not needed elsewhere in the trace. The average overhead is presented for four groups of applications: SunSpider [13] and V8 [15] benchmark suites, and two sets of applications from the image processing and gaming domains (more details on the benchmarks are provided in Section 4.5). These values range from a low of 22% to a high of 42%, which represent a significant runtime penalty.

In this work, we focus on reducing this overhead using a multi-threaded dynamically decoupled execution framework called *ParaGuard*. We decompose traces generated by TraceMonkey into two concurrent threads. The main thread consists of the code to implement the bulk of the user program, while the *ParaGuard* thread performs most of the runtime checks. With this model, the main thread speculatively executes ahead assuming that the checks will not fire and the common execution scenario will proceed. When a check does fail, it reverts back to the interpreter and safely discards the improper speculative work. During speculative execution, the program is sandboxed to make sure no catastrophic execution failures happen until ParaGuard checks have been validated. In multicore systems with under-utilized cores, we can execute the main and guard threads concurrently to increase performance.

The contributions offered by this work are as follows:

- We propose ParaGuard, a method to dynamically decompose a type-specialized trace into two concurrent threads: the first speculatively performs the core computation along the expected path of control and the second verifies that the assumptions used to create the trace are valid.
- We introduce several optimizations including guard branch aggregation and profile-based snapshot elimination to increase the efficiency of the decoupled execution.

4.2 Background

In statically typed languages such as C or C++, the compiler can generate efficient machine code based on the type information provided by the programmer. However, in dynamically typed languages such as JavaScript, variable types can change at runtime and therefore, the compiler cannot generate machine code specialized for only one specific type. This forces the compiler to generate generalized machine code with the ability to handle potential dynamic type changes, causing the code to be considerably slower than the statically typed machine code. Some static compile-time type inference techniques can be applied to dynamically typed languages, but such techniques are far too slow for a language like JavaScript that needs to be loaded and compiled quickly in the web browser.

There have been a number of efforts to efficiently compile and execute JavaScript applications on different browsers. One of the most recent proposals is TraceMonkey [40] by Mozilla which is implemented on top of SpiderMonkey [12] and is now integrated in their web browser, Firefox [7].

TraceMonkey uses a trace-based compilation method that reduces JavaScript execution time by exploiting high performance type-specialized machine code when possible. It starts off by running the JavaScript application in a bytecode interpreter and at the same time identifies and records hot bytecode execution sequences. These sequences, called traces, are then compiled to native code. In TraceMonkey, traces are formed out of individual hot loops. This choice is based on the assumption that hot loops are mostly type-stable, thereby allowing most of the program execution to be expressed by type-specialized and natively compiled traces.

Each compiled trace consists of a single path in the program with a specific value-type mapping. However, this type-mapping is not guaranteed to be always correct, because different code paths may be taken or different types may be assigned to a value in subsequent loop iterations. Therefore, executing the same trace for later loop iterations is based on the speculation that the path and types will match what was observed during recording. These speculations are verified using a number of checks (called *guards*) along the trace. The guards are inserted wherever there is a need to check for alternate typing, control flow paths or other runtime checks (as described in the beginning of Section 4.3). If these checks fail, the trace exits and reverts back to interpreting the bytecode. Likewise, if the exit becomes hot, a branch trace is generated and compiled to cover the new path. In this way, a trace tree is eventually formed which covers all hot paths in the loop.

Figure 4.2 describes the major phases of JavaScript execution in TraceMonkey. These

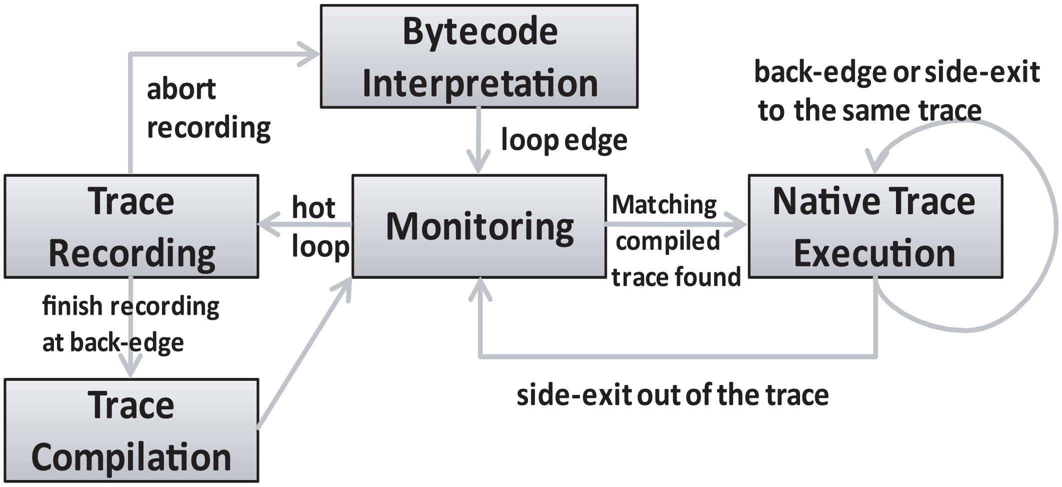

Bytecode Interpretation
abort recording
loop edge
back-edge or side-exit to the same trace
Trace Recording
hot loop
Monitoring
Matching compiled trace found
Native Trace Execution
finish recording at back-edge
Trace Compilation
side-exit out of the trace

Figure 4.2: JavaScript tracing and type specialization in TraceMonkey. This state machine describes how the trace monitor manages trace-based just-in-time compilation.

phases happen in the *trace monitor* which coordinates the whole tracing process. Initially, the program starts in the bytecode interpreter, and when the interpreter reaches a loop edge, the trace monitor is called to determine whether a new trace should be recorded or an existing native trace could be executed for the loop. At the start of execution, since there are no compiled traces, the trace monitor simply profiles the number of loop edge crossings and enters the recording state after a loop becomes hot. During recording, the code along the trace is recorded in a low-level intermediate representation (LIR) which encodes all the operations and types in the trace. The LIR also contains guards to ensure that the control flow and types are identical to what was observed during recording. If the recorder is unable to continue recording, for example when faced with `eval` calls or reaching the trace length limits in a small-memory device, it chooses to abort the recording. On such an abort, the monitor discards the recorder and and returns to the monitoring state. The monitor also keeps track of how many times the recording has failed for a trace starting at each program counter (PC) value. Therefore, if a particular PC causes too many aborted recordings, the monitor *blacklists* the PC and will not attempt to record it again.

The recording is finished when execution reaches the loop header or exits the loop. Subsequently, the trace is compiled to the native code based on the types and control path of the recorded trace. From then on, whenever the monitor interprets a backward jump to a PC with a matching compiled trace (with the same type map), it enters native execution mode. In this mode, before calling the native trace, the monitor allocates a trace activation record containing imported local and global variables, temporary stack space, and space for arguments to native calls.

The monitor then calls the trace native code with the activation record as an argument. The native code returns with a pointer to a structure containing information about how the trace exited. Based on this information, the monitor restores interpreter state by copying back the imported variables from the trace activation record. The monitor behaves differently afterwards, based on the success of the trace return. If the trace exits unsuccessfully (e.g., due to having garbage collection triggered, running out of native stack, or noticing other abnormal conditions), the monitor returns to the monitoring state. However, if the trace exits successfully (e.g., due to running out of native code or hitting a branch condition for which no native code exists yet), the monitor checks whether the side exit PC has become hot or not. If not, it just keeps monitoring the interpretation to find other hot traces. If it has become hot, the monitor moves on to the recording state immediately, starting a new branch trace from that point and patching the side exit to jump directly to that branch. Using this approach, a single trace expands to a multiple-exit trace which could span a fairly large portion of the frequent execution graph.

In practice, loops are typically entered with only a few different combinations of variable types. Therefore, a small number of traces per loop is sufficient to run a program efficiently. TraceMonkey is able to achieve speedups of 2x to 20x on programs for which tracing is feasible [40].

4.3 ParaGuard: Concurrent Guard Execution

During LIR generation, the following categories of guards can be inserted into the trace.

Loop guards: They are inserted at the end of the loop and check for the loop termination condition. In general, loop traces end with an unconditional branch to the top of the loop and this guard ensures that the execution exits the loop when the correct condition is met.

Branch and case guards: When the LIR corresponding to a trace is generated, conditional branches and case statements are first replaced with unconditional ones, taking the same path that had been taken during trace recording. Guard instructions are then inserted to actually check the branch/case conditions and abort the trace if a different path needs to be taken.

Condition mismatch guards: These guards are inserted to terminate trace execution in case a condition, relied upon at recording time, no longer holds. In some of these situations, the alternate path of execution is so rare or difficult to handle in the native code, that it is preferable to have it interpreted rather than traced and compiled. One example is a negative array index access which requires string-based property lookups, compared to a positive index access which is merely a simple memory access. Type mismatch guards are also included in this category, and they check if the actual type during native execution matches with what was observed during

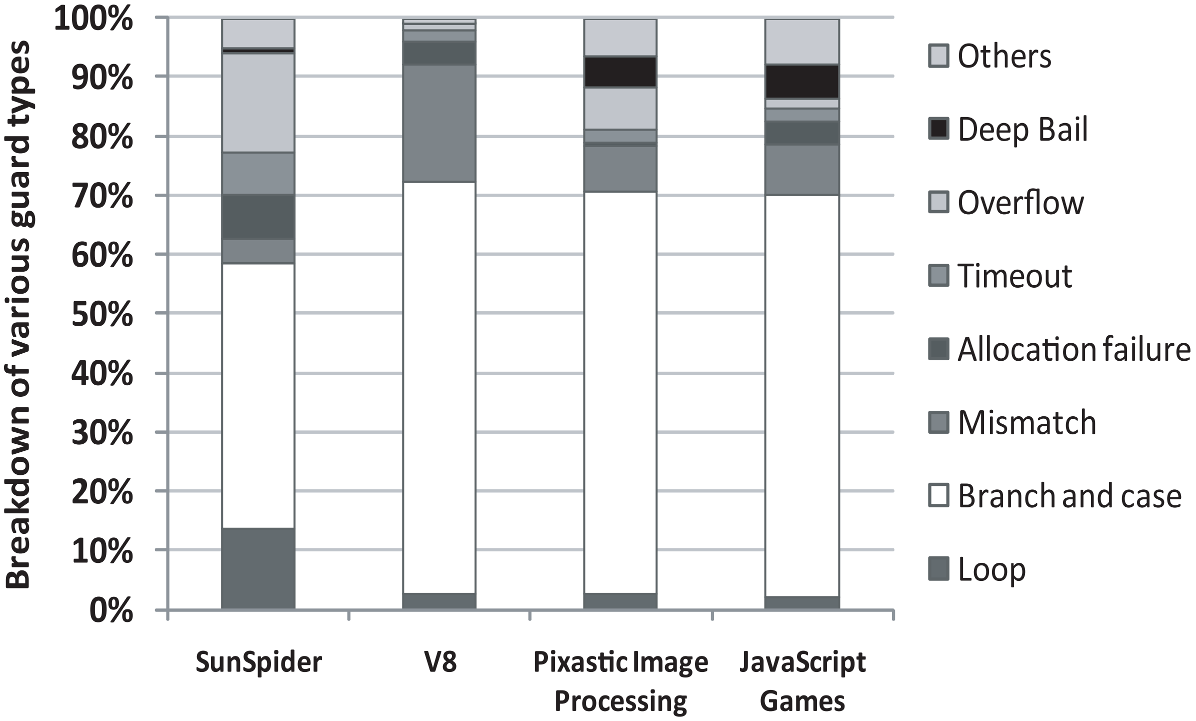


Figure 4.3: Breakdown of different types of guards in SunSpider, V8, Pixastic image processing and JavaScript games.

recording.

Miscellaneous guards: There are several other categories of guards such as allocation failure, execution timeout, variable overflow, and deep bail guards. Deep bail guards are triggered when during the execution of a native C function call in the trace, a trace exit is triggered.

Figure 4.3 shows the average relative ratio of different guard types in SunSpider and V8 suites, and our suite of image processing programs and JavaScript games. Miscellaneous guards comprise the top five sections in each bar. As can be seen, branch guards are the most frequently generated guards across all benchmarks. Condition mismatch, loop and overflow guards are other common ones.

In the *ParaGuard* technique (Figure 4.4), the majority of guards are moved to another trace

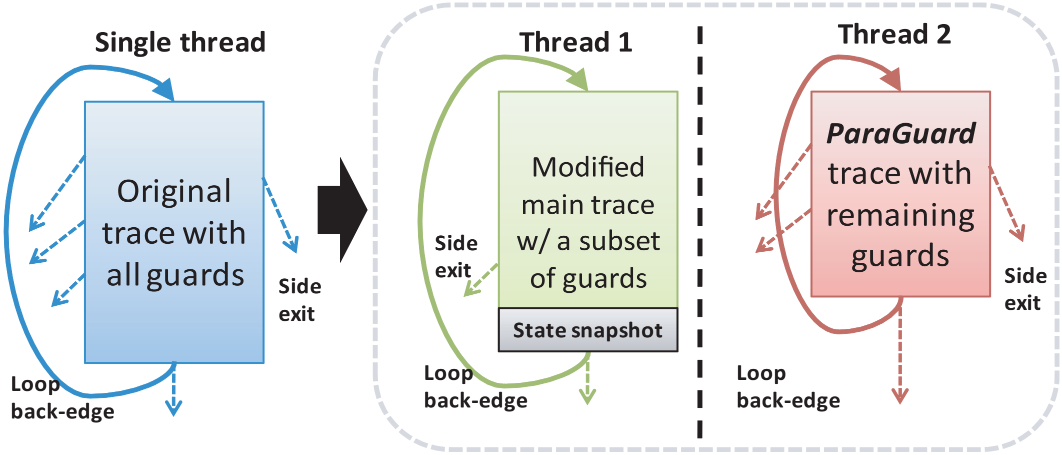


Figure 4.4: Offloading guard execution to the ParaGuard thread.

(ParaGuard trace) and are executed in a separate thread (ParaGuard thread), in parallel to the main trace. ParaGuard trace code is generated along with the main trace and is invoked at the same time during trace monitoring. The following subsections describe how we generate ParaGuards and restore the correct state of the interpreter after a ParaGuard is triggered and the trace is aborted. In Section 4.4, two optimizations are introduced to further improve the performance of our technique.

4.3.1 ParaGuard Generation

The optimizations in TraceMonkey are performed in two pipelined phases over the trace. During trace recording, immediately after the recorder emits an LIR instruction, the instruction is sent through the forward optimization pipeline. This forward pass consists of several optimizations including common subexpression elimination and expression simplifications such as constant folding. The second phase is a backward pass which goes through the whole trace from bottom to top after trace recording is complete. The optimizations in this pass include dead code elimination and dead data-stack and call-stack store elimination. After an LIR instruction passes the last stage in the optimization pipeline, the code generator emits the corresponding machine instructions.

Traditional guards are generated and inserted in the LIR during the forward pass. However, since we want to move guard instructions along with the LIR instructions that they depend on (their backward slice), we need to generate ParaGuards as an extra pipeline stage after all optimizations in the backward pass. We call this stage, *guard promotion*. The goal of guard promotion is to identify LIR instructions (guards and non-guards) that can be moved to the ParaGuard trace. A non-guard instruction is moved to the ParaGuard trace if it is only used for computing the inputs of a relocated guard. Furthermore, some instructions are marked for duplication in the ParaGuard trace, since they need to be re-executed there to minimize communication between the ParaGuard and main threads. During guard promotion, two groups of instructions are constructed. The first category is “to-be-copied” which contains the instructions duplicated on both the main and ParaGuard traces. The second group, called “to-be-moved”, consists of all instructions that are moved from the main trace to the ParaGuard trace by the end of the guard promotion pass. This pass is performed in two steps:

Step 1: This is essentially a partial implementation of backward slicing. Starting from each guard instruction in the trace, the compiler keeps track of *def* instructions for the guard’s source operands. Likewise, it tracks *def*s of the source operands of those *def* instructions. This procedure is continued recursively, traversing *def/use* chains and marking *def*s as “to-be-copied”. The destinations of these *def* instructions are also kept in a list for use in the second step. To avoid violating memory consistency between the main and ParaGuard thread, tracking *def*s is stopped after reaching a load instruction. Because if the load is copied or moved to the ParaGuard trace, the code needs to ensure that the load in the ParaGuard thread is not executed before the corresponding store in the main thread. Enforcing this requires adding locking primitives, which can cause high overheads.

Step 2: The goal of this step is to remove the *def*s that are only used in the guard's backward slice from the main trace. First the candidate guard for moving is marked as "to-be-moved". As the trace is traversed backwards, all *use*s of the candidate guard's source operands are recursively kept in a "use-set". When a *def* marked as "to-be-copied" (during step 1) is reached, its "use-set" is checked to see whether all its members are marked as "to-be-moved". If so, it is clear that this *def* is not going to be used in the main trace before the guard instruction, if all "to-be-moved" instructions are moved to the ParaGuard trace. Furthermore, a *def*'s destination liveness after the guard instruction should also be checked. In order to do that, the live set at the guard instruction is used and if the *def*'s destination operand is not a member of this live set, the *def*'s category can safely be changed from "to-be-copied" to "to-be-moved". These live sets are already generated prior to the guard promotion pass. To summarize, a *def* must meet three conditions to qualify for relocation to the ParaGuard trace:

1. It is marked as "to-be-copied".
2. All its uses before the guard are marked as "to-be-moved".
3. Its destination is not live after the guard instruction.

In addition to this analysis, guard promotion uses a heuristic that rejects promotion of the guard instructions whose backward slice is either very small or should be mostly copied to the ParaGuard trace rather than moved. Therefore, by the end of the guard promotion pass, some guards still remain in the main trace.

```
var myArray = new Array();
function init() {
  var j = 0;
  for (j = 0; j < 200; ++j)
    myArray[j] = j*2;
}
```

Figure 4.5: Sample JavaScript source code.

At runtime, live-in values to the ParaGuard trace are copied to a per-guard single-reader/single writer buffer, similar to the buffers in [69], which is written by the main trace and read by the ParaGuard trace. Initializing these per-guard buffers is done in the ParaGuard thread and is off the critical path in the main trace. The initial sizes of these buffers are determined at compilation time and in case more space is needed at runtime, they are dynamically expanded. During native execution, ParaGuard trace can start or resume execution once these values are written in the buffers by the main trace.

Figure 4.5 shows an example JavaScript code snippet. TraceMonkey's LIR for this code can be seen in Figure 4.6. Backward slices for each guard are highlighted with a different gray shade. Instructions belonging to multiple backward slices are highlighted with the same shade as the earliest observed guard in the trace. For instance, the backward slice for guard instruction 30 consists of instructions 29, 24 and 2. Likewise, the backward slice for instruction 26 are instructions 25, 24 and 2, and for instruction 23 are instructions 22, 21, 10, 6 and 2. Instructions marked with (*) are "to-be-copied" and the ones with (+) are "to-be-moved" after performing the guard promotion algorithm on the guards. This algorithm decided not to move the guard at instruction number 23, since it would have only saved two instructions (22 and 23) on the main trace, while either js_Array_set had to be re-executed in the ParaGuard trace or its return value had to be copied to the ParaGuard trace buffer.

```
label1:
(*)1 :  cx = ldq state[16]        // load context pointer
   2 :  ld1 = ld sp[-8]           // load 'j' from stack
(+)3 :  ld2 = ld cx[0]            // load context object
(+)4 :  eq3 = eq ld2, 0           // check if context is valid
(+)5 :  xf eq3                    // side exit if it's not
(*)6 :  $globl0 =ldq state[848]   // load myArray pointer from
                                  // trace activation record
   7 :  stqi sp[0] = $globl0      // store myArray on stack
   8 :  sti sp[8] = ld1           // store j on the stack
   9 :  sti sp[24] = 2            // store 2 on the stack
(*)10:  mul1 = mul ld1, 2         // multiply j by 2
(+)11:  ov2 = ov mul1             // check overflow on mul op
(+)12:  xt4:  xt ov2              // side exit if mul overflows
(+)13:  eq2 = eq mul1, 0          // check if mul1 is zero
(+)14:  xt eq2                    // side exit if so
   15:  sti sp[16] = mul1         // store mul result on stack
(*)16:  ldq1 = ldq $globl0[8]     // load myArray class
(+)17:  qi1 = qiand ldq1,         //
      quad #FFFFFFFF:FFFFFFFC
(+)18:  cl = quad #0:803D20       //
(+)19:  arrayg = qeq qi1, cl      // check if class is an array
(+)20:  xf arrayg                 // side exit if not
   21:  returng=js_Array_set(     // set myArray element
      $globl0 ld1 mul1)
   22:  eq1 = eq returng, 0       // check js_Array_set return value
   23:  xt eq1                    // side exit if failed
(*)24:  add1 = add ld1, 1         // add 1 to j
(+)25:  ov1 = ov add1             // check add for overflow
(+)26:  xt ov1                    // side exit if overflows
   27:  sti sp[-8] = add1         // store add result on stack
   28:  sti sp[8] = 200           // store 200 on stack
(*)29:  lt1 = lt add1, 200        // check loop condition
(*)30:  xf lt1                    // exit trace if finished
   31:  sti sp[-8] = add1         // store add result on stack
(*)32:  j -> label1               // jump back to the top
```

Figure 4.6: Original TraceMonkey's Low-level IR for the source code in Figure 4.5. Instructions marked with (*) are to be copied and the ones with (+) are to be moved to the ParaGuard trace.

Finally, Figures 4.7 and 4.8 show the modified main trace and the generated ParaGuard trace after applying guard promotion respectively. The same gray shades have been applied to guard instructions' backward slices.

```
label1:
1  :  cx = ldq state[16]        // (*) load context pointer
2  :  ld1 = ld sp[-8]           // load ‘‘j’’ from stack
PG1:  st shared_buf[0] = ld1    // store ld1 in the shared_buff
6  :  $globl0 =ldq state[848]   // (*) load myArray pointer
                                // from trace activation record
7  :  stqi sp[0] = $globl0      // store myArray on stack
8  :  sti sp[8] = ld1           // store j on the stack
9  :  sti sp[24] = 2            // store 2 on the stack
10 :  mul1 = mul ld1, 2         // (*) multiply j by 2
15 :  sti sp[16] = mul1         // store mul result on stack
16 :  ldq1 = ldq $globl0[8]     // (*) load myArray class
21 :  returng=js_Array_set(     // set myArray element
    $globl0 ld1 mul1)
22 :  eq1 = eq returng, 0       // check js_Array_set return val
23 :  xt eq1                    // side exit if failed
24 :  add1 = add ld1, 1         // (*) add 1 to j
PG2:  count = add count, 1      // inc snapshot counter
27 :  sti sp[-8] = add1         // store add result on stack
28 :  sti sp[8] = 200           // store 200 on stack
29 :  lt1 = lt add1, 200        // (*) check loop condition
30 :  xf lt1                    // (*) exit trace if finished
PG3:  eq2 = eq count, N         // check snapshot condition
PG4:  jt eq2 -> label2          // jump if snapshot needed
31 :  sti sp[-8] = add1         // store add result on stack
32 :  j -> label1               // (*) jump back to the top
label2:
   barrier paraguard_finish
   take_snapshot()
   count = 0
   j -> label1
```

Figure 4.7: Main Trace LIR after guard promotion.

PG* instructions highlighted in black are added to these traces during guard promotion. PG1 copies ld1 to the shared buffer between the main and ParaGuard traces. PG5 is the barrier waiting for this value in the ParaGuard trace and PG6 is loading it from the shared buffer. As can be seen, guard promotion has moved 13 out of 32 instructions in the original trace, while only adding four instructions. Instructions PG2, PG3, PG4, PG7, PG8, and PG9 are used for taking the native state snapshot for interpreter state recovery as described in the next subsection.

```
label1:
1' :  cx = ldq state[16]        // (*) load context pointer
3  :  ld2 = ld cx[0]            // (+) load context object
4  :  eq3 = eq ld2, 0           // (+) check if context is valid
5  :  xf eq3                    // (+) side exit if it's not
6  :  $globl0 =ldq state[848]   // (*) load myArray pointer
PG5:  barrier shared_buf[0]     // wait for of shared_buf[0]
PG6:  ld1 = ld shared_buf[0]    // load ld1 from shared_buf[0]
10' :  mul1 = mul ld1, 2        // (*) multiply j by 2
11 :  ov2 = ov mul1             // (+) check overflow on mul op
12 :  xt4:  xt ov2             // (+) side exit if mul overflows
13 :  eq2 = eq mul1, 0          // (+) check if mul1 is zero
14 :  xt eq2                    // (+) side exit if so
16':  ldq1 = ldq $globl0[8]     // (*) load myArray class
17 :  qi1 = qiand ldq1,         // (+)
   quad #FFFFFFFF:FFFFFFFC
18 :  cl = quad #0:803D20       // (+)
19 :  arrayg = qeq qi1, cl      // (+) check if class is array
20 :  xf arrayg                 // (+) side exit if not
PG7:  count = add count, 1      // inc snapshot counter
24':  add1 = add ld1, 1         // (*) add 1 to j
25 :  ov1 = ov add1             // (+) check add for overflow
26 :  xt ov1                    // (+) side exit if overflows
29':  lt1 = lt add1, 200        // (*) check loop condition
30':  xf lt1                    // (*) exit trace if finished
PG8:  eq3 = eq count, N         // check snapshot condition
PG9:  jt eq3 -> label2          // jump if snapshot needed
32':  j -> label1               // (*) jump back to the top
label2:
   bdcast paraguard_finish
   j -> label1
```

Figure 4.8: ParaGuard trace LIR after guard promotion.

4.3.2 Recovering Interpreter State using Selective Snapshots

As mentioned in Section 4.2, before invoking a trace, the interpreter builds a trace activation record that consists of the temporary stack space, space for arguments to native calls, and all imported global and local variables. These global and local values are copied from i the interpreter state to the trace activation record and the trace is later called like a normal call-through-pointer in C. After a guard is triggered and the trace call returns, the interpreter state is restored by copying the imported global and local variables from the trace activation record back to the interpreter state.

When using ParaGuard, this process gets more complicated. Since the guards trigger asynchronously, the main thread may have corrupted its state by executing instructions past the original guard location and overwriting the correct state. Therefore, some form of checkpointing support is needed for imported native variables, so that when a guard triggers in the ParaGuard trace, execution can roll back to a previous snapshot of the correct execution state.

Traditional rollback support such as those in software transactional memory would incur a high performance overhead and is unacceptable here. Thus, instead of making a backup copy of memory locations on every memory write, we use a *bulk* snapshot mechanism in which the frequency of taking memory snapshots is reduced to every *N* iterations. The exact value of *N* is determined dynamically according to a runtime heuristic which is based on the loop's instruction count, total iteration count, and number of memory operations per iteration. When the execution on the main trace reaches the loop guard and the trip count is a multiple of *N*, it stops at a barrier, waiting for the ParaGuard thread to catch up. In most cases, there is no waiting, because the ParaGuard trace is shorter than the main trace. Subsequently, the main trace takes the state snapshot, after which it continues executing. Since TraceMonkey does not perform tracing if the code path contains I/O accesses, the snapshot taking mechanism does not have to deal with checkpointing I/O operations.

In order to further reduce the overhead of bulk snapshots, a *selective* snapshot is taken which only includes critical memory locations. These locations are all trace live-outs including

```
lt0 = lt ld1,min0  // compare with min index
jt -> updateMin    // if smaller, replace min
gt0= gt ld1,max0   // compare with max index
jt -> updateMax    // if larger, replace max
label0:  ...
        ...
updateMin:
    min0 = ld1
    j -> label0
updateMax:
    max0 = ld1
    j -> label0    // resume execution
```

Figure 4.9: Extra code added after `PG7` in Figure 4.8. `label0` is inserted right before instruction `24'`. `updateMin` and `updateMax` code segments are inserted after the `label2` code segment.

stack, heap and global variables, objects and data structures. Snapshots of scalar non-object variables are taken by simply cloning their value, while live-out objects are deep-copied. The deep-copying process is set up such that there are no duplicate copies of the same object in the snapshot in case of cycles in the object graph or when two variables point to the same object. For live-out arrays, an accumulative snapshot mechanism is employed where after an array snapshot is taken before the loop, during each N iteration period at runtime, the minimum and maximum accessed array indices are recorded. Subsequently, all elements between these indices are stored into the array's accumulative snapshot. Since all array indices are already passed to the ParaGuard trace to be checked by the condition mismatch guards, keeping track of these maximum and minimum values is performed inside the ParaGuard trace. Therefore, they impose no extra overhead on the main trace. These values are later sent back to the main trace at the time of periodic snapshot taking. Figure 4.9 shows the extra code for this purpose that needs to be added to Figure 4.8.

TraceMonkey uses a mark-and-sweep garbage collector (GC) and has an API function to add variables to the GC's *root set* to prevent anything the root points to from getting collected. Since there will be no references to the snapshots from within the JavaScript application, the garbage collector needs to be asked explicitly not to touch them until the next snapshot is taken by adding the snapshot entries to the root set. Furthermore, because heap objects are deep-copied while taking snapshots, no object in the snapshots points back to the actual application heap. Therefore, although as explained later, snapshots are recovered once a GC is triggered, in theory, there would be no issue of the GC collecting objects in the heap that are pointed to by the snapshot.

When a guard triggers inside the ParaGuard or the main trace, the runtime aborts both threads by sending a signal, restores the previous snapshot and moves back to the interpreter. The rollback operation itself does not add extra overhead compared to the original tracing technique, since it performs the same value forwarding that would have been done for updating the interpreter's state using the native trace data.

Another important issue is what happens when a GC is scheduled. In the original tracing technique, the trace aborts when a GC is invoked. In ParaGuard, the latest correct snapshot is restored after a GC call is triggered. The control is later handed off to the interpreter from the execution location of the previous snapshot. Finally, in order to ensure execution safety in the main trace and avoid catastrophic failures such as null pointer dereference in the native code, signal handlers were defined to catch runtime exceptions, roll back execution to a previous snapshot and switch to the interpretation mode.

In Figure 4.7, instructions `PG2, PG3` and `PG4` are used to branch to `label2` every N iterations. At `label2`, the main thread waits on a condition, set by the ParaGuard thread and marks the end of its execution. When the condition is set, the main trace starts to take the snapshot. Likewise, `PG7`, `PG8`, and `PG9` are used to branch to `label2` in the ParaGuard trace. After branching, the ParaGuard thread broadcasts the barrier release condition to the main thread.

4.4 Optimizations on ParaGuard

In order to further improve the performance benefit of guard promotion, two additional optimizations are introduced. As mentioned in Section 4.3.2, before starting the snapshot taking process, the main thread needs to wait for the ParaGuard thread to catch up. Therefore, the ParaGuard thread should be made as fast as possible. We introduce the *guard branch aggregation* optimization, during which, mid-trace guard conditions are aggregated into a single variable, branches are removed, and at the end of each *N* iterations, the single condition variable is checked for any possible triggered guard. Furthermore, taking snapshots can impose a high overhead on the runtime. To tackle this issue, we propose *profile-based snapshot elimination*, in which, based on a profile of previous executions, the guards that are likely to trigger are kept on the main trace, and snapshots are removed altogether from the program.

4.4.1 Guard Branch Aggregation

Taking a snapshot of the trace state at every N iterations gives us the opportunity to perform another optimization, called guard branch aggregation, in the ParaGuard trace. At the end of each N iteration chunk, we only need to know if trace execution was successful or not and knowing which guard actually triggered is not important. Regardless of the triggered guard, execution is started from the previous snapshot. Therefore, guard branch executions can be

```
label1:
1' :  cx = ldq state[16]      // (*) load context pointer
3  :  ld2 = ld cx[0]          // (+) load context object
4  :  ga_flag &= eq ld2, 0    // (+) check if context is valid
PG5:  barrier shared_buf[0]   // wait for of shared_buf[0]
PG6:  ld1 = ld shared_buf[0]  // load ld1 from the shared_buf[0]
10':  mul1 = mul ld1, 2       // (*) multiply j by 2
11 :  ga_flag &= ov mul1      // (+) check overflow on mul op
13 :  ga_flag &= eq mul1, 0   // (+) check if mul1 is zero
16':  ldq1 = ldq $globl0[8]   // (*) load myArray class
17 :  qi1 = qiand ldq1,       // (+)
   quad #FFFFFFFF:FFFFFFFC
18 :  cl = quad #0:803D20     // (+)
19 :  ga_flag &= qeq qi1, cl  // (+) check if class is array
PG7:  count = add count, 1    // inc snapshot counter
24':  add1 = add ld1, 1       // (*) add 1 to j
25 :  ga_flag &= ov add1      // (+) check add for overflow
29':  lt1 = lt add1, 200      // (*) check loop condition
30':  xf lt1                  // (*) exit trace if finished
PG8:  eq3 = eq count, N       // check snapshot condition
PG9:  jt eq3 -> label2        // jump if snapshot needed
32':  j -> label1             // (*) jump back to the top
label2:
GA1:  xt ga_flag              //side exit if guard triggered
   bdcast paraguard_finish
   j -> label1
```

Figure 4.10: ParaGuard Trace LIR after applying guard branch aggregation. GA1 is the final and only guard check in the ParaGuard trace.

postponed until the end of each N iteration execution chunk in the ParaGuard trace. The two final instructions for every guard are the guard condition generator and the branch itself. Guard branch aggregation combines all guard conditions to a single variable which is later checked by a final branch at the end of the trace after each N iteration period. After applying this optimization, we have essentially converted a trace with a single input and multiple output edges, to one with a single input and two output edges. One downside to using this approach is that in case one of the middle guards fails, the trace has to execute until the end of the iteration chunk. However, in type-stable loops this does not cause any serious performance issues.

Figure 4.10 shows the LIR before and after applying guard branch aggregation. As the figure illustrates, five guard branches (instructions 5, 12, 14, 20, and 26 in Figure 4.8) can be aggregated to just one branch, `GA1`.

4.4.2 Profile-based Snapshot Elimination

In some traces, the overhead of taking snapshots turns out to be quite high, mainly due to the high overhead of taking heap and array snapshots. In these traces, the number of unique memory updates per loop is high and causes the snapshot taking mechanism to be inefficient. This effect can be detected early on during trace execution by monitoring the snapshot taking overhead. When detected, the native trace is aborted and the execution falls back to the original tracing mode without guard promotion. After switching to normal tracing execution, triggered guards are recorded and stored on the client. Since these operations are done inside the JavaScript engine, the profile information can be stored on the client's file system. During the next execution of the same JavaScript program on the client, the guard promotion phase only moves the guards that, according to the stored profile, have not triggered during previous executions. After guard promotion, since no snapshots are taken, if a guard is triggered in the ParaGuard trace, the execution aborts native execution, reverts back to interpretation from the beginning of the loop and adds that guard to the profile for use during future executions.

However, if a guard is triggered in the main trace, extra measures should be taken to enable the interpreter to continue from the guard point rather than the beginning of the loop. During guard promotion, the main execution thread stores the sequential order of all guards (both in the main and ParaGuard traces) in a list referenced by the program counter. If a guard triggers in the main trace, it checks to see if all previous guards in the ParaGuard trace have passed successfully. If so, it falls back to the interpreter and continues interpretation from the guard point. Otherwise, it waits for the remaining guards in the ParaGuard trace to pass. Meanwhile, if a guard triggers in the ParaGuard trace, the execution rolls back to the beginning of the loop in the interpretation mode.

The ParaGuard execution model when applying profile-based snapshot elimination is that the first time a JavaScript application is executed, profile is collected if taking snapshots seem to be too costly. From then on, whenever the same application is run on the client, this profile information can be used and updated. Therefore, the first execution of the application, in the worst case, is almost as fast as the baseline tracing execution. In later executions, the application will be enjoying the extra performance benefits of ParaGuard.

4.5 Experimental Evaluation

4.5.1 Methodology

We evaluated our technique on the TraceMonkey version distributed with Firefox 3.7a1pre using four sets of benchmarks. In addition to the two popular benchmark suites, SunSpider [13] and Google V8 [3], we put together two other suites consisting of 12 image processing filters and 5 games implemented in JavaScript. The image processing filters were extracted from the Pixastic JavaScript Image Processing Library [11]. This library contains 28 filters and effects, out of which the 12 most compute-intensive filters were selected. In the JavaScript game suite, four of the benchmarks (Collision demo [2], Thunder fighter [4], Super JS fighter [5], and Invaders from earth [6]) are demos written using the gameQuery JavaScript game engine [1].

The last benchmark is a PacMan game written in JavaScript [9]. All benchmarks were run 10 times, and the average execution time is reported.

In evaluating the profile-based snapshot elimination optimization, we used different input sets for profiling and actual execution in all 4 benchmark suites. In SunSpider and V8, default inputs are used for actual execution and smaller inputs were generated for the profile run. For the image processing benchmarks, different images were used for profiling and execution. In the gaming benchmarks, since the input to all of them involved some kind of random element along with interactions with the user, the evaluation was more involved. In order to make the performance comparisons feasible, the fact that the behavior of these programs are uniform during the execution time was exploited. Therefore, they were executed for a fixed number of events at the beginning of the benchmark without any user interaction involved. All random events during the execution were recorded and fed back to the program for all runs (different random events were recorded for profiling and actual runs). For instance, in the `PacMan` game, the paths *ghosts* were taking were fixed and the application ran until a *ghost* hit the *PacMan* which stayed still at its original position. Likewise, in the `Collision Demo` benchmark, all box locations, orientations and movement paths were fixed and the benchmark ran until 10 small boxes collided with the main box in the center. Similar measures were taken in the other three programs as well.

SunSpider has 26 JavaScript programs. However, TraceMonkey does not support recursion, the `eval` function, and regular expression `replace` operations, limiting the number of programs that can be traced properly [40]. Consequently, we excluded the following six benchmarks from our experiments: `controlflow-recursive`, `access-binary-trees`, `date-format-to-fte`,

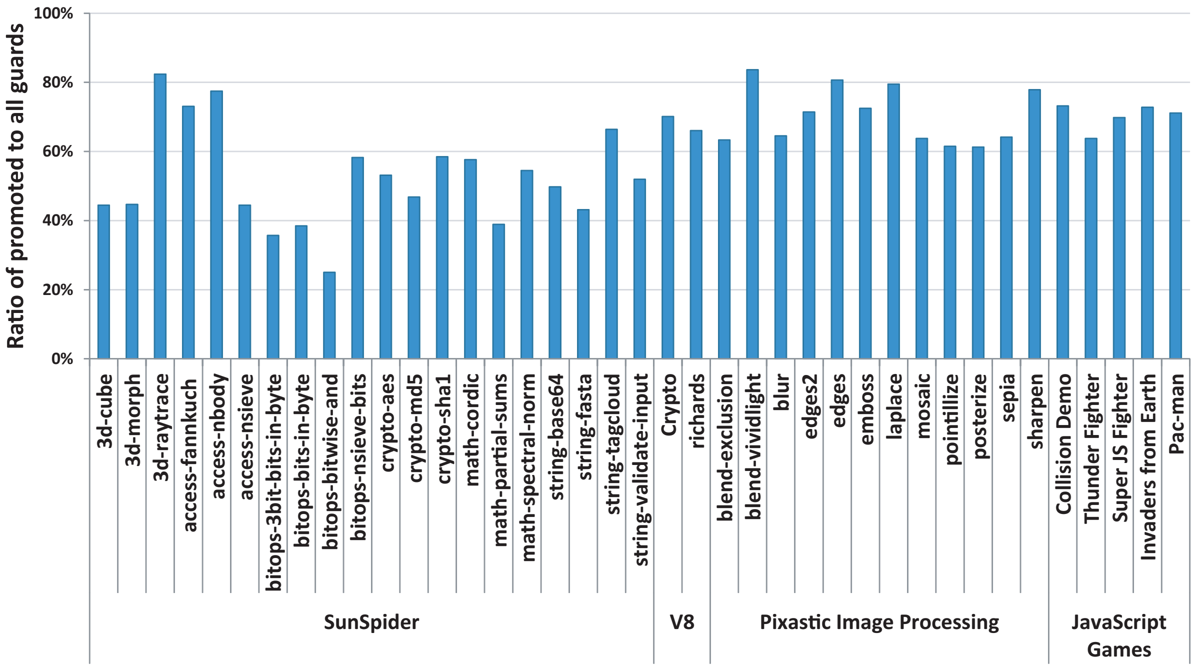


Figure 4.11: Ratio of promoted guards to total number of guards.

`date-format-xparb`, `string-unpack-code`, and `regexp-dna`.

In the V8 suite, we excluded the `RegExp` benchmark due to its dependence on the regular expression library inside the engine rather than tracing. In addition, `DeltaBlue`, `RayTrace`, and `EarleyBoyer` perform poorly on the tracing JIT as only a small fraction of execution is spent running natively, mainly due to the lack of support for recursion in TraceMonkey. Therefore, we excluded them from our results as well. All experiments were performed on a system with an Intel Core i7 processor running at 3.20 GHz, and 4 GBs of main memory.

4.5.2 Results

Figure 4.11 presents the number of guards that passed the promotion heuristic and were moved to the ParaGuard trace. These ratios are based on the dynamic LIR instruction count. The heuristic basically rejects the promotion of all guard instructions whose backward slice

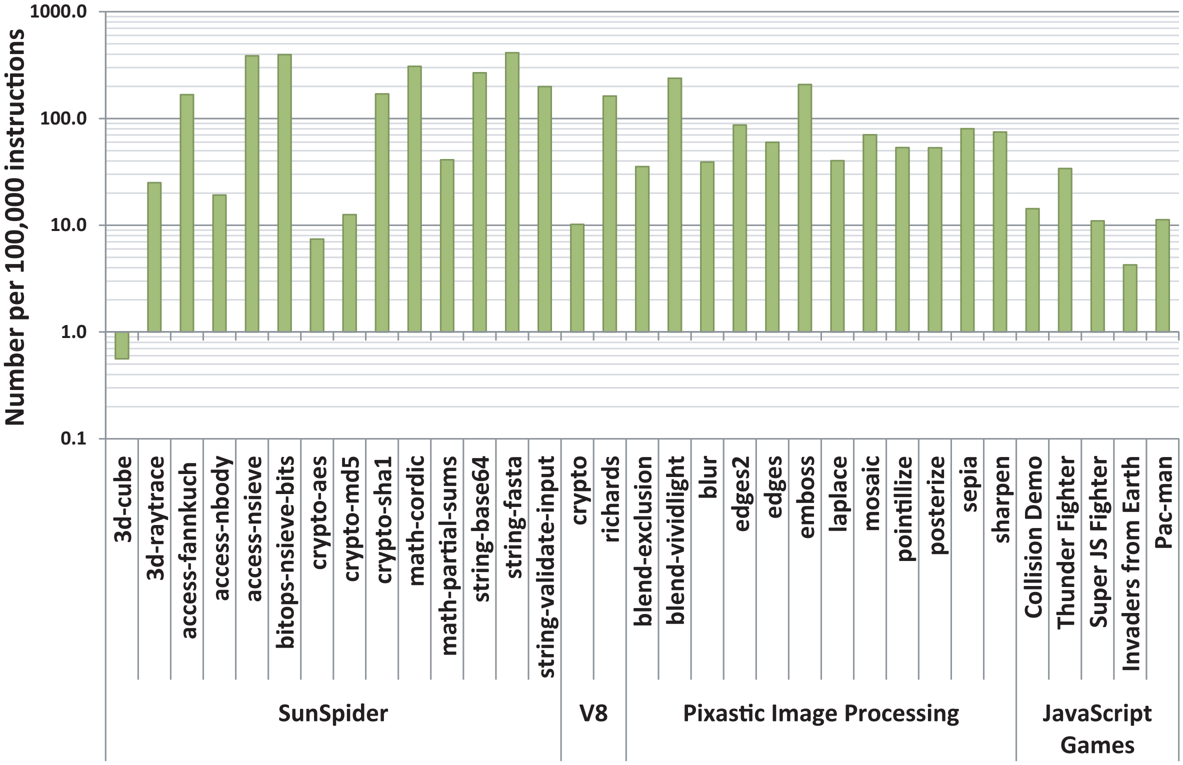


Figure 4.12: Number of triggered guards in ParaGuard during every 100,000 instructions in the guard promotion technique without applying the profile-based snapshot elimination. The y-axis is in logarithmic scale.

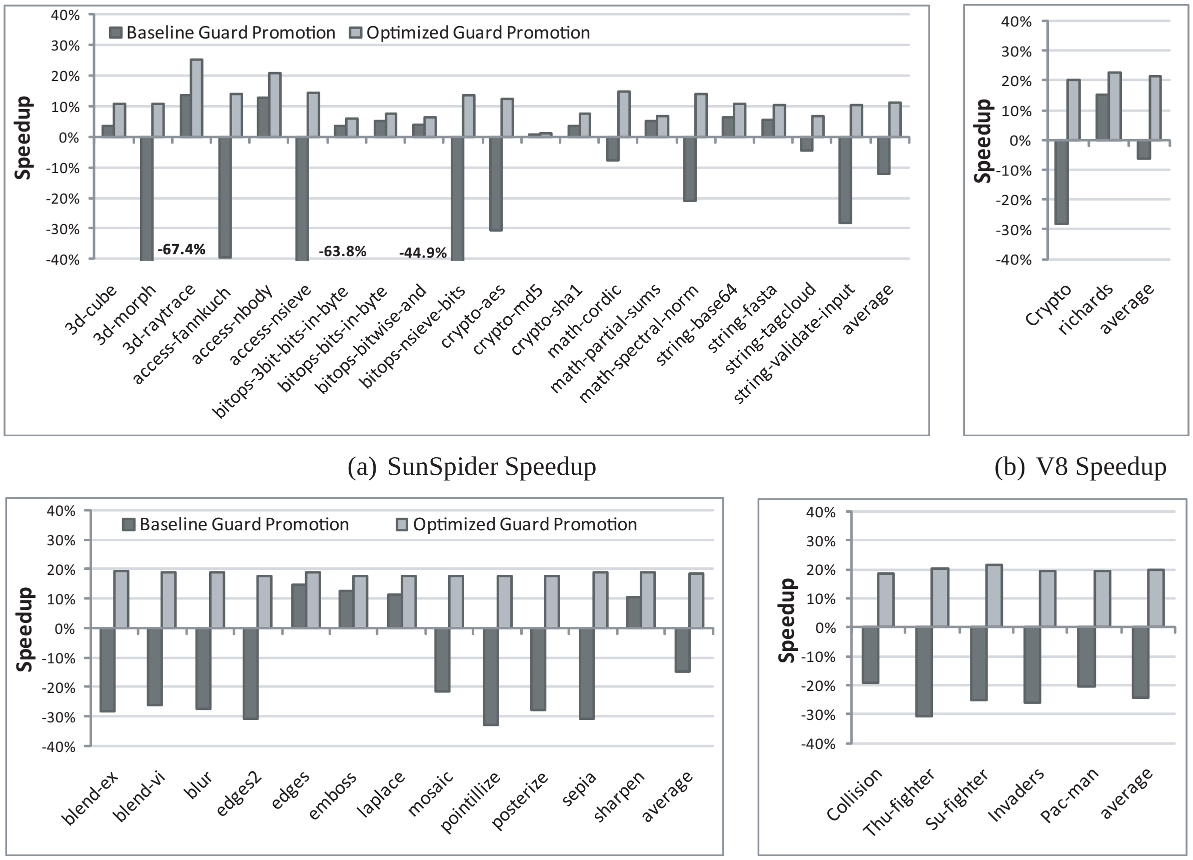
is either very small or should be mostly copied to the ParaGuard trace rather than moved. In addition to loop guards, which are always present in the main trace after guard promotion and are counted as non-promoted guards, most guards that check the integrity of various function return values (such as allocation functions) get rejected by the guard promotion heuristic. In order to move these guards, guard promotion either has to copy the corresponding function calls or move the return value directly using the buffers between the main and ParaGuard thread. Both of these approaches are inefficient, since they add overhead while only saving the guard comparison and branch on the main trace. However, many branch/case, overflow and mismatch guards successfully pass the heuristic and are moved to the ParaGuard trace. As can be seen, the ratio of moved guards varies between 25% and more than 80%.

Figure 4.12 shows the number of triggered guards in the ParaGuard trace, per 100,000 program instructions after applying guard promotion. This figure shows that many hot loops in these applications are type-stable and have infrequent changes in control-flow. This is the reason for effectiveness of the original tracing approach [40] and also the reason behind infrequent roll-backs from snapshots in our method. The majority of these triggered guards are branch guards after which ParaGuard rolls back the state and continues recording other paths of the branch in the interpretation mode.

We originally applied guard branch aggregation to the ParaGuard trace. However, since the ParaGuard trace is shorter than the main trace in all benchmarks, in practice, applying this optimization proved ineffective on the overall performance. Furthermore, due to the infrequent number of side exits in these benchmarks (Figure 4.12), the drawback from identifying guard failures after N iterations rather than at each individual guard was negligible. Therefore, we present the performance results without applying guard branch aggregation. The effect of this optimization on ParaGuard thread's CPU utilization is discussed later in this section.

Figure 4.13 shows the results of applying ParaGuard to the four benchmark suites on 2 processors, where one of them is running the main thread and the other one is running the ParaGuard thread. The left bars in this figure represent the speedup gained compared to sequential trace-based execution after applying guard promotion. The right bars show the resulting speedup after performing profile-based elimination of state snapshots.

Applying guard promotion by itself leads to an average slowdown of 12.2%, 0.1%, 14.7% and 24.2% on SunSpider, V8, image processing and gaming benchmarks, respectively, on two processors compared to the original tracing on one processor. The main reason for the slow-



(a) SunSpider Speedup

(b) V8 Speedup

(c) Image processing Speedup

(d) Games Speedup

Figure 4.13: ParaGuard speedup on 2 processors compared to the baseline tracing. The left bars show the speedup after guard promotion and the right bars show the speedup after applying the profile-based snapshot elimination optimization.

downs in these benchmarks is the large overhead of taking snapshots due to high number of individual array and heap accesses. In some of the benchmarks (16 out of 39 programs), where variable accesses are mostly scalar or multiple iterations update the same array or heap elements, the overhead of taking snapshots is much less and an average speedup of 8% is achieved.

After performing the profile-based snapshot elimination, all triggered guards during previous executions are kept in the main trace. The distribution of the number of these guards is similar to Figure 4.12. As can be seen in Figure 4.13, applying this optimization improves the performance of SunSpider, V8, image processing and game benchmarks to 11.2%, 21.4%,

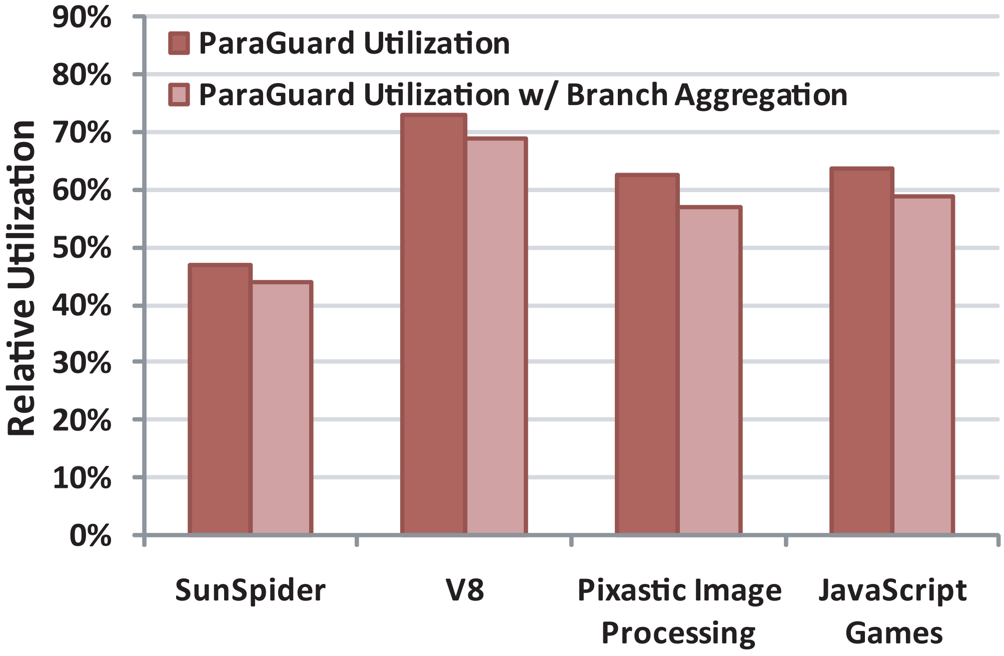


Figure 4.14: Utilization of the ParaGuard thread relative to the main thread before and after applying guard branch aggregation optimization.

18.3% and 19.8% over the baseline tracing, respectively. This improvement is mainly caused by the elimination of the snapshot taking process, and since the guard behaviors are quite stable with different inputs, the number of guards triggered in the ParaGuard trace after applying this optimization is close to zero. The main source of overhead in the execution is the synchronization between the main and the ParaGuard traces.

The highest variation in the profile-based promotion results exists in the SunSpider benchmark suite. This is mainly due to various ratios of promoted guards and also the non-uniform benefit from original tracing in these benchmarks. For instance, `crypto-md5` spends less than 20% of its total execution time in the native mode, and thereby, total performance benefit of our technique is around 1% in this benchmark. Overall, across the 39 benchmarks we studied, the ParaGuard technique achieves an average of 15% speedup over the original tracing technique.

Figure 4.14 shows the CPU utilization of the ParaGuard thread relative to the main thread with and without applying the guard branch aggregation optimization. The average utilization across all our benchmarks is 55% and guard branch optimization is able to reduce it to an average of 51%. This level of utilization shows a potential for using one processor for running ParaGuard threads in two JavaScript execution instances at the same time with each ParaGuard thread exploiting approximately half of the processing power in the extra core. Therefore, for instance, using 3 processors, two JavaScript programs can be accelerated with ParaGuard.

4.6 Related Work

The idea of running traces for specializing hot code regions was proposed in the Dynamo binary rewriting system [22]. Dynamo utilizes run-time information to find hot patches and optimizes machine code accordingly. It also uses trace linking to connect traces together if possible. The idea of trace trees (extending the trace to incorporate new branches rather than forming new traces) used in [30, 40], had been proposed by Gal et al. [41] for Java which is a statically typed language. Our work is based on Mozilla's TraceMonkey, the trace-based JIT compiler described in [40] and released as a part of recent versions of Firefox [7]. TraceMonkey is able to achieve more than 10x speedup on some programs in the SunSpider suite compared to previous versions of SpiderMonkey on Firefox (which is an interpreter-only JavaScript engine). All this performance is achieved by intelligent type specialization and the tracing mechanism. Chang et al. [30] proposed a trace-based JIT compiler implemented on top of Adobe's Tamarin-Central (Tamarin-Tracing) which is their VM for implementing ActionScript and can execute JavaScript programs without any modifications. They also investigate using simpler opcodes in their IR and achieve up to 116% performance improvement over the non-traced code on SunSpider benchmarks. As we showed, by dynamically decomposing execution to main and ParaGuard traces and using extra resources in multicore systems, additional speedups can be achieved on top of tracing techniques on multicore systems. A recent proposal [43] presents a concurrent trace-based JIT in which the compilation from LIR to native code is performed as a background thread. This technique can achieve an average of 6% and a maximum of 25% speedup on the SunSpider benchmark suite. We chose a different approach and actually tried to parallelize the execution by decoupling and parallel execution of runtime checks rather than performing the compilation in parallel with the monitoring/recording. However, these two approaches are orthogonal and can be applied simultaneously.

SlipStream processors [67] speculate on certain code path and executes a pruned version of the program itself in parallel with the original execution. In SlipStream, the speculation support is provided by hardware. In [53], the authors introduce a compilation framework for transformation of the program code to a TLS compatible version and perform thread level speculation. The Mitosis compiler [68] proposes a general framework to extract speculative threads as well as pre-computation slices (p-slices) that allow speculative threads to start earlier. MSSP [94] transforms code into master and slave threads to expose speculative parallelism. It creates a master thread that executes an approximate version of the program containing a frequently executed path, and slave threads that run to check results. All of these speculative multi-threading works parallelize the main computation for purposes of prefetching or exploiting computational parallelism, where as in ParaGuard, we perform domain-specific runtime checks in parallel with the main computation in a dynamic language. Furthermore, in contrast to these works, we propose an all-software solution which works on commodity hardware. The LRPD test [72] performs runtime array tracking by using shadow arrays to follow exactly what array elements are touched in each thread. However, in our array snapshot optimization, we only keep track of range of array accesses.

4.7 Summary

As the web becomes the ubiquitous platform for execution of more complicated applications, a growing amount of computation is being handed-off to the client to minimize network traffic and improve user experience. The flexibility and ease of prototyping in the JavaScript language has made it the language of choice for most client-side web applications. However, as JavaScript applications are becoming larger and more computation intensive, there is more need for building high performance JavaScript engines in the client's browser. Trace-based JIT compilation is one approach towards tackling this issue. In this work, we proposed ParaGuard, which decouples execution from the runtime checks in a trace-based JavaScript engine and accelerates the execution by utilizing extra resources on multicore systems. We also introduced optimizations to further improve the performance. We showed that ParaGuard obtains an average of 15% speedup on two processors across 2 industry-standard benchmark suites, SpiderMonkey and V8, and two sets of JavaScript applications from the image processing and gaming domains.

CHAPTER 5

Summary and Conclusion

As multicore systems become the dominant mainstream computing platform, one of the most difficult challenges the industry faces is software. Applications with large amounts of explicit thread-level parallelism naturally scale performance with the number of cores, but sequential applications realize little to no gains with additional cores.

In this dissertation, we investigate solutions to this problem through automatic speculative parallelization that frees the programmer from the difficult task of parallel programming and offers hope to scale the performance for the vast amount of legacy sequential software. We looked into the automatic parallelization problem in two sequential application domains of C/C++ programs and client-side web applications written in JavaScript. Due to the different language properties and deployment methods, each application domain poses different set of challenges that needed to be tackled. There have been previous proposals on speculative parallelization of C/C++ and Java applications. However, the key distinctive goal in our work is realization of parallelism on commodity hardware without any hardware speculation support for C/C++ and JavaScript. Furthermore, this dissertation is the first work proposing automatic parallelization of JavaScript web applications.

In Chapter 2, we presented the required compiler technology along with a speculative runtime system, *STMlite*, for automatic parallelization of sequential C/C++ applications. By centralizing the commit stage in STMlite, making use of software bloom filter based signatures of read and written addresses, and lifting some of the general STM requirements such as strong atomicity while retaining correctness, we were able to reduce much of the overhead involved in conventional software transactional memory systems.

As many software and services are pushed to the web front, and web applications are becoming more complex, a growing portion of computation is shifted to the client side by developers to improve responsiveness in these applications and reduce the load on the network. Therefore, there is an increasing demand for high performance client-side execution, often implemented in JavaScript. This has led to many recent efforts to improve the performance of JavaScript engines in web browsers. Furthermore, considering the wide-spread deployment of multicores in today's computing systems, exploiting parallelism in these applications is a promising approach to meet their performance requirements. However, JavaScript has traditionally been treated as a sequential language with no support for multithreading, limiting its potential to make use of the extra computing power in multicore systems. In this work, to exploit hardware concurrency while retaining traditional sequential programming model, we introduced automatic runtime parallelization methods for JavaScript applications on the client's browser.

In Chapter 3, we proposed *ParaScript*, a runtime scheme for identifying parallelizable loops, generating the parallel code on-the-fly, and speculatively executing it. The ultra low-cost speculation engine consists of a checkpointing scheme and a runtime dependence detection mechanism. It is shown that by employing these schemes, JavaScript applications achieve considerable performance improvements over sequential execution in current browsers using the extra resources on commodity multicore systems.

The last part of this dissertation in Chapter 4, focuses on optimizing trace-based compilation for JavaScript programs. Trace-based just-in-time compilation techniques have been proposed to address the performance bottleneck in JavaScript applications by compiling hot execution traces down to the binary code. During compilation, runtime checks (called guards) are inserted to the binary to abort execution in case a condition not predicted at binary code generation time happens. We introduce *ParaGuard* to off-load these checks in tracing compilers to another thread, while speculatively executing the main trace. If a check fails, ParaGuard aborts the native trace execution and reverts back to interpreting the JavaScript bytecode.

This dissertation have introduced many novel techniques for static and dynamic parallelization of sequential C/C++ and JavaScript programs on commodity hardware. These techniques prove useful in extracting parallelism from legacy applications, while paving the way to explore parallelism potential in emerging application domains such as web applications, where parallelism have rarely been investigated before.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] gameQuery - a javascript game engine with jQuery - http://gamequery.onaluf.org/.

[2] gameQuery - Collision Demo - http://gamequery.onaluf.org/demos/2/.

[3] Google V8 JavaScript Engine - http://code.google.com/p/v8.

[4] http://gamequery.onaluf.org/demos/3/.

[5] http://gamequery.onaluf.org/demos/4/.

[6] Invaders from Earth - http://www.senadbajramovic.com/game/.

[7] Mozilla - Firefox web browser & Thunderbird email client - http://www.mozilla.com.

[8] Online photo editing, online photo sharing - PhotoShop.com - http://www.photoshop.com.

[9] PacMan 2 - http://www.masswerk.at/javapac/js-pacman2.html.

[10] Picnik-Online photo editing in your browser - http://www.picnik.com.

[11] Pixastic: JavaScript Image Processing - http://www.pixastic.com.

[12] Spidermonkey engine - http://www.mozilla.org/js/spidermonkey/.

[13] SunSpider JavaScript Benchmark - http://www.webkit.org/perf/sunspider/sunspider.html.

[14] The Render Engine - http://www.renderengine.com/.

[15] V8 Benchmark Suite - http://code.google.com/apis/v8/benchmarks.html.

[16] Web hypertext application technology working group specifications for web workers - http://whatwg.org/ww.

[17] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 185–196, 2009.

[18] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proc. of the '06 Conference on Programming Language Design and Implementation*, pages 26–37, 2006.

[19] Mattew Allen, Srinath Sridharan, and Gurindar Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 85–96, 2009.

[20] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[21] Pedro V. Artigas, Manish Gupta, Samuel Midkiff, and José Moreira. Automatic loop transformations and parallelization for Java. In *Proc. of the 2000 International Conference on Supercomputing*, pages 1–10, 2000.

[22] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. of the ’00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[23] Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In *SPAA ’02: 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 99–108, 2002.

[24] William Blume and Rudolf Eigenmann. The range test: a dependence test for symbolic, non-linear expressions. In *Proc. of the 1994 International Conference on Supercomputing*, pages 528–537, 1994.

[25] William Blume et al. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[26] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. Revisiting the sequential programming model for multi-core. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 69–81, December 2007.

[27] Brian D. Carlstrom et al. The Atomos transactional programming language. In *Proc. of the ’06 Conference on Programming Language Design and Implementation*, pages 1–13, June 2006.

[28] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):46–58, 2008.

[29] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[30] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proc. of the 2009 international conference on Virtual Execution Environments*, pages 71–80, 2009.

[31] Shailender Chaudhry, Robert Cypher, Mangus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Hakan Zeffer, and Marc Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, 2009.

[32] Michael K. Chen and Kunle Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proc. of the 7th International Conference on Parallel Architectures and Compilation Techniques*, page 176, October 1998.

[33] Michael K. Chen and Kunle Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 434–446, 2003.

[34] Keith Cooper et al. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[35] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proc. of the 2006 International Symposium on Distributed Computing*, 2006.

[36] Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, pages 21–33, 2007.

[37] Zhao-Hui Du et al. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proc. of the '04 Conference on Programming Language Design and Implementation*, pages 71–81, 2004.

[38] Matthew Frank. *SUDS: Automatic parallelization for Raw Processors*. PhD thesis, MIT, 2003.

[39] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the '98 Conference on Programming Language Design and Implementation*, pages 212–223, June 1998.

[40] Andreas Gal et al. Trace-based just-in-time type specialization for dynamic languages. In *Proc. of the '09 Conference on Programming Language Design and Implementation*, pages 465–478, 2009.

[41] Andreas Gal, Michael Franz, and Christian W. Probst. HotpathVM: An Effective JIT for Resource-constrained Devices. In *Proc. of the 2006 international conference on Virtual execution environments*, pages 144–153, June 2006.

[42] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the USENIX Security Symposium*, pages 151–163, August 2009.

[43] Jungwoo Ha, Mohammad Haghighat, Shengnan Cong, and Kathryn McKinley. A concurrent trace-based just-in-time compiler for JavaScript. Technical Report TR-09-06, University of Texas, Austin, February 2009.

[44] Mary Hall, Jennifer Anderson, Saman Amarasinghe, Brian Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[45] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[46] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, page 102, June 2004.

[47] Tim Harris and Keir Fraser. Language support for lightweight transactions. *Proceedings of the OOPSLA'03*, 38(11):388–402, 2003.

[48] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan & Claypool Publishers, 2010.

[49] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. *Proc. of the '06 Conference on Programming Language Design and Implementation*, 41(6):14–25, 2006.

[50] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 339–353. Springer-Verlag, 2002.

[51] Tory A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proc. of the '04 Conference on Programming Language Design and Implementation*, pages 59–70, June 2004.

[52] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[53] Wei Liu et al. POSH: A TLS compiler that exploits program structure. In *Proc. of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, April 2006.

[54] Virendra J. Marathe, William N. Scherer, and Michael L. Scott. Adaptive software transactional memory. In *Proc. of the 2005 International Symposium on Distributed Computing*, pages 354–368, September 2005.

[55] Pedro Marcuello and Antonio Gonzalez. Thread-spawning schemes for speculative multithreading. In *Proc. of the 8th International Symposium on High-Performance Computer Architecture*, page 55, February 2002.

[56] Thomas Mason. LAMPVIEW: A Loop-Aware Toolset for Facilitating Parallelization. Master's thesis, Dept. of Electrical Engineeringi, Princeton University, August 2009.

[57] Mojtaba Mehrara, Jeff Hao, Po chun Hsu, and Scott Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proc. of the '09 Conference on Programming Language Design and Implementation*, pages 166–176, June 2009.

[58] Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. In *Proc. of the 17th International Symposium on High-Performance Computer Architecture*, pages 87–98, February 2011.

[59] Mojtaba Mehrara, Thomas Jablin, Dan Upton, David August, Kim Hazelwood, and Scott Mahlke. Multicore Compilation Strategies and Challenges. *IEEE Signal Processing Magazine*, 26(6):55–63, November 2009.

[60] Mojtaba Mehrara and Scott Mahlke. Dynamically accelerating client-side web applications through decoupled execution. In *Proc. of the 2011 International Symposium on Code Generation and Optimization*, April 2011.

[61] James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch. Crom: Faster Web Browsing Using Speculative Execution. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 127–142, April 2010.

[62] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of IISWC08*, 2008.

[63] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 69–80, New York, NY, USA, 2007. ACM.

[64] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, February 2006.

[65] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University, February 1997.

[66] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[67] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 269–280, 2000.

[68] Carlos Garcia Quinones et al. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proc. of the ’05 Conference on Programming Language Design and Implementation*, pages 269–279, June 2005.

[69] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas Jablin, and David August. Speculative parallelization using software multi-threaded transactions. In *18th International*

Conference on Architectural Support for Programming Languages and Operating Systems, pages 65–76, 2010.

[70] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, 2004.

[71] Paruj Ratanaworabhan, Benjamin Livshits, David Simmons, and Benjamin Zorn. Jsmeter: Characterizing real-world behavior of javascript programs. Technical Report MSR-TR-2009-173, Microsoft Research, December 2009.

[72] Lawrence Rauchwerger and David A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160, 1999.

[73] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 1–12, 2010.

[74] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 185–196, November 2006.

[75] Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *Proceedings of the OOPSLA'08*, pages 181–194, 2008.

[76] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, October 1998.

[77] Nir Shavit and Dan Touitou. Software transactional memory. *Journal of Parallel and Distributed Computing*, 10(2):99–116, February 1997.

[78] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *Proc. of the '07 Conference on Programming Language Design and Implementation*, pages 78–88, 2007.

[79] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible Decoupled Transactional Memory Support. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 139–150, 2008.

[80] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[81] Michael F. Spear, Virendra J. Marathe, William N. Scherer Iii, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proc. of the 2006 International Symposium on Distributed Computing*, 2006.

[82] Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA '08: 20th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 275–284, 2008.

[83] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[84] J. Greggory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[85] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 2002 International Conference on Compiler Construction*, pages 179–196, 2002.

[86] Chen Tian, Min Feng, and Rajiv Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 62–73, 2010.

[87] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew Bridges, Guilherme Ottoni, and David August. Speculative Decoupled Software Pipelining. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 49–59, September 2007.

[88] K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: Automatically Securing Distributed Web Applications Through Replicated Execution. In *Proceedings of the Conference on Computer and Communications Security*, pages 173–186, October 2009.

[89] Cheng Wang, Youfeng Wu, Edson Borin, Shiliang Hu, Wei Liu, Dave Sager, Tin fook Ngai, and Jesse Fang. Dynamic parallelization of single-threaded binary programs using

speculative slicing. In *Proc. of the 2009 International Conference on Supercomputing*, pages 158–168, 2009.

[90] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[91] Efe Yardimci and Michael Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *2006 Symposium on Computing Frontiers*, pages 127–138, 2006.

[92] Luke Yen et al. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 261–272, February 2007.

[93] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, pages 290–301, February 2008.

[94] Craig Zilles and Gurindar Sohi. Master/slave speculative parallelization. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 85–96, November 2002.