Efficient Design and Implementation of Software Thread-Level Speculation

Zhen Cao

Doctor of Philosophy

School of Computer Science

McGill University

Montreal,Quebec

2015-12-10

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

Copyright © 2015 by Zhen Cao

DEDICATION

in dedication to my parents

ACKNOWLEDGEMENTS

There are many people I hope to acknowledge in this thesis. First and foremost, I thank my supervisor Professor Clark Verbrugge. His expertise on compilers and system software are of enormous help to the design and implementation of the MUTLS framework. Without his supervision and suggestions, the thesis would have been impossible. Professor Laurie Hendren asked thought-provoking questions on my research topics and provided various important suggestions on my project presentation and the final thesis. The thesis work is built on the LLVM compiler infrastructure, and I would like to thank all who have made contributions and support to create such a successful project.

As a Ph.D. student of the Sable research group of School of Computer Science of McGill University, I received many help from other students in the group. This thesis is motivated by Christopher John Francis Pickett's Ph.D. thesis work, which provided precious experiences and ideas for my research in the area. Nurudeen Lameed, Anton Dubrau, Ismail Badawi and Matthieu Dubet explained to me various design, implementation and usability issues of the McLab/McVM framework, which greatly helped me with the MUTLS and McLab/McVM integration work. Ismail Badawi also helped me publish the MUTLS project online in the Sable research group project page. Rahul Garg and Sameer Jagdale presented me the design of the GPU back-end for McVM. Sameer Jagdale also patiently answered my many GPU programming questions. Erick Lavoie and Sujay Kathrotia told me about their work on integrating JavaScript VM with Matlab.

Ismail Badawi and Sebastien Lemieux-Codere translated the English abstract of the thesis into French. I am very grateful for their help.

I would like to thank my internal examiner Professor Brett Meyer and external examiner Professor Chen Ding for their valuable comments and suggestions that significantly improved the initial thesis.

The experiments of the thesis are performed on a 64-core AMD Opteron server, on which Nurudeen Lameed helped to install the software required by the MUTLS system. I also thank School of Computer Science Help Desk members including Ron Simpson, Andrew Bogecho and Kailesh Mussai for their continuous maintenance effort. Otherwise, the extensive data presented in the thesis would not have been possible.

Alexander Krolik worked on the MUTLS project exploring asymmetric block-ization for one of his undergraduate research courses. He did outstanding work and achieved interesting results. It was pleasant working with him. I hope the research experience would prove rewarding in his future endeavours.

I have been teaching assistant of four McGill undergraduate courses, including one year each of COMP-303 software development and COMP-251 algorithms and data structures, and two years of COMP-409 concurrent programming. I thank the excellent McGill undergraduate students for their enthusiasm and inspiration, and would like to have the privilege to wish them bright future. I would like to thank other teaching assistants Bentley Oakes, Simon-Keita Brisson and Liana Yepremyan for their great work as well.

I have been receiving generous funding support throughout my course of Ph.D. studies, including 4 years of Natural Sciences and Engineering Research Council of Canada (NSERC) funding, and 3 years of Graduate Excellence Award and 1 year of Lorne Trottier Fellowship from McGill University. I was also granted travel support from ACM SIGPLAN Professional Activities Committee (PAC) and SIGMICRO to attend the CGO 2015 Student Research Competition (SRC) sponsored by Microsoft Research.

I thank Sebastien Lemieux-Codere for teaming up with me in 2012, and Abhishek Gupta in 2013 and 2014, to participate in the IEEEXtreme programming competition. We achieved impressive results: 5th, 5th and 2nd in Canada and 41st, 39th and 27th in the world. Thank IEEE for organizing these successful events and McGill IEEE Student Branch for the full support.

Paul Kruszewski, McGill Ph.D. Alumnus and CEO of the TandemLaunch incubated company Wrnch, assigned me a contract as consultant to port a Windows application to Mac. I also attended McGill Entrepreneurs Society events and an IEEE sponsored PERSWADE Workshop on Innovation, Intellectual Property and Entrepreneurship. These experiences further inspired me to pursue entrepreneurship.

I want to thank my parents for their continuous moral support for me to complete a better thesis and Ph.D. degree.

Last but not least, I would like to thank all who have contributed to this and related fields of the thesis, including all works in the references. Hopefully this would serve as another stone in the already grand building of literature. I would feel greatly

appreciated if the readers would find the thesis helpful to their research, study or work.

ABSTRACT

Software approaches to Thread-Level Speculation (TLS) have been recently explored, bypassing the need for specialized hardware designs. These approaches, however, tend to focus on source or VM-level implementations aimed at specific language and runtime environments. In addition, previous software approaches tend to make use of a simple thread forking model, reducing their ability to extract substantial parallelism from tree-form recursion programs. We propose a Mixed forking model Universal software-TLS (MUTLS) system to overcome these limitations. Our work demonstrates that actual speedup is achievable on existing, commodity multi-core processors while maintaining the flexibility of a highly generic implementation context, though memory-intensive benchmarks could be improved by further optimizations. Our experiments also indicate that a mixed model is preferable for parallelization of tree-form recursion applications over the simple forking models used by previous software-TLS approaches.

We then improve the performance of the MUTLS system by proposing memory buffering optimizations. Traditional "lazy" buffering mechanisms enable strong isolation of speculative threads, but imply large memory overheads, while more recent "eager" mechanisms improve scalability, but are more sensitive to data dependencies and have higher rollback costs. We thus describe an integrated system that incorporates the best of both designs, automatically selecting the best buffering mechanism.

Our approach builds on novel buffering designs with specific optimizations that improve both lazy and eager buffer management, allowing us to achieve 71% geometric mean performance of the OpenMP manually parallelized version.

Fork-heuristics play a key role in automatic parallelization using TLS. Current fork-heuristics either lack real parallel execution environment information to accurately evaluate fork points and/or focus on hardware-TLS implementation which cannot be directly applied to software-TLS. We propose adaptive fork-heuristics as well as a feedback-based selection technique to overcome the problems. Experiments show that the adaptive fork-heuristics and feedback-based selection are generally effective for automatic parallelization using software-TLS, achieving respectively 56% and 76% performance of the manually annotated speculative version. More advanced automatic workload distribution strategies can also help to improve the effectiveness of the automatic parallelization approaches.

Finally, dynamic languages such as Python are difficult to statically analyze and parallelize, which is an ideal context for the dynamic parallelization software-TLS approach. We integrate the Python JIT specializing compiler Numba with the MUTLS system to get a parallelizing compiler for Python. Experiments on 6 benchmarks show that the speculatively parallelized version achieve speedups from 1.8 to 16.4 and from 2.2 to 40.5 for the JIT compiled python programs with and without accounting for JIT compilation time, respectively.

Overall, we demonstrate that software-TLS can be an efficient and effective approach for automatic parallelization on commodity multi-core processors for a variety of language contexts/execution environments.

ABRÉGÉ

Plusieurs approches au niveau logiciel pour la spéculation des fils d'exécution (TLS) ont récemment étées explorées, évitant la nécessité de concevoir du matériel informatique spécialisé. Cependant, ces approches ont tendence à se concentrer sur des implémentations au niveau du code source or des machines virtuelles qui sont destinées à des langages ou environnements particuliers. De plus, les approches logicielles précédentes ont eu tendance à utiliser un modèle de fork de fils d'éxécution plutôt simple, réduisant leur capacité à extraire du parallélisme important à partir de programmes récursifs de forme arbre. Nous proposons un système "Mixed forking model Universal software-TLS (MUTLS)" afin de surmonter ces limitations. Notre recherche démontre que des accélérations sont réalisables sur des processeurs multi-core sur le marché aujourd'hui, tout en conservant la flexibilité d'un contexte de mise en oeuvre très générique – cela dit, la performace sur des programmes de référence à utilisation intensive de mémoire pourraient être améliorés par des optimisations supplémentaires. Nos expériences indiquent aussi que, pour la parallélisation d'applications récursives de forme arbre, un modèle mixte est préférable aux modèles simple utilisés par les approches de TLS logicielles précédentes.

Nous améliorons ensuite la performance de système MUTLS en proposant des optimisations de mise en mémoire tampon. Les mécanismes traditionnels "paresseux" de mise en mémoire tampon permettent d'isoler efficacement les fils spéculatifs, au coût de hautes surcharges de mémoire, tandis que les mécanismes "strictes" plus récents sont plus évolutifs, mais plus sensibles aux dépendences de données et ont

des coûts de restauration plus élevés. Nous décrivons donc un système intégré qui incorpore le meilleur des deux conceptions, sélectionnant automatiquement le meilleur mécanisme de mise en mémoire tampon. Notre approche se fonde sur de nouvelles conceptions de mise en mémoire tampon avec des optimisations qui améliorent la gestion"paresseuse" et "stricte" de la mémpire tampon, nous permettant ainsi d'atteindre une performance moyenne géométrique de 71% par rapport à une version parallelisée manuellement avec OpenMP. L'application de ces optimisations de mise en mémoire tampon est donc une composante importante de l'ensemble d'optimisations nécessaire pour une mise en oeuvre TLS logicielle efficace et pratique.

Les heuristiques de fork jouent un rôle clé dans la parallélisation automatique avec TLS. Les heuristiques n'ont pas suffisamment d'information sur l'environnement d'exécution parallèle pour évaluer avec précision les points de fork et/ou se concentrent sur des implémentations matérielles de TLS qui ne sont pas applicables aux implémentations logiciels de TLS. Nous proposons des heuristiques adaptives ainsi qu'une technique de sélection réactive afin de surmonter ces problèmes. Des expériences démontrent que les heuristiques adaptatives et notre technique de sélection réactive sont généralement efficaces pour la parallélisation automatique avec TLS, atteignant respectivement 56% et 76% de la performance des versions spéculative annotées à la main. Des stratégies plus sophistiquées de distribution de charge de travail pourraient aussi aider à améliorer l'efficacité des approches de parallélisation automatique.

Finalement, les langages dynamiques tel que Python sont difficiles à analyser statiquement et paralléliser, mais forment un context idéal pour l'approche logicielle

TLS à la parallélisation dynamique. Nous intégrons Numba, le compilateur précialisé en JIT pour Python, avec le système MUTLS, afin d'obtenir un compilateur parallélisant pour Python. Nos expérience sur 6 programmes de références démontrent que la version parallelisée qui utilise TLS atteint des accélérations de 1.8 à 16.4 par rapport à la version JIT si on inclus le temps de compilation et de 2.2 à 40.5 si on exclus le temps de compilation.

Dans l'ensemble, nous démontrons que la TLS logicielle peut être une approche efficace pour la parallélisation automatique sur des processeurs multi-core standard pour une variété de langages et d'environnements d'exécution.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

As multi-core and many-core processing have been the trend of CPU architectures to advance performance, and stream processors such as GPUs have been increasingly more capable for general purpose computing, thread-level parallelism is becoming more and more critical to exploit computing powers from modern hardware.

However, traditional shared memory [21] parallel programming paradigms such as OpenMP [19], pthread [23] and OpenCL [18] are error-prone as they require the programmer to explicitly deal with parallel execution details such as inter-thread data communication and synchronization. In addition, parallel programs are difficult to debug due to nondeterministic execution and subtle issues such as race condition [25], deadlock [3] and memory inconsistency [14].

To address this problem, *speculative*, or *optimistic*, parallelism, has been proposed [144, 152, 51, 132, 176, 60, 94]. Speculative parallelism guarantees correctness of parallel programs in terms of sequential execution equivalence. The runtime system tracks parallel execution and automatically rolls back offending threads, thereby releasing the programmer from the responsibility for correctness of parallelization. Parallelizing programs by speculative approaches is straightforward. Given their sequential counterpart, if the sequential program is correct, the parallel program is guaranteed to be correct. Another advantage of speculative approaches is that

they can be applied to parallelize pointer-based irregular applications that are input-dependent and cannot be parallelized by traditional approaches such as OpenMP [94].

1.1 Thread-Level Speculation (TLS)

Thread-level speculation (TLS), or speculative multithreading (SpMT) is a safety-guaranteed approach to automatic or implicit parallelization [153, 128, 63, 162, 125, 135, 179]. Speculative threads are optimistically launched at *fork points*, executing a code sequence from *join points* well ahead of their parent thread. Safety is preserved in this speculative model by buffering reads and writes of the speculative thread. Once the parent thread reaches the join point the latter may be *joined,* committing speculative writes to main memory and merging its execution state into the parent thread, provided no read conflicts have occurred. In the presence of conflicts the speculative child execution is discarded or rolled back for re-execution by the parent. The situations are illustrated in Figure 1–1.

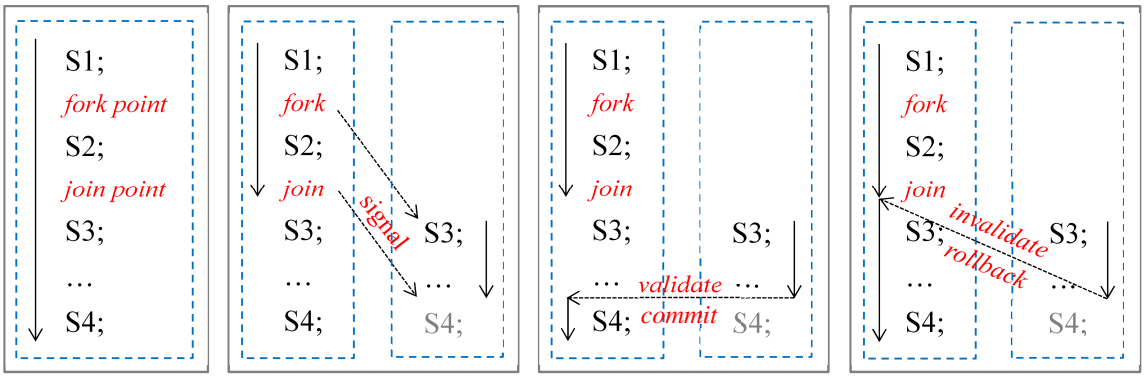


Figure 1–1: Thread-Level Speculation (TLS)

As shown in Figure 1–1(a), the sequential program is annotated with a pair of *fork point* and *join point*, which can be performed manually by the programmer or

automatically by tools such as the compiler or profiler, as will be discussed in later chapters. During program execution in Figure 1–1(b), when a thread (in the left blue box) reaches the fork point just before S2, the thread forks a speculative child thread (in the right blue box) executing from S3 that is just after the join point. To guarantee safety of the speculative execution, memory reads/writes of the child speculative thread are buffered. When the parent thread reaches the join point just before S3, it then signals the child thread to join, which now executes to some future point just before S4.

When receiving notification from the parent thread to join, the child thread validates its speculative buffering. A conflict is detected if the child speculative thread has read from an address a value that is different from the current value in the main memory (stale read). If there is no conflict, as illustrated in Figure 1–1(c), the child thread commits its write buffer to the main memory, and the parent thread jumps to the execution point S4 of the child thread and continues execution, in which case the parent and the child threads are effectively parallelized. Otherwise, the child thread execution is wasted and rolled back, while the parent thread continues execution from S3, as shown in Figure 1–1(d).

As can be seen from the cases (c) and (d) of Figure 1–1, appropriate selection of fork/join points is a critical factor of effective TLS. The selection of fork/join points can be either automated with heuristics [51, 63, 162, 140, 177, 61, 103, 44] or manually specified based on programmer directives [137, 124, 43]. Research into the former has demonstrated TLS as a promising technique for automatic parallelization.

Thread-level speculation has received significant attention in terms of hardware development as a feasible technique for automatic parallelization [51, 63, 162, 140, 107]. Hardware-centric TLS uses dedicated hardware to manage speculative threads and memory buffering, which usually are efficient and low-overhead runtime system implementations: no memory buffering overhead and thread fork, validation, commit and rollback overhead is in the order of 10 cycles. However, it also tends to be constrained to small granularity parallelism due to limited hardware buffering resources for maintaining the speculative data, which are typically 16KB to 64KB per CPU/thread. Both reasons lead hardware-TLS to finer-grained parallelism in which most threads are less than 1000 cycles. A tight analysis on the SPEC CPU2006 benchmarks showed that the speedup potential uniquely achievable by TLS at the innermost loop level is the order of 1% [91], suggesting TLS needs be applied at a larger granularity to be more effective [60].

Software approaches to TLS have been explored as well, which typically use operating system (OS) level threads as speculative threads and main memory as speculative buffering. Memory accesses of a speculative thread are redirected to its own buffer and recorded in the read/write sets, which are usually implemented as hash tables with memory addresses as hash keys for efficiency. With the advantage of directly applying to existing commodity multiprocessors, software-TLS has been receiving increasingly more attention [132, 124, 179, 45, 116]. Software-TLS also has the advantage of much greater and more flexible resource limits, especially in terms of memory. This potentially allows for larger granularity in the parallelism. However,

software-TLS runtime systems incur larger overhead than their hardware counterpart, which can be partially addressed by applying to larger granularity parallelism to amortize the overhead. Research works have also been proposed to optimize the software-TLS memory buffering [148, 52, 125, 179, 46].

1.2 MUTLS

TLS approaches differ in terms of *forking models*: how they create and manage speculative threads. Two main forking models exist: *in-order* and *out-of-order*. Existing software models have been primarily based on one or the other of these strategies, which allow for good exploitation of parallelism in loops and deep method calls respectively, as will be discussed in sections 2.3.1 and 2.3.2. These simple forking models, however, have limitations with respect to the ability to extract parallelism, and a reliance on pure in-order or pure out-of-order design limits the amount of parallelism that can be found in more complex programs with nested levels of parallelism, including ones that make extensive use of tree-form recursion, such as found in depth-first search and divide-and-conquer programs. These parallelism opportunities can be effectively exploited using the *mixed* forking model, as will be discussed in section 2.3.3.

The immediacy of application of software-TLS also requires some tradeoff in terms of increased overhead and compilation complexity, with existing research efforts based on prototype, language-specific implementations. Realistic and convincing evaluation of such designs, however, requires consideration of a full compiler infrastructure, one that enables both deep investigation and application to a variety of compilation contexts.

In this thesis we propose the Mixed-model Universal software-TLS (MUTLS) system to overcome both limitations of existing software-TLS approaches. First, MUTLS uses a mixed forking model to maximize the potential to extract parallelism in more general classes of programs. Second, MUTLS is universal in that it is language and architecture neutral. Our approach is to build a pure software-TLS design using the popular LLVM compiler framework [7]. We integrate our design into LLVM's machine and language-agnostic intermediate representation (IR), enabling generic application of TLS to arbitrary input and output contexts. This has the advantage of providing a full and non-trivial compiler context for evaluating TLS, as well as allowing the full range of source and hardware pairings enabled by the LLVM framework.

As was discussed in section 1.1, the runtime system and selection of fork/join points are two important issues for effective TLS implementation, which we will address by proposing memory buffering optimizations and adaptive fork-heuristics, respectively. Dynamic languages is difficult to statically analyze and parallelize due to the lack of type information, which is an ideal context for software-TLS and motivates us to explore the benefits of TLS for the dynamic language context.

1.3 Contributions

The thesis makes the following range of contributions.

- We propose the tree-form, mixed forking model which incurs less cascading rollbacks than previous mixed forking models.
- We propose the language and architecture independent software-TLS system MUTLS and modify front-ends for C/C++ and Fortran to support user-driven

speculation. From this we are able to generate native executables (or JIT-based execution) for non-trivial benchmarks to evaluate performance, illustrating the potential of our approach as a means to explore and compare the use of TLS in different language contexts. Software-TLS faces significant challenges in terms of balancing overhead concerns with the many possible design decisions possible in TLS implementation. Our system simplifies this research exploration by allowing for practical experimentation within an optimizing compiler context. We also integrate the tree-form mixed forking model into the MUTLS framework, demonstrating that the more advanced tree-form mixed forking model can be implemented in a flexible software-TLS system across source languages and target architectures.

- With a programmer-directed approach, we perform a deep experimental analysis of the performance of the MUTLS software-TLS system, demonstrating real speedup on both C/C++ and Fortran benchmarks. We also performed the first experiment on depth-first search (DFS) and divide-and-conquer (D&C) tree-form recursion benchmarks and show that a mixed model is preferable to in-order and out-of-order models for tree-form recursion applications.
- We propose memory buffering optimizations to improve the performance and scalability of software-TLS. One of the proposed optimizations also enables the application of software-TLS to any granularity parallelism without causing buffering overflow. We also propose the readonly-page optimization and

the buffering integration mechanism to automatically identify readonly, independent and dependent variables on-the-fly and utilize appropriate optimization/buffering for different variables, which can effectively combine the strength of different optimization/buffering designs in each thread.

- We propose adaptive fork-heuristics to enable effective automatic parallelization using software-TLS. We also propose a feedback-based selection technique to reduce the adaptive fork-heuristics overhead through re-compilation using the heuristics log files. We then integrate the adaptive fork-heuristics and feedback-based selection technique into the MUTLS system and implement related compiler transformations and optimizations to achieve an effective automatic parallelizing compiler using software-TLS, demonstrating that software-TLS can be a practical approach for automatic parallelization on commodity multi-core processors.
- We integrate the LLVM-based Python specializing JIT compiler Numba with the software-TLS system MUTLS to obtain a parallelizing compiler for Python, demonstrating that MUTLS can be an effective framework for parallelization of dynamic languages. Our implementation and optimization takes relatively small effort, which also demonstrates that the language and architecture independent software-TLS approach employed by MUTLS is as well valuable in dynamic language context.

1.4 Roadmap

Chapter 2 describes the background for the subsequent chapters, including the LLVM compiler infrastructure, TLS speculation and forking models as well as the

software-TLS memory buffering implementations. In Chapter 3, using an incremental approach, we present the MUTLS software-TLS system design, implementation and optimizations. We then perform experiments on the MUTLS system design of Chapter 3 and present the results in Chapter 4. In Chapter 5, we address the software-TLS runtime system issue as was discussed in section 1.1 by proposing memory buffering optimizations to improve the performance of memory-intensive applications for the MUTLS software-TLS system implemented in Chapter 3. In Chapter 6, we address the problem of automatically selecting appropriate fork/join points discussed in section 1.1 by proposing adaptive fork-heuristics and feedback-based selection, which then allows us to implement automatic parallelization in the MUTLS system. Chapter 7 implements and experiments with a software-TLS system for Python by integrating the Python specializing JIT compiler Numba with the MUTLS framework. We present related research works in Chapter 8 and finally conclude the thesis and discuss possible future work in Chapter 9.

CHAPTER 2
Background

In this chapter, we introduce the background to implement and optimize the MUTLS software thread-level speculation (TLS) system. First, we describe the LLVM compiler framework [7] and intermediate representation, which is the infrastructure that MUTLS is integrated into and based on. Then we discuss TLS concepts of speculation and forking models, which inform various design decisions of the MUTLS system which will be discussed in Chapter 3. Finally, we present TLS memory buffering mechanisms which form the basis of MUTLS memory buffering optimizations of Chapter 5.

2.1 LLVM

MUTLS is purely based on the well-defined LLVM intermediate representation (IR). LLVM [96] is a compiler framework that allows for multiple source languages and target architectures through the use of a generic, Static Single Assignment (SSA) based IR. This intermediate form is a type-safe, expressive assembly code which can be regarded as a universal abstract machine capable of representing all high-level languages.

There are many analyses and optimizations in LLVM, each implemented as a pass. Each pass can specify its required and/or preserved analyses, so transformation passes can use analysis information and assume the IR already underwent specific transformations. There are two sorts of passes: LLVM-IR based passes and Machine

Function passes. Transformations of the former are purely based on the LLVM-IR—i.e. from well-formed LLVM-IR to well-formed LLVM-IR. Machine Function passes are performed in the code generator for specific target architectures. The approach we use implements an LLVM-IR based pass and thus is inherently target independent. The LLVM intermediate representation and passes are illustrated in Figure 2–1.

LLVM has three execution models: static execution, just-in-time (JIT) compilation and interpretation. The first model compiles LLVM-IR to native executable that can directly run on hardware, which is suitable for statically typed languages such as C/C++ and Fortran. While the latter two use LLVM execution engine to dynamically JIT compile/interpret LLVM-IR at runtime, which enables LLVM to serve as backend for dynamically typed languages such as Matlab and Python that typically run in a virtual machine (VM) and/or JIT runtime environment.

LLVM-IR has three equivalent forms: in-memory compiler IR, human readable assembly language and machine bytecode. LLVM compiler analysis and transformation passes perform on the in-memory compiler IR form, which can be saved to and loaded from external storage such as hard disks as the assembly language and/or machine bytecode forms. The assembly language and machine bytecode forms each has its own advantage: the former makes it easier for the user to debug passes and visualize the IR, while the latter is more compact and structured, thus preferable for processing by the compiler, particularly for JIT compilation and/or interpretation.

A sample C program as well as an excerpt of the corresponding LLVM-IR in human readable assembly language form is presented in Figure 2–2. The C program

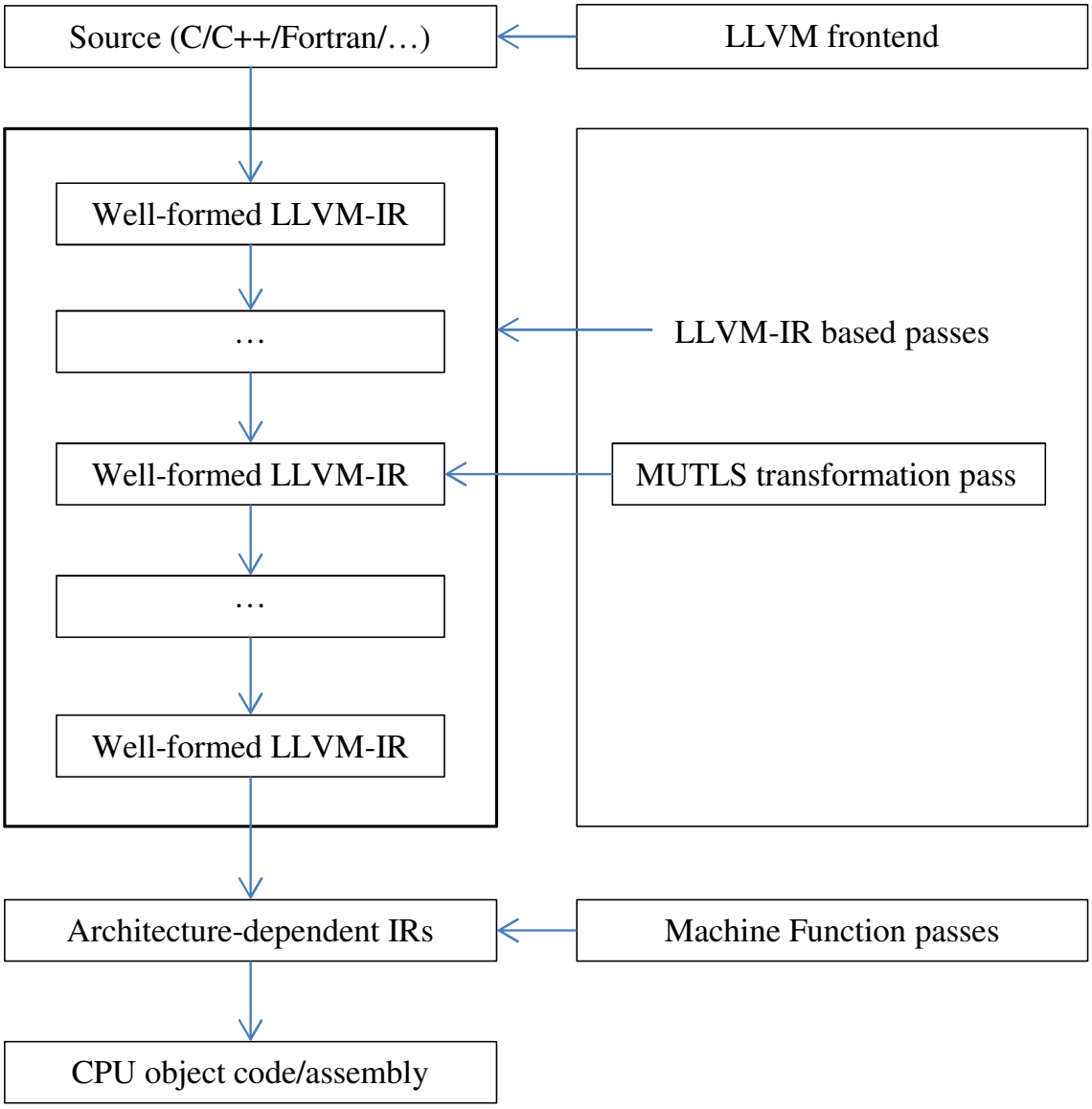


Figure 2–1: LLVM intermediate representations and passes

is a translation unit that implements a function to compute the square root sum of an array `a` of size `n`. The LLVM-IR in 2–2(b) is optimized by the LLVM compiler

infrastructure using the O2 level optimization. In the assembly language form LLVM-IR, global identifiers such as function names (`@f` and `@sqrt`) are prefixed with "`@`". While local identifiers such as register variables (`%0` to `%7`, `%exitcond` and `%.lcssa`), function parameters (`%n` and `%a`) and labels (`%entry`, `%"3"` and `%"5"`) are prefixed with "`%`", except for basic block identifiers (`entry`, `"3"` and `"5"`) which are suffixed with "`:`". Identifiers without "`@`" and "`%`" prefixes or "`:`" suffixes are LLVM keywords such as function definition/declaration (`define`/`declare`), types (`double`, `i1` and `i64`), attributes (`nocapture` and `readonly`), and instructions (`icmp`, `br`, etc). The semicolon "`;`" starts a comment line.

The parameter `%a` of the LLVM function `@f` has two attributes for passes to perform better optimizations: the first attribute `nocapture` means that the function `@f` does not store the pointer `%a` in an object with lifetime longer than `@f`, while the second `readonly` indicates that `@f` does not write to an address derived from the pointer `%a`. The body of the function `@f` consists of three basic blocks. The first block `entry` has two instructions: the first instruction compares the 64-bit integer argument `%n` with the constant 0, and assigns true/false to the boolean (`i1`) register variable `%0` if they are equal/unequal. The second instruction then jumps to the block `"5"` if `%0` is true (`%n` is 0), and to the block `"3"` otherwise.

The first two instructions `%1` and `%2` of the basic block `"3"` are *phi-nodes*, which are a type of nodes specific to the SSA-form control-flow graph (CFG) or IR. In SSA-form IR, each variable can only be assigned a value once (hence the name Static Single Assignment), and in the case of multiple assignment to a variable (e.g. the `i` and `s` variables of the function `f` in Figure 2–2(a)), phi-nodes are introduced to

```
#include <math.h>
double f(size_t n, double a[]){
        double s=0;
        for(size_t i=0;i<n;i++) s+=sqrt(a[i]);
        return s;
}
```

(a) Sample C program

```
define double @f(i64 %n, double* nocapture readonly %a) {
entry:
  %0 = icmp eq i64 %n, 0
  br i1 %0, label %"5", label %"3"

"3":                                              ; preds = %entry, %"3"
  %1 = phi double [ %6, %"3" ], [ 0.000000e+00, %entry ]
  %2 = phi i64 [ %7, %"3" ], [ 0, %entry ]
  %3 = getelementptr double* %a, i64 %2
  %4 = load double* %3, align 8
  %5 = tail call double @sqrt(double %4)
  %6 = fadd double %1, %5
  %7 = add i64 %2, 1
  %exitcond = icmp eq i64 %7, %n
  br i1 %exitcond, label %"5", label %"3"

"5":                                              ; preds = %"3", %entry
  %.lcssa = phi double [ 0.000000e+00, %entry ], [ %6, %"3" ]
  ret double %.lcssa
}

declare double @sqrt(double)
```

(b) LLVM IR excerpt for the C program in (a)

Figure 2–2: LLVM IR: human readable assembly language form

discern the different versions of the variable. For this program, the phi-nodes `%1` and `%2` are corresponding to the variables `s` and `i`, respectively. The phi-node register variable `%1` means that the value of `%1` is 0.000000e+00 (floating point 0) if the block `"3"` is branched to from `entry`, and is `%6` if the block is branched to from `"3"`. Similar is for `%2`.

The register variable `%3` is a pointer derived from the pointer `%a` with index `%2`, `%4` is the loaded double value from `%3`, and `%5` is the value returned by the function call `sqrt(%4)`. The `tail` marker in the call instruction is a compiler hint indicating that the function `sqrt` does not access the stack of the caller function `@f` and thus is candidate for tail call optimization. Then `%6` computes the sum of the `s` value of previous iterations `%1` and return value of the `sqrt` function call of the current iteration `%5`, while `%7` computes the loop index of the next iteration (`%2+1`). At last, `%exitcond` compares the loop index `%7` with the number of iterations `%n`, and branches to the block `"5"` if `%exitcond` is true and to `"3"` to execute the next iteration otherwise.

The last basic block `"5"` assigns the phi-node `%.lcssa` to 0 if it is branched to from entry (%n is 0), and to `%6` if branched to from `"3"` (square root sum of the array `%a`). Finally, `%.lcssa` is returned from the function `@f`.

2.2 TLS Speculation Model

Traditionally, TLS approaches have been characterized by different *speculation models,* based on the selection of fork/join points. Loop-level speculation [51, 63, 162, 125] speculates on loop iterations, with loop iteration boundaries as fork/join joints (illustrated in Figure 2–3(a)). Method-level speculation (MLS) [50, 133] selects

method (function) calls as fork points and speculates on their continuations (Figure 2–3(b)). Arbitrary point speculation [140, 60] generally imposes no constraints on the selection of fork/join points as long as they reside in the same function (Figure 2–3(c)). In principle all these forms are equivalent, although the required code transformations make conversion technically challenging.

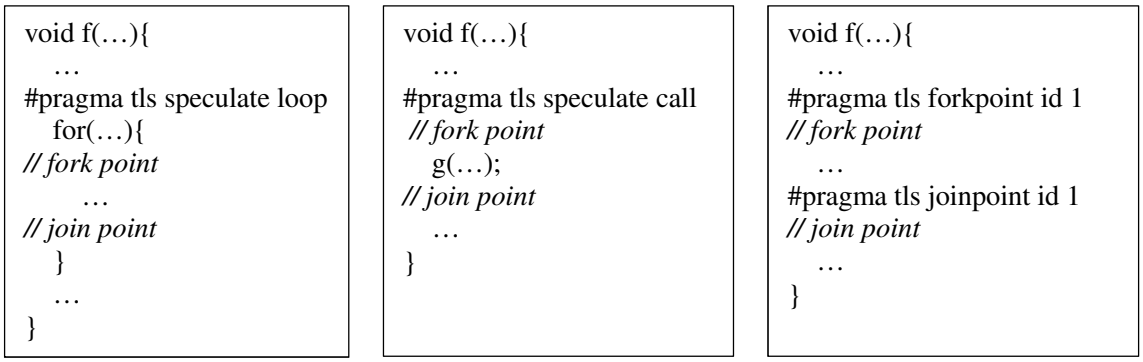


Figure 2–3: Speculation Models. (a) Loop-Level Speculation. Speculative threads are launched at the beginning of a loop iteration that speculatively executes the next loop iteration. (b) Method-Level Speculation (MLS). Speculative threads are launched when reaching a method (function) call that speculatively executes the continuation of the call. (c) Arbitrary Point Speculation. Speculative threads are launched at an annotated fork point that speculatively executes from the corresponding (with the same id) annotated join point.

2.3 TLS Forking Model

Within any of these speculation models, different *forking models* can be used to define how the existence of multiple speculative threads is managed. Each of *in-order*, *out-of-order* and *mixed*, provides different choices, and has greater or lesser affinity for different program contexts. The forking models are also orthogonal to

the speculation models (e.g. an arbitrary point speculation can be either in-order, out-of-order or mixed).

2.3.1 In-Order Forking Model

In the in-order forking model, only the most recently speculated (most speculative) thread can fork a new speculative thread. This model is particularly appropriate for loop-based speculation, with future loop iterations forked in iteration order. The non-speculative parent begins the first iteration, forking a speculative child thread to execute the second, which can then fork a speculative grand-child to execute the third, and so on. Therefore, a typical scenario is that speculative threads are created in the order of their sequential execution: if the start of thread A would be prior to the start of thread B in sequential execution, then thread A is forked by its parent prior to B, and hence the name of this fork model.

This model has the advantage that N threads can efficiently parallelize a loop of N iterations, but the disadvantage that if a speculative thread has to rollback all subsequently speculated threads should also cascade rollback, as well as the constraint that parallelism not found in the most speculative thread may not be exploited.

The in-order forking model is demonstrated in Figure 2–4. The vertical green lines represent parallel thread execution, with points further toward the bottom representing sequentially later execution. Dash green lines indicate committed speculative thread execution and thus reduced program execution time. Blue lines represent forking a speculative thread at a fork point to start execution from the corresponding join point. While red lines represent the thread joining process. In this case, the non-speculative thread T1 forks a child speculative thread T2, which immediately

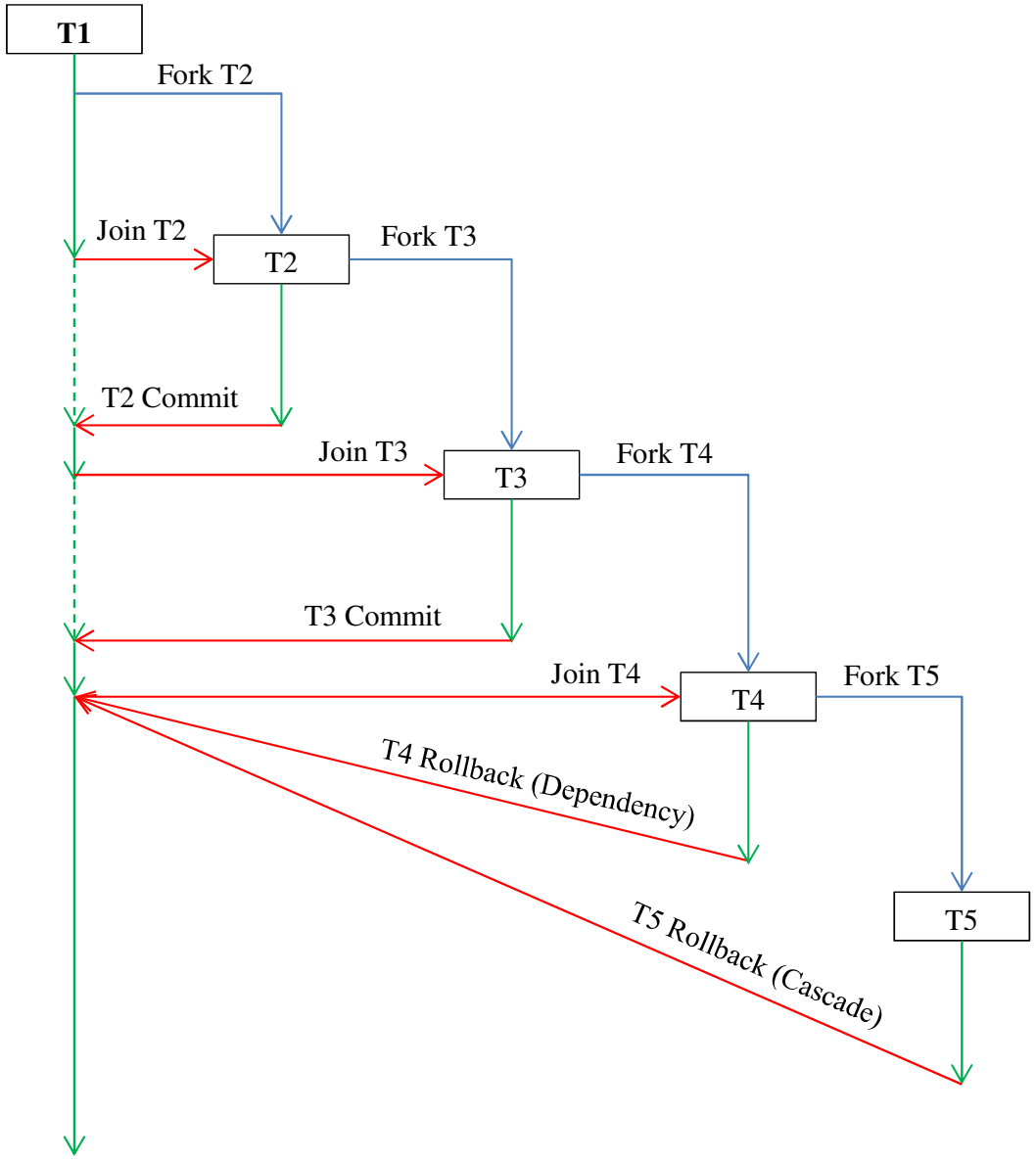


Figure 2–4: In-order Forking Model. The non-speculative thread **T1** forks speculative thread T2, which then speculates T3, and so on. T2 and T3 in-order commit during joining. When T1 joins T4, T4 detects dependency and rolls back, which causes T5 to cascade rollback.

forks the grand-child speculative thread T3, which continues the process to speculate threads T4 and T5. If the example is an instance of loop-level speculation, then each of the 5 threads executes one loop iteration. We can see that among the threads T1 to T5, the higher the thread number, the later the represented sequential execution. It can also be noted that the non-speculative thread joins the speculative threads in the same order as they are speculated (the speculative threads in-order commit). If one of the speculative threads rolls back, for example, due to memory dependencies, then all subsequently speculated threads need also rollback, as is the case for T4 and T5.

2.3.2 Out-of-Order Forking Model

The out-of-order model usually applies to method-level speculation. As the non-speculative parent thread enters a function call, a new thread is forked to execute the method continuation. This process can continue recursively, resulting in speculative threads being forked in the order that the parent descends into nested method calls, and so joined in the reverse order, as the parent returns from each call.

Since all speculative threads are the direct children of the non-speculative thread, this approach avoids the cascading rollback concerns found in in-order models due to "increasing" speculation. It also easily applies to more arbitrary code constructions as long as they exhibit nesting structures, such as C++ nested block statements. The out-of-order model, however, has the disadvantage of limited parallelism on loop-level speculation since the non-speculative thread has to complete an iteration before reaching the fork point again to speculate another thread. The inability to launch speculative threads from speculative threads prevents more than one iteration

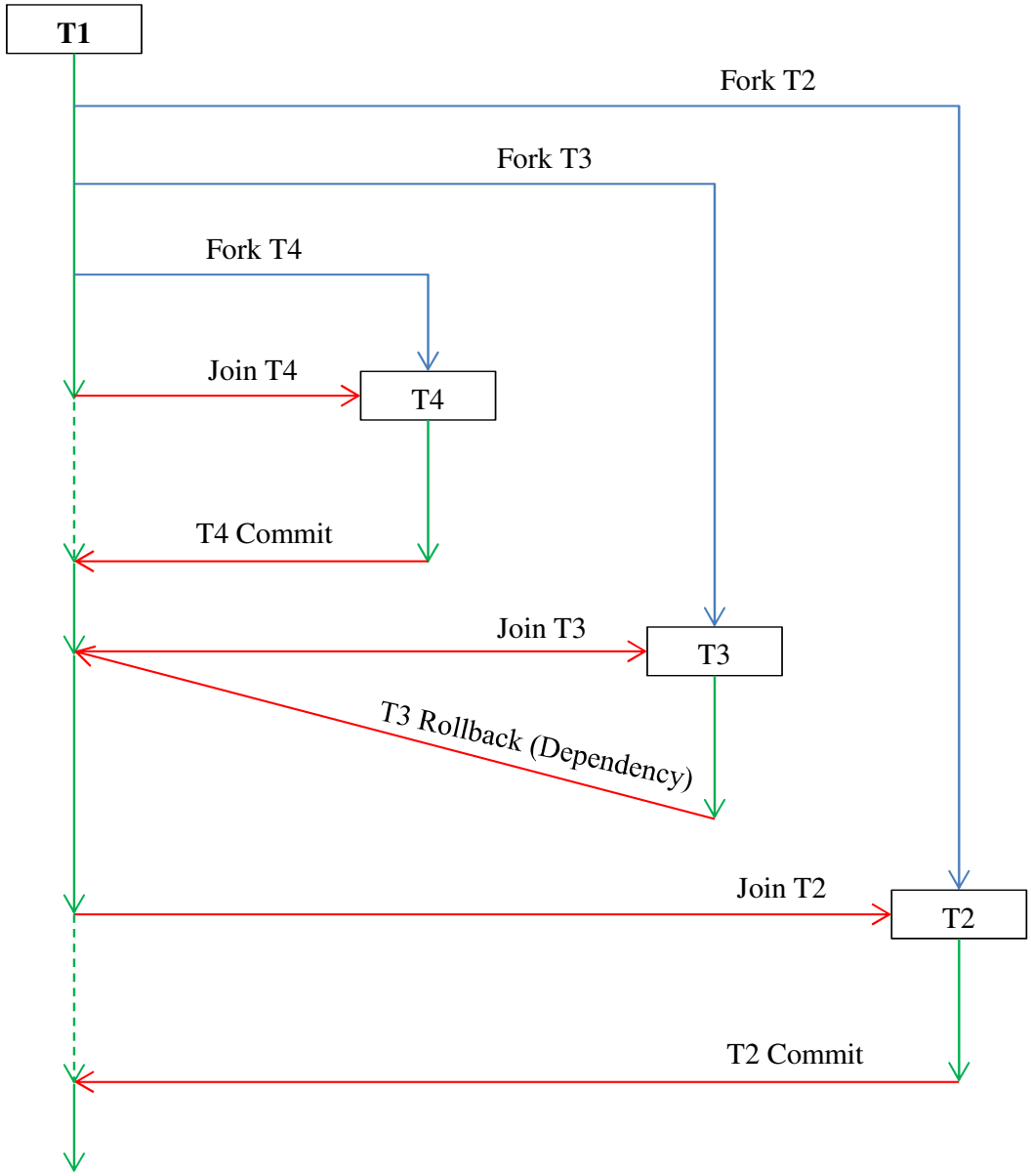


Figure 2–5: Out-of-order Forking Model. The non-speculative thread **T1** forks speculative threads T2, T3 and T4 that represent reverse sequential execution order (i.e. T2 represents latest sequential execution, while T4 represents the earliest), and joins in T4, T3, T2 order. Though T3 rolls back due to dependency, T2 does not cascade rollback and can still commit.

from executed speculatively, bounding parallelism to just two threads irrespective of loop dependencies.

The out-of-order model is illustrated in Figure 2–5. The non-speculative thread T1 first forks a speculative thread T2, and continues execution for a while. Then thread T1 forks a second speculative thread T3 that starts execution from a sequentially earlier join point, as opposed to in-order speculation. Afterward, T1 reaches a third fork point and speculates thread T4 that represents sequentially earliest execution. Since the speculated threads T2, T3 and T4 represent sequential execution that is reverse to their speculated order, they are also joined in the reverse speculation order. For method-level speculation, T2, T3 and T4 are forked at different nested call frames, and thus threads speculated at deeper call frames execute sequentially earlier function call continuation and are joined earlier. As the speculative threads are all children of the non-speculative thread T1, they have no control dependencies among each other, and thus even if a speculative thread (T3) rolls back, the remaining speculative threads (T4) can still commit.

2.3.3 Mixed Forking Model

Mixed model is by far the most powerful forking model. It maximizes parallelism opportunities by allowing all threads to speculate new threads, and thus has the strength of both in-order and out-of-order models. One scenario in which it outperforms in-order and out-of-order models is tree-form recursion, where in-order speculation can only extract the top-level parallelism and out-of-order can only descend into one branch, while a mixed model theoretically can fork a whole tree of threads.

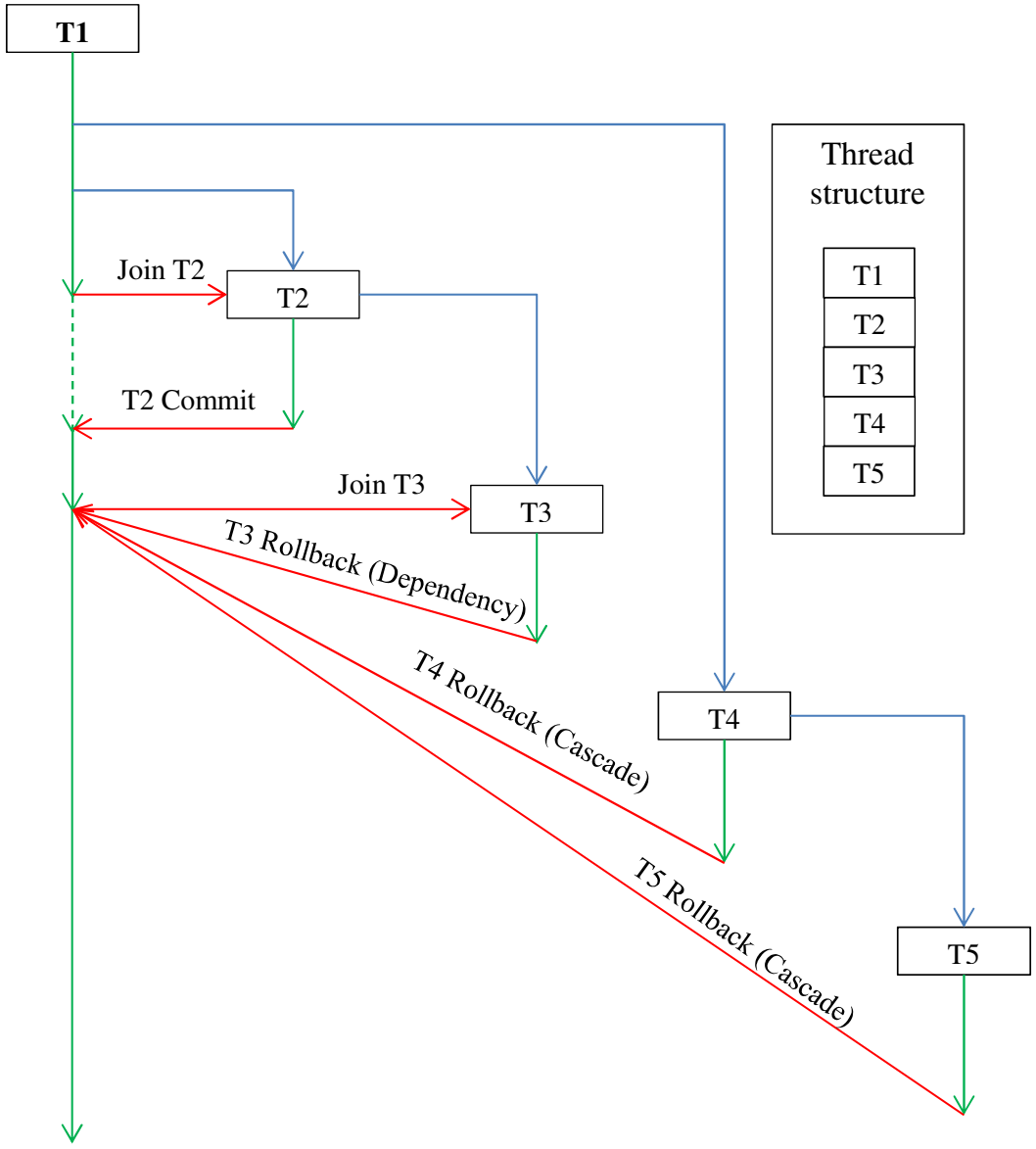


Figure 2–6: Linear-form Mixed Forking Model. The non-speculative thread **T1** out-of-order speculates T4 and T2, which then speculate T5 and T3, respectively. When T3 rolls back due to dependency, T4 and T5 cascade rollback though they are not speculated by T3.

The flexibility of the mixed model also involves different designs. One part is how the system organizes the speculated threads. Previous mixed model systems assigned an order number to each speculative thread and organize them in a simple linear form, as a sequence of execution of the program. This design has a similar disadvantage to the in-order model: if a sequentially earlier thread rolls back, then all subsequent threads roll back even if they present no conflicts, which is not rare since function calls usually indicate independent tasks. An example of linear-form mixed model speculation is shown in Figure 2–6. The non-speculative thread T1 forks speculative threads T4 and T2 at two fork points. Then T4 and T2 speculate their own child threads T5 and T3, respectively. However, since the linear-form mixed forking model simply assigns order numbers to speculative threads and organizes the speculative threads in a linear form as in-order speculation, if any speculative thread rolls back, threads representing sequentially later execution also rollback, even if they have no control dependencies, as is the case for T3, T4 and T5.

The approach we develop here uses a novel mixed model that organizes the threads in a tree-form, and only has cascading rollbacks within its subtree, which is demonstrated in Figure 2–7. The example present the same thread execution scenario as the one in Figure 2–6, except that the thread structure is organized as a tree, with each child thread represented as a child node of the parent thread node. It can be seen that since T4 and T5 are not child nodes of the speculative thread T3, they do not have control dependencies. Therefore, threads T4 and T5 are not affected and can still commit after thread T3 rolls back due to memory access dependencies.

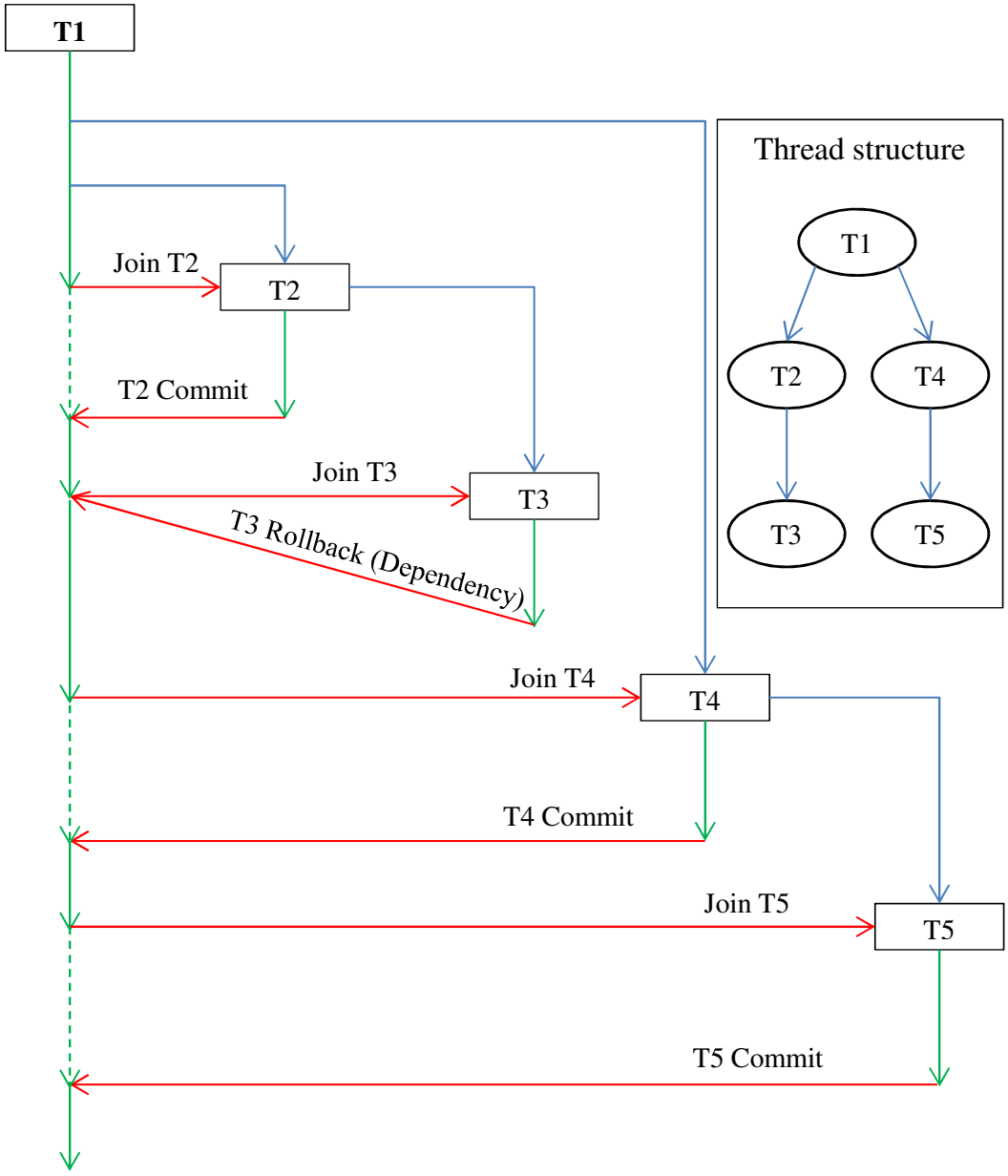


Figure 2–7: Tree-form Mixed Forking Model. The non-speculative thread **T1** out-of-order speculates T4 and T2, which then speculate T5 and T3, respectively. After T3 rolls back due to dependency, T4 and T5 can still commit.

2.4 Memory Buffering

There are different approaches to TLS memory buffering. Garzaran et al. [71] proposed a 2-dimensional taxonomy of hardware-TLS memory buffering approaches. One dimension is separation of speculative task state within a CPU and the other is merging of speculative task state to the main memory. The former deals with the issue of reducing CPU idle time to improve speculative work time coverage in the hardware architecture, which is beyond the scope of the thesis which focuses on software-TLS. The latter has three categories: Eager AMM (Architectural Main Memory), Lazy AMM and FMM (Future Main Memory). AMM buffers speculative state in the CPU and commits it to the main memory after a speculative thread completes execution, while FMM directly accesses the main memory and buffers the memory data, which is used to restore the main memory state if the speculative thread rolls back. Eager AMM commits all buffered data at commit time while Lazy AMM only commits a cache line when another speculative thread uses it again. Lazy and eager version management software-TLS buffering implements Eager AMM and FMM, respectively, while Lazy AMM has not been proposed for software-TLS memory buffering.

2.4.1 Lazy Version Management Buffering

Two mechanisms to implement the lazy version management software-TLS buffering have been proposed, based on there being a non-speculative thread [133, 45] or only speculative ones [148, 124, 179]. Both have their own advantages: the former does not buffer the non-speculative thread and thus can guarantee worst-case run time excluding threading overhead, while the latter buffers memory accesses of

all threads which enables better optimizations for the speculative threads. In our work we use a lazy version management buffering with a non-speculative thread, the architecture of which is illustrated in Figure 2–8.

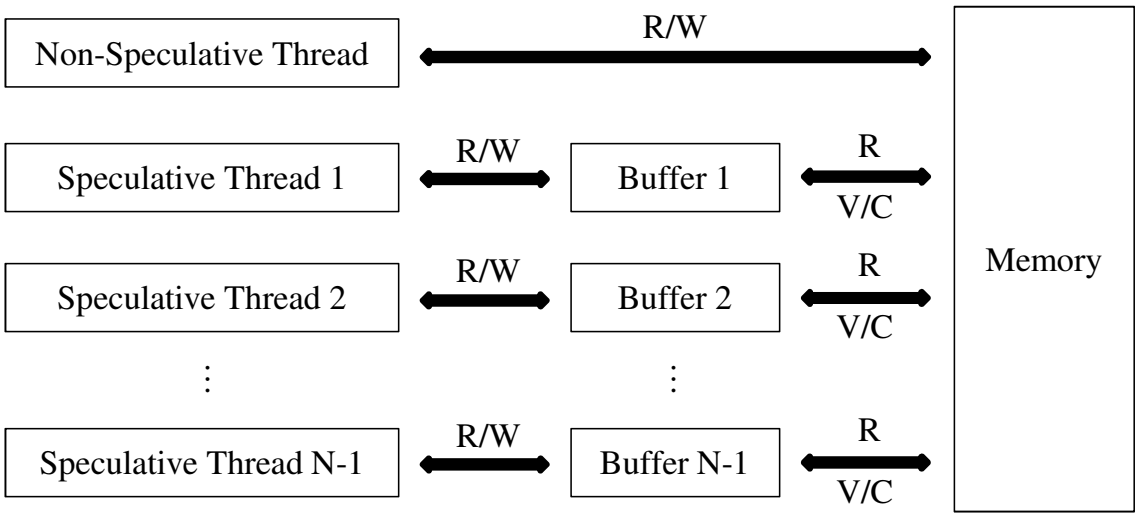


Figure 2–8: Lazy Version Management Software-TLS Buffering with Non-Speculative Thread

There is a non-speculative thread and $N-1$ speculative threads. The non-speculative thread directly reads/writes the main memory, while memory accesses of each speculative thread $t(1 \leqslant t \leqslant N-1)$ are redirected to the thread-private lazy buffer t. If the data read by a speculative thread t is not in the buffer t, it is first read from the main memory to the buffer, and then returned to the speculative thread. When the non-speculative thread joins speculative thread t, the speculative thread validates its buffer t, and then commits the buffer to the main memory if there is no dependency, and discards the buffer otherwise.

The lazy buffer is usually implemented as part of the software-TLS system meta-data, which is a separately allocated region of main memory that is disjoint from the

program data main memory region. The lazy buffer memory region and the program data memory region are thus conceptually referred to as the *buffer* and the *main memory*, respectively. During buffering validation, the speculative thread compares each read value in its buffer with the main memory version, and then commits the buffer if all read values are equal to the main memory version and discards the buffer otherwise. During buffer commit, all written data values in the buffer are copied to the main memory. After buffer commit or if discarded during rollback, the lazy buffer can then be re-initialized for use by the next speculative thread execution.

2.4.2 Eager Version Management Buffering

Existing eager version management buffering mechanisms such as SpLIP [125] and MiniTLS [179] maintain a shadow buffer for each speculative thread, and adds a new version to the buffer each time a variable is written, as the FMM implementation. The architecture is shown in Figure 2–9.

There are N symmetric speculative threads, which read/write the shared load-/store vector and have their own shadow buffer. When a speculative thread t reads/writes a memory address, it first checks the load/store vector; if there is another thread accessing the address and at least one thread writes the address (i.e. the two threads have RAW, WAR or WAW dependencies), then the thread representing sequentially later execution rolls back (other threads may also need cascade rollback, as SpLIP and MiniTLS are loop-level in-order speculation as was discussed in section 2.3), otherwise the speculative thread t registers the memory access in the load/store vector and reads/writes the main memory address. In the case of memory writes, the speculative thread also saves the written address/data pair into its shadow buffer t.

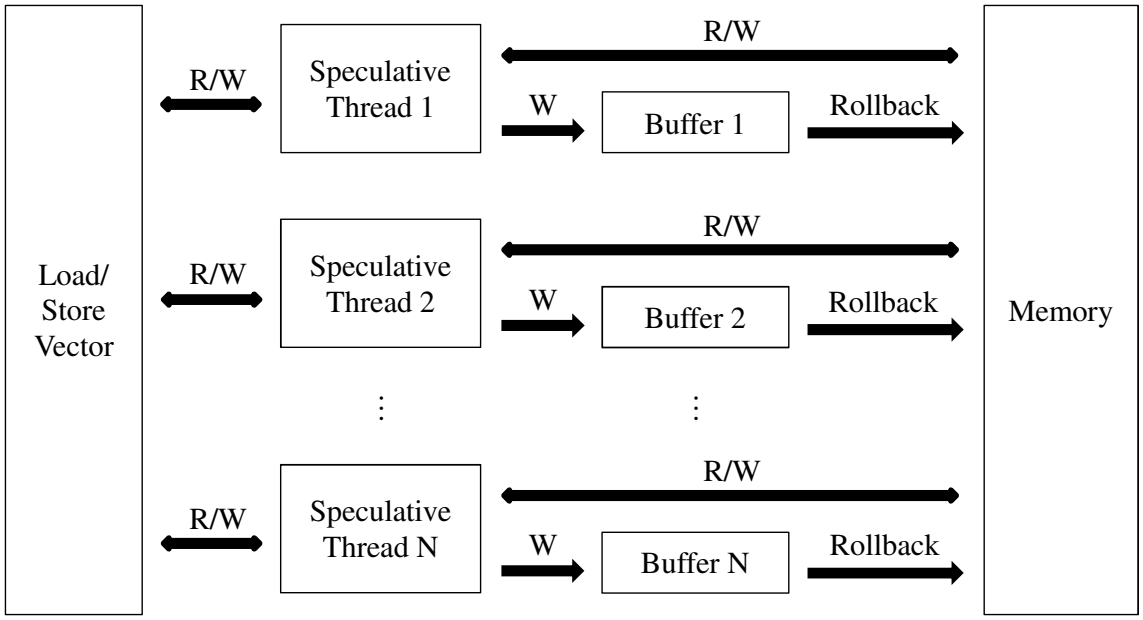


Figure 2–9: Existing Eager Version Management Software-TLS Buffering

When a speculative thread rolls back, it scans each address/data pair in its shadow buffer and restores the data to the main memory address. The load/store vector and the shadow buffer are also usually implemented as part of the software-TLS system metadata as was discussed in subsection 2.4.1.

CHAPTER 3
MUTLS Framework

MUTLS (Mixed-model Universal software Thread-Level Speculation) [45] is a language and architecture independent software-TLS system based on the well-defined LLVM [7] intermediate representation (IR). LLVM is a popular compiler infrastructure with many powerful analysis and transformation passes for program optimization. Since the MUTLS transformation pass is a purely LLVM-IR based pass (from well-formed LLVM IR to well-formed LLVM IR), it is fully integrated into the LLVM compiler framework, which can take advantage of full optimizations as well as all source languages and target architectures enabled by the LLVM framework. With a mixed forking model, MUTLS is also able to exploit more parallelism from tree-form recursion applications.

The MUTLS framework design of this chapter adopts a lazy version management buffering as was discussed in section 2.4.1. MUTLS has two types of threads: a non-speculative thread and speculative threads. The non-speculative thread represents logically earliest execution that never rolls back and is not buffered. Memory accesses of speculative threads are buffered and causes the offending threads to rollback if validation detects RAW dependencies at thread join time. Since speculative threads are usually slower than the non-speculative thread due to buffering cost, checkpoints are inserted in loops and before nested function calls so that a speculative thread can be joined whenever needed, which guarantees that the software-TLS system is

efficient even for memory-intensive applications. This feature also allows MUTLS to be an arbitrary-point speculation system [140, 60, 45] that has more parallelism potential than loop-level speculation and method-level speculation.

The design of the system involves changes to the LLVM back-end and its front-ends. The latter are minor, allowing easy portability of multiple languages. Most of the complexity resides in the back-end, where we require code to support forking, joining, buffering, and commit or rollback.

The front-end annotates *fork/join/barrier points* with LLVM intrinsic functions to specify where to fork/join/barrier speculative threads. The annotation intrinsic functions can be inserted manually by the programmer, or automatically by the compiler, profiler, or other tools. A speculative thread is created at a fork point, starts execution from the corresponding join point, is joined (merged) when the non-speculative thread reaches the join point, and barriered at a corresponding barrier point if the speculative thread reaches it. The back-end comprises an LLVM speculator transformation pass and a TLS runtime library. The runtime library defines application programming interface (API) functions for certain behaviours such as forking/joining threads and buffering loads/stores. The speculator pass transforms the incoming IR based on the annotated fork/join/barrier points and delegates specific speculation behaviours to the TLS runtime library.

This chapter makes the following contributions.

- We describe MUTLS, the first software-TLS implementation on a source language and target architecture independent intermediate representation (IR). Our design is capable of adding TLS features to any LLVM input language

and executes on any architecture supported by LLVM, significantly extending previous TLS systems which only support a single language context and/or architecture.

- We integrate the tree-form mixed forking model proposed in section 2.3 into the MUTLS system, demonstrating that complex mixed fork models can be hosted in a language and architecture independent software-TLS implementation.

3.1 Front-End Design

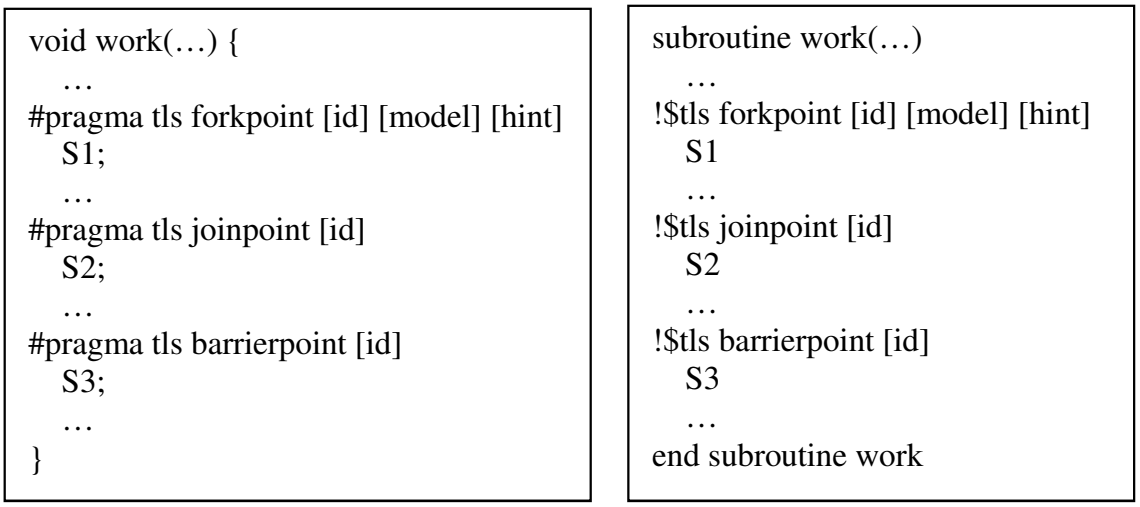
LLVM has two official front-end distributions: Clang and GCC/DragonEgg. The Clang front-end implements C-family languages of C/C++ and objective-C/C++, while the GCC/DragonEgg front-end supports many GCC programming languages such as C/C++ and Fortran. Since MUTLS is designed to support multiple families of programming languages including Fortran, we choose to use the GCC/DragonEgg LLVM front-end.

We add a built-in function `__builtin_forkjoinpoints(type, id, model, hint, arg)` to the GCC front-end for the user to specify fork and join points. In order to avoid unnecessary rollbacks due to thread interference, we also add a *barrier point* which barriers the speculative thread if it is not in nested function calls (i.e. it is at the same stack frame level as was speculated). The argument `type` specifies the type of the point (fork point, join point, barrier point, etc). The `id` denotes the id of the point: threads speculated at a fork point start execution from the join point with the same id. The forking model of the fork point is specified by `model`. The `hint` is used by adaptive fork heuristics and `arg` is point specific argument.

The GCC tree intermediate representation is transformed to the LLVM-IR by the DragonEgg GCC plugin [4]. We add an LLVM intrinsic function `llvm.forkjoinpoints(type, id, model, hint, arg)` to be processed by the LLVM speculator pass. We then transform calls to the GCC built-in function `__builtin_forkjoinpoints` to the `llvm.forkjoinpoints` intrinsic function calls in DragonEgg.

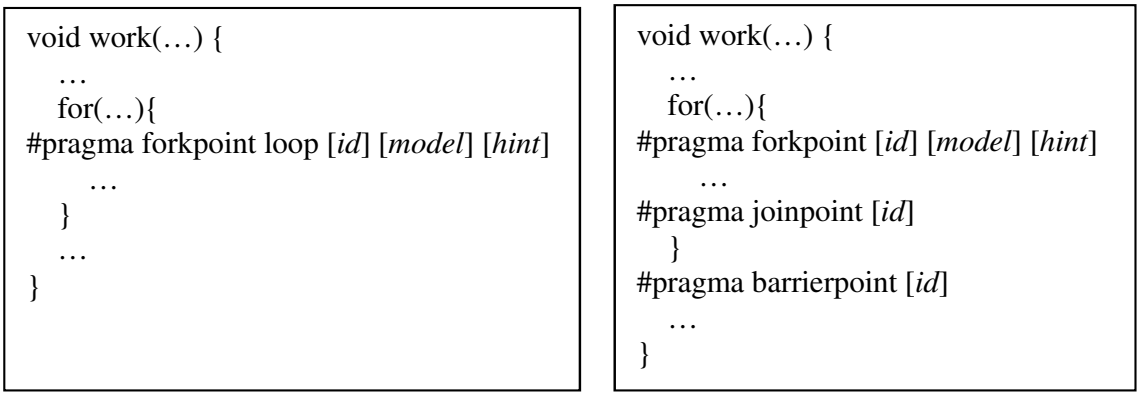
In order for MUTLS to be easier to use, we also add OpenMP-like fork/join/barrier point pragmas to the front-end. The GCC front-end then transforms the pragmas to the `__builtin_forkjoinpoints` builtin function calls. Each pragma can optionally specify an id number using the form "`id` *n*". The fork point pragma can also specify the forking model by "`inorder`|`outoforder`|`mixed`" and an adaptive fork heuristics hint type "`must`|`maybe`". For example, a fork point pragma for C program can be "`#pragma tls forkpoint id 3 maybe`". The default values for id, model and hint are 1, mixed and must, respectively. Examples of input C and Fortran programs are given in Figure 3–1.

Since loops are a common and important type of speculative regions, we specially define two point types for loops: *loop* and *loopblock*. The back-end automatically transforms these points to groups of fork/join/barrier points. The *loopblock* point also performs the "blockize" transformation as will be discussed in section 6.2.1 that statically distributes the workloads as blocks of loop iterations and speculates on the blocks to reduce thread fork/join overhead. A C program with the loop fork point and a group of fork/join/barrier point pragma directives is shown in Figure 3–2.



(a) C program **(b) Fortran program**

Figure 3–1: User-directed TLS source code. Before S1 the parent thread forks a speculative thread to execute S2, and synchronizes with it once it reaches that point. The speculative thread will be barriered before S3 if it reaches that point.



(a) Loop Fork Point **(b) Group of Fork/Join/Barrier Points**

Figure 3–2: MUTLS pragmas for loop speculative regions.

3.1.1 Front-End Implementation

To implement the front-end, we need to add the built-in function `__builtin_fork-joinpoints` that has 5 integer parameters and returns an integer: `DEF_GTLS_BUILTIN`

(BUILT_IN_GTLS_FORKJOINPOINTS, "forkjoinpoints", BT_FN_INT_INT_INT_INT_INT_INT, ATTR_NULL) in the added *gcc/tls-builtins.def* file that is included in *gcc/builtins.def*.

The main work is then to implement the pragmas that automatically generate the built-in function calls. Our approach is to implement the same workflow as the OpenMP implementation. We use the C front-end as as an example; the C++ and Fortran front-end implementations are different but generally follow the same principle.

We first add pragma entries PRAGMA_TLS_FORKPOINT, PRAGMA_TLS_JOINPOINT and PRAGMA_TLS_BARRIERPOINT in enum prgama_kind, and then register to defer the pragma processing to the parser in the preprocessor function init_pragma. The parser then parses each deferred pragma line in the c_parser_pragma function shown in Figure 3–3. The function returns true if it is an OpenMP construct statement such as "#pragma omp for" and false if it is a function call. Since MUTLS pragmas are transformed to built-in function calls to __builtin_forkjoinpoints, they all return false. The c_parser_pragma function first peeks the pragma kind id and then processes the pragma in the corresponding case clause of the switch statement. If the pragma is not processed in the switch statement, then it is an external pragma and the function calls c_invoke_pragma_handler to process the pragma line.

We show the fork point implementation to demonstrate MUTLS pragma processing. In the switch statement of the c_parser_pragma function, the PRAGMA_TLS_FORKPOINT clause first checks the pragma context context to be pragma_compound to ensure the pragma is in a compound statement, and then calls the c_parser_tls_forkpoint function whose excerpt is presented in Figure 3–4. It first sets the pragma option

```
static bool c_parser_pragma (c_parser *parser, enum pragma_context context){
  unsigned int id = c_parser_peek_token (parser)->pragma_kind;
  gcc_assert (id != PRAGMA_NONE);
  switch (id){
    case PRAGMA_TLS_FORKPOINT:
      if (context != pragma_compound){
        if (context == pragma_stmt) c_parser_error (parser, "%<#pragma
          tls forkpoint%> may only be used in compound statements");
        goto bad_stmt;
      }
      c_parser_tls_forkpoint (parser);
      return false;

    case PRAGMA_TLS_JOINPOINT:
      …

    default:
      if (id < PRAGMA_FIRST_EXTERNAL){
        if (context == pragma_external){
        bad_stmt:
          c_parser_error (parser, "expected declaration specifiers");
          c_parser_skip_until_found (parser, CPP_PRAGMA_EOL, NULL);
          return false;
        }
        c_parser_omp_construct (parser);
        return true;
      }
  }
  c_parser_consume_pragma (parser);
  c_invoke_pragma_handler (id);
  …
  return false;
}
```

Figure 3–3: GCC Front-End Parser for Pragmas

```
static void c_parser_tls_forkpoint (c_parser *parser){
   int type = 1000, id = 1, model = 0, hint = 0, arg = 0;
   location_t loc = c_parser_peek_token (parser)->location;
   c_parser_consume_pragma (parser);
   while (c_parser_next_token_is_not (parser, CPP_PRAGMA_EOL)) {
      if(c_parser_next_token_is (parser, CPP_NAME)) {
         const char *p = IDENTIFIER_POINTER (c_parser_peek_token (parser)->value);
         if (strcmp (p, "loop") == 0){
            type = 1100;
            c_parser_consume_token (parser);
         }
         else if (strcmp (p, "id") == 0){
            c_parser_consume_token (parser);
            id = TREE_INT_CST_LOW(c_parser_peek_token (parser)->value);
            c_parser_consume_token (parser);
         }
         …
      }
      else{
         c_parser_error (parser, "%<#pragma tls forkpoint%> expect identifier");
         c_parser_consume_token (parser);
      }
   }
   c_parser_skip_to_pragma_eol (parser);
   add_stmt(get_tls_builtin_call (loc, type, id, order, hint, arg));
}
tree get_tls_builtin_call (location_t loc, int type, int no, int order, int hint, int arg){
   tree x, arglist;
   x = built_in_decls[BUILT_IN_GTLS_FORKJOINPOINTS];
   tree args[5] = { build_int_cst (NULL_TREE, type), build_int_cst (NULL_TREE, no),
      build_int_cst (NULL_TREE, order), build_int_cst (NULL_TREE, hint),
      build_int_cst (NULL_TREE, arg) };
   x = build_call_expr_loc_array (loc, x, 5, args);
   return x;
}
```

Figure 3–4: GCC Front-End Parser for Fork Point Pragmas

variables to the default values. Then in the while loop, it parses each pragma option such as "`loop`", "`id`", "`inorder`" and "`must`" and sets the corresponding variable for the parsed option. After the pragma line is parsed, it calls `get_tls_builtin_call` to create the built-in function call statement and then calls `add_stmt` to add the statement in the tree IR.

3.2 Back-End Overview

Our support for TLS in LLVM consists of two main parts: an LLVM speculator transformation pass, and a TLS runtime library. The LLVM speculator pass modifies the incoming IR based on the annotated fork and join points, delegating more complex behaviours to the MUTLS runtime library.

The architecture of the MUTLS runtime library is shown in Figure 3–5. It provides a set of application programming interface (API) functions that are called by the speculated program generated by the LLVM speculator pass. The runtime library contains 4 layers of classes/functions to implement the API functions, which from top to bottom are the system module layer, support module layer, utility layer and interface layer.

At the top layer, the API functions call the corresponding method of the ThreadManager module to perform the required operations. ThreadManager serves as the top-level manager in charge of implementing the API functions, whose each method invokes one or more of the MUTLS runtime library subsystems: thread data, global buffer and local buffer.

The subsystems are independent of each other so that any subsystem can be redesigned without disturbing others. The thread data subsystem maintains for

each thread a ThreadData object to manage the thread status data. The local buffer subsystem manages individual buffering of local variables for each speculative thread through the per-thread ThreadLocalBuffer objects, while the global buffer subsystem maintains both a SharedGlobalBuffer object for buffering of shared global variables and ThreadGlobalBuffer objects for buffering of thread-private versions of global variables.

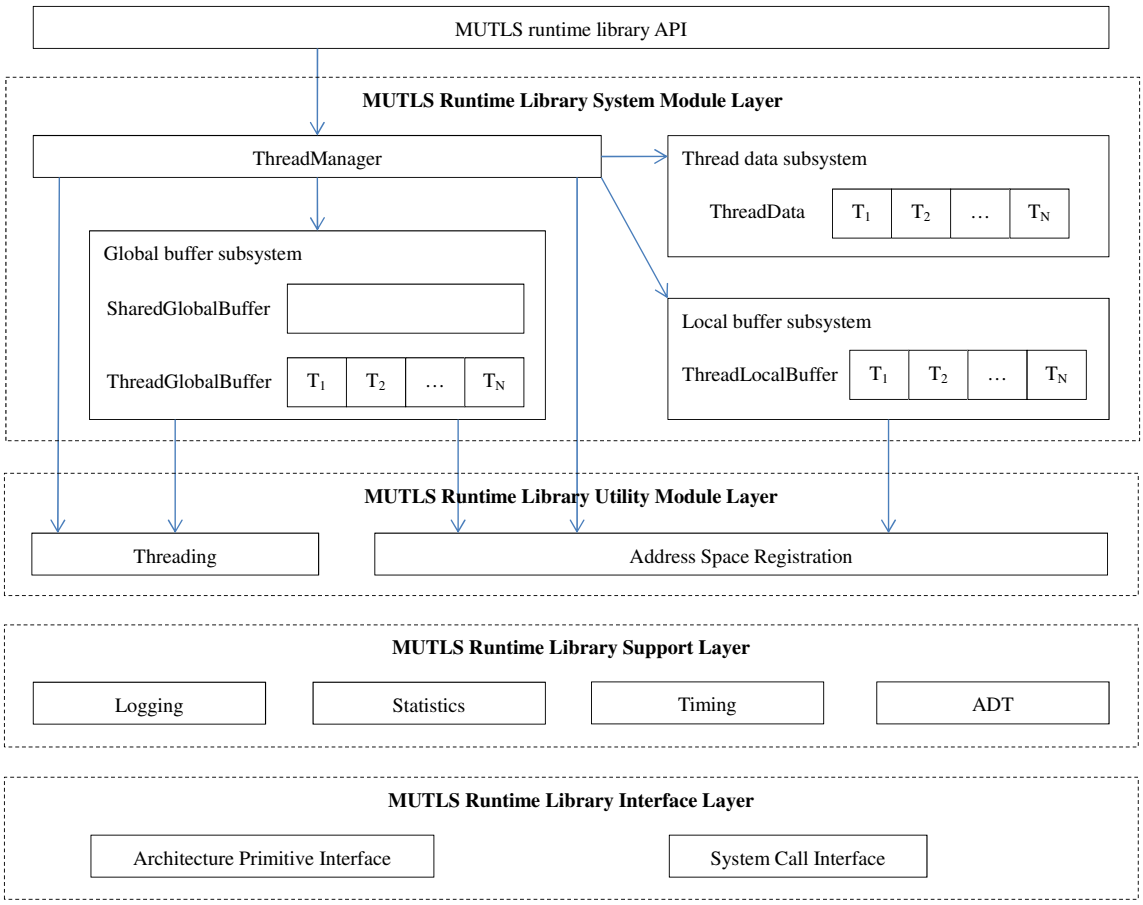


Figure 3–5: MUTLS Runtime Library Architecture

The second layer is the support module layer, which provides support classes to implement well-defined functionality commonly used by various subsystems of the system module layer. The address space registration module is called by the ThreadManager module to register the address spaces, and by the global/local buffer subsystems for address space checking/mapping, as will be discussed in section 3.6. It registers the address spaces (start and end addresses) of global variables and thread stack frames, and finds the corresponding variable or thread stack frame of given addresses during thread execution.

The threading module is used by the ThreadManager module to parallelize the user program, and by the global buffer subsystem for TLS runtime system optimizations such as parallelized validation/commit and parallel shared buffer rollback that will be discussed in sections 5.1.1 and 5.3, respectively. The threading module also provides *virtual CPUs* for the upper system module layer. A virtual CPU is the MUTLS abstraction of an operating system (OS) thread, and is identified by *rank*, which is an ID number from 0 to $P-1$, where P is a configurable parameter denoting the total number of virtual CPUs. It provides for the upper layer a unified interface for accessing the thread resources by abstracting the underlying OS threads as disjoint thread work groups. In this way, it is natural and efficient to support both speculative computation and TLS runtime optimizations. The thread work group class provides the method `run_work(f, id, n)`, which starts a work group of `n` threads to execute the function `f` with the thread number `id` and group size `n` arguments.

The next layer is the utility layer, which provides utility classes for use throughout the runtime library. The logging, statistics and timing utility classes implement message logging, event statistics and timing for various purposes such as debugging, diagnostics and profiling. The implementations allow flexible message format, data, and event types and names, so that they are easy to use in a variety of modules. The bottom layer is the interface layer that abstracts architecture and operating system dependent utilities to provide a standard view of the underlying system functionality such atomic operations, locking primitives and system CPU and memory resources.

The MUTLS runtime library is written in C++, and can be compiled into native static or dynamic libraries for linking into any executable. However, to enable further optimizations provided by LLVM such as inlining of library calls, we compile it into a bytecode library to link with the speculated LLVM bytecode of the source program.

The following sections first discuss speculative thread state transition and the forking models, then describe the transformations of the LLVM-IR for TLS execution and give further details on memory management. Finally we discuss two optimizations to reduce the stack frame maintenance overhead and increase thread work coverage, respectively.

3.3 State Transition

The speculative thread state transition of the baseline MUTLS framework design is illustrated in Figure 3–6. For simplicity, the baseline design binds each thread to a corresponding virtual CPU (OS thread, as was discussed in section 3.2). Each virtual CPU can be BUSY or IDLE, and is initialized IDLE at the beginning of program execution. If a CPU is IDLE, the corresponding speculative thread does not exist;

only when the CPU is BUSY can a speculative thread with the corresponding rank be created and scheduled to the CPU for running. This design has the advantage to separate the ThreadData abstraction from the underlying threading implementation, and enable simpler implementation of thread coverage optimizations.

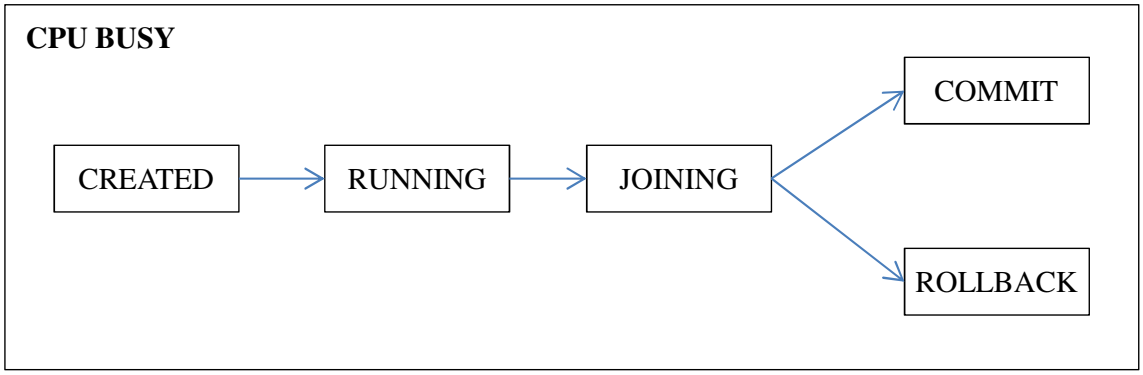


Figure 3–6: Speculative Thread State Transition - Baseline MUTLS Framework Design

Efficient thread polling and synchronization during the JOINING state is performed using a simple flag-based barrier. The ThreadData object for each speculative thread maintains two volatile variables: `sync_status` which is set to JOINING by the non-speculative thread if the speculative thread is notified to synchronize, and `valid_status` which is set to COMMIT or ROLLBACK by the speculative thread after the global buffer validation and commit/rollback. The two variables are respectively initialized to RUNNING and NULL during the fork process. When reaching a join point, the non-speculative thread locates the corresponding ThreadData object as will be discussed in section 3.4 below. It then sets `sync_status` to JOINING and busy-waits for the `valid_status` to be non-NULL. The speculative thread busy-waits for `sync_status` to be JOINING if it enters a terminate point, or simply returns if

it enters a check point and encounters a RUNNING. As will be seen in section 4.2, the checking overhead in check points is small, even for the 3x+1 benchmark whose inner loop iteration has few instructions. Once both threads are synchronized the speculative thread can validate and commit/rollback. After the speculative thread completes commit/rollback, the lifetime of the thread ends and the virtual CPU is set to IDLE.

3.4 Forking Model

MUTLS is a tree-form mixed forking model software-TLS system. However, it also natively supports in-order and out-of-order forking models since they are restricted forms of the mixed forking model. This section mainly describes how the tree-form mixed model is implemented in MUTLS. Other forking model implementations just add checking criteria when a thread tries to speculate a child thread in the `MUTLS_get_CPU` library call that will be discussed in section 3.5.2.

The mixed forking model assumes that the direct children of a thread follow the out-of-order model; that is, a later speculated thread represents logically earlier sequential execution. It also assumes that a thread subtree represents a continuous interval of execution with its root representing the earliest logical execution, and that different thread subtrees represent disjoint intervals of execution. As a result, a "reverse in-order traversal" of the thread tree follows the sequential execution order. This assumption is similar to previous work [176, 101]. However, unlike their approaches, the runtime system does not rely on the mixed-model assumption to be correct. As a result, the compiler can enable mixed forking model speculation at any fork point without having to guarantee that the program control flow satisfies

the mixed order requirement. Mixed model can thus be selected based on heuristics, profiling or programmer directive hints instead of relying on the compiler to prove the mixed order control flow structure in different function call chains.

Each thread maintains a *children* stack storing the ranks of its direct children, as well as a *ranks* array for each function call stack frame storing the speculated thread ranks in the current stack frame. When forking a thread, it pushes the new thread rank into *children*. In the `MUTLS_synchronize` library call that will be discussed in section 3.5.3, it pops a child rank from *children* and checks if it is equal to the corresponding rank stored in the *ranks* array that stores the speculated child thread ranks in the current stack frame. If it is not, then it means the program did not follow the mixed-model assumption, i.e. not joining the last speculated child thread, which can happen due to erroneous control flow or misspecified join points. In this case, it sets the `sync_status` of the child thread to be NOSYNC and the process continues until the rank is found or *children* is empty. In either case, the corresponding entry in *ranks* is set to 0 to allow speculation on that point again. If *children* is empty, the child thread has already rolled back and false is returned. Otherwise, it appends the *children* of the child thread to the *children* of the non-speculative thread, synchronizes with the child thread and returns the rank in the argument. Note that even if the child thread is invalid and requires rollback, other speculative threads are still preserved in the *children* of the non-speculative thread. This process is different from previous software approaches, having the advantage that local conflicts do not incur global rollbacks.

3.5 Back-End Transformation

Target dependent information such as access to the actual instruction pointer (IP) and stack frame are generally not available in LLVM-IR, nor are some low-level operations such as the ability to directly jump to another function. Maintaining the LLVM SSA-form after each transformation is also required. These issues make thread-level speculation for LLVM more difficult. We perform a number of IR-based code transformations to support TLS. The following subsections discuss the various steps involved in modifying the LLVM-IR for TLS execution. We first present the basic code preparation, then trace through the implementation design from fork to join, and describe a stack frame reconstruction mechanism to implement joining of speculative threads into nested function calls.

3.5.1 Preparation Transformation

The LLVM speculator pass transforms each function with annotation of fork and join points, as well as their nested function calls since we allow speculative threads to enter nested function calls. For each such function we perform 4 basic modifications to prepare for speculation. We need to (1) generate a speculative version of the function, (2) generate helper functions for interaction with the TLS runtime library, (3) split and number the basic blocks at appropriate points for synchronization between the speculative and non-speculative versions of the function, and (4) assign local buffering addresses (offsets from stack pointer) for local variables. Below we detail these steps.

(1) We first clone the function and add two integer parameters *counter* and *rank* to generate the *speculative function*. These arguments are used to direct the

code entry to the correct starting point and track the CPU index, respectively. To ensure safety, every load and store operation within the function is replaced by an MUTLS runtime library API function call to `MUTLS_load_{int32, int64,` etc} and `MUTLS_store_{int32, int64,` etc}. Note that since we inline the TLS runtime library into the source LLVM code, these function calls are reduced to more efficient direct memory accesses.

That we generate a separate speculative version instead of using one single function is for performance optimization and practical implementation: the speculative version is buffered while the non-speculative version is not. Also, since the speculative version has additional parameters *counter* and *rank* to support TLS, function calls to the speculative version require different call arguments from those to the non-speculative version. Other parallelization systems such as OpenMP also adopt the separate function design.

(2) We generate a *stub* function with the suffix ".stub" as the entry point for speculative threads. This prologue function fetches arguments of the speculative function through the `MUTLS_get_regvar_{int32, ptr,` etc} library calls, and then calls the speculative function with the arguments. These arguments are stored by the parent thread in a generated *proxy* function, which has the same signature as the speculative function and stores the function arguments to the ThreadLocalBuffer object by the `MUTLS_set_regvar_{int32, ptr,` etc} library calls. The proxy function then calls the `MUTLS_speculate` library function with the stub function address to fork a new thread.

The stub function is needed since thread entry functions should have the same function prototype (function parameter and return types), while speculative versions do not. The proxy function is for convenient implementation and IR visualization, as calling the proxy function is more natural than its implementation.

(3) Speculative termination and synchronization require a number of checkpoints to be inserted into the code. At each annotated fork point the basic block is split to generate a *speculation block*, at each join point annotation the basic block is split to generate a *join point block*, and at each barrier point annotation the basic block is split generate a *barrier point block*. Speculation is necessarily terminated at each external, indirect and exception handling function call through a *terminate point block* (other than for known, safe external calls such as `abs`, `exp`, etc), and prior to the method return point through a *return point block*. Before each internal function call, a basic block is split to generate an *enter point block*, and inside each loop a block is split to generate a *check point block*. Except for the speculation block, all these blocks are potential *synchronization blocks*, and are numbered by the *synchronization counter* starting from 1, which is then used to build the speculation and synchronization tables, as will be discussed in subsections 3.5.2 and 3.5.3.

Though we allow speculative threads to enter nested function calls in the MUTLS system, for ease of implementation we do not allow them to exit the speculative entry function (the stack frame at which the thread was speculated), as illustrated in Figure 3–7. Therefore, we need to split the return point block to check whether the speculative function is at the bottom speculative stack frame, and if so barrier the speculative thread.

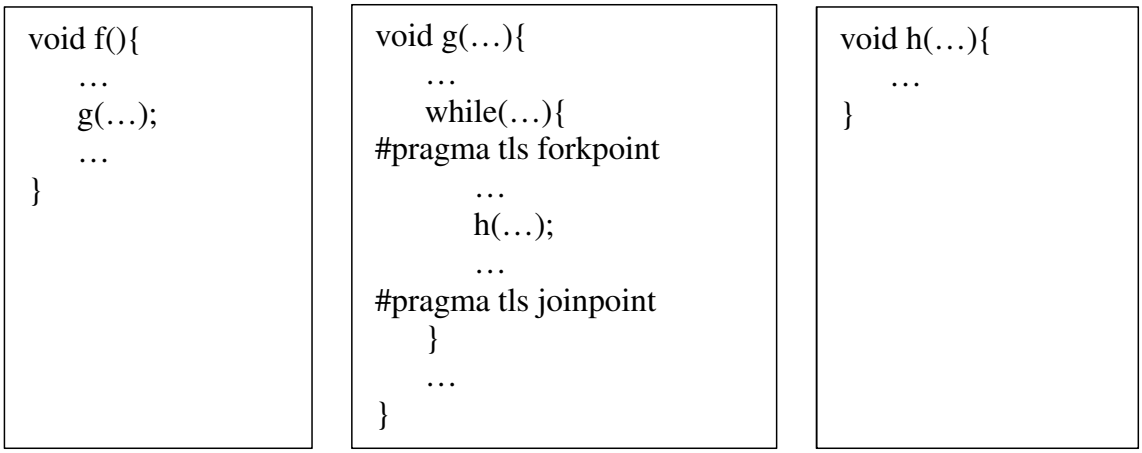


Figure 3–7: The non-speculative thread enters function `g` from function `f` and then speculates a child thread in function `g`. The child speculative thread can enter and return from the speculative version of function `h`, but cannot return from function `g` and execute function `f`.

That we need to split the speculation/synchronization blocks is because during thread forking/joining, the speculative/non-speculative thread needs to jump to the exact execution point of the parent/child thread. For the same reason, we number the speculation and synchronization blocks so that a block in a speculative function corresponds to the same execution point in the non-speculative function if the two blocks have the same speculation or synchronization number. Since we barrier the speculative thread at each barrier point annotation to avoid unnecessary rollbacks, barrier points are also potential synchronization points. We note that check points and barrier points are for performance optimization, but not correctness requirement of the software-TLS system, though check points can also avoid potential infinite loops in speculative threads due to misspeculation, as was analyzed in detail by García-Yágüez et al. [69].

(4) Registers cannot be used to transfer data between threads. Thus for each local (register and stack) variable live at the beginning of a synchronization block of the function, we allocate and assign an offset so we can save and restore such data through the runtime library at speculation and synchronization points during thread forking and joining, as will be discussed in subsections 3.5.2 and 3.5.3.

```
int work(args){
   S1;
   for(i = 0; i < n; i++){
#pragma tls forkpoint id 1
      S2;
      a[i] = abs(b[i]);
#pragma tls joinpoint id 1
   }
#pragma tls barrierpoint id 1
   S3;
#pragma tls forkpoint id 2
   S4;
   s = sum(n, a);
#pragma tls joinpoint id 2
   S5;
   printf("%d\n", s);
   S6;
}
```

(a) Original version

```
void work.proxy(args, counter, rank){
   MUTLS_save_local(args, rank)
   MUTLS_speculate(work.stub, counter, rank);
}
void work.stub(rank){
   args = MUTLS_restore_local(rank)
   counter = MUTLS_restore_local(rank)
   work.speculative(args, counter, rank);
}
```

(b) Proxy and stub functions

```
int work.speculative(args, counter, rank){
   S1;
   for(i = 0; i < n; i++){
      // check point block, synchronization block 3
      // speculation block 1
      S2;
      tmp = MUTLS_load_int32(&b[i], rank);
      MUTLS_store_int32(&a[i], abs(tmp), rank);
      // join point block 1, synchronization block 1
   }
   // barrier point block 1, synchronization block 4
   S3;
   // speculation block 2
   S4;
   // enter point block, synchronization block 5
   s = sum(n, a);
   // join point block 2, synchronization block 2
   S5;
   // terminate point block, synchronization block 6
   printf("%d\n", s);
   S6;
   // return point block, synchronization block 7
}
```

(c) Speculative version

Statement markers
MUTLS library calls
MUTLS local variable library calls
MUTLS speculator pass generated functions
MUTLS speculator pass split blocks

Figure 3–8: Preparation Transformations performed by the Speculator Pass

A schematic of the preparation transformation is given in Figure 3–8. We use C pseudo code instead of LLVM assembly for compactness. The program in Figure 3–8(a) has a loop fork/join/barrier point group (id 1) and a method fork/join point group (id 2). It calls a safe external function `abs` that can be ignored by the MUTLS system, a nested function `sum` that needs to be considered a synchronization point, and an unsafe external function `printf` that should terminate speculation. A function call is safe if it does not write global/argument memory and has no side-effect (no I/O access, no fault/trap and not throwing exceptions), and unsafe otherwise.

The speculative version in Figure 3–8(c) demonstrates replacement of memory reads/writes with `MUTLS_load/store_*` runtime library function calls as discussed in the preparation transformation step (1), as well as the split and numbered basic blocks discussed in the step (3). It generated 2 speculation blocks (one for each fork point) and 7 synchronization blocks (two join point blocks and one each for check, barrier, enter, terminate and return point block). The two speculation blocks and the two join point blocks are split at the two fork point annotations and join point annotations, respectively. We split a check point block in the loop header block, a barrier point block at the barrier point annotation, an enter point block before the `sum` internal function call, a terminate point block before the unsafe `printf` external function call and a return point block before the function return instruction. We number the synchronization blocks in the order of join, check, barrier, enter, terminate and return point, and assign the synchronization counter to each synchronization blocks. The speculation and synchronization counters of the speculation and synchronization blocks are maintained in the speculator transformation pass object, and will be used

to build the speculation table and synchronization table to be discussed in sections 3.5.2 and 3.5.3, respectively.

The following sections will discuss the transformations in detail.

3.5.2 Fork

Since each function may contain multiple fork/join points, the speculator transformation pass allocates an integer array *ranks* on the stack to store the ranks of the child threads speculated in the current function frame. The length n_{ranks} of *ranks* is the number of join points in the function. At most one thread can be speculated on at each fork/join point id; if a fork/join point id is not speculated on, the corresponding entry in *ranks* is 0. In this way, we can choose to speculate on a fork point in some but not all of the call stack frames of the same function, enabling the ability to effectively utilize CPU resources to extract substantial parallelism and reduce overhead from recursive function calls. For example, since tree-form recursion programs usually perform more work on higher-level recursive calls, we can apply the selected speculation enabling optimization that enables speculation in the higher-level call frames but disables it in deeper levels to make the speculative parallelization more efficient, as shown in Figure 3–9.

At each fork point, the LLVM speculator pass generates a library call `MUTLS_get_CPU` to assign a rank to the speculative thread, passing the *ranks* array, the forking model, fork/join point id and the thread rank. If no CPU is IDLE (not running a speculative thread, as was discussed in section 3.3), speculation will not be performed; otherwise the child rank is stored in the corresponding entry of *ranks* and control is branched to the speculation block. The speculation block speculates the local

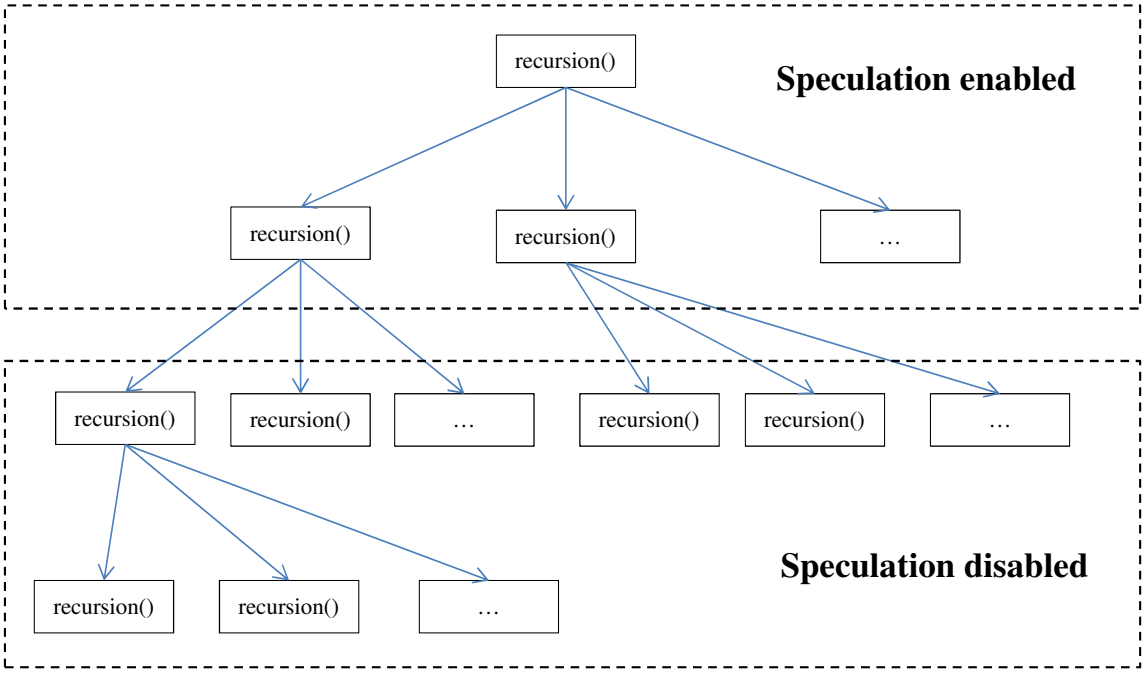


Figure 3–9: Selected Speculation Enabling. MUTLS can enable/disable speculation in different stack frames.

variables live at the beginning of the join point block as will be discussed in section 3.6.4, and then forks a speculative thread by calling the proxy function generated in preparation step (2). After saving the function arguments to the runtime library, the proxy function calls `MUTLS_speculate` to initialize the ThreadData object and sets the CPU state as BUSY. The resulting speculative thread retrieves the arguments within the stub function, and then enters the actual speculative code.

The speculative entry point is conceptually somewhere within a function, but since LLVM does not allow branching directly to this starting point some gymnastics are performed to redirect entry control flow. For this we use the *speculation table*, implemented as a *switch* LLVM instruction that directs incoming control flow to the

block indicated by the *counter* argument. A 0-counter indicates normal entry, while a non-0 value indicates some internal starting point. This approach bypasses the inability of LLVM to branch directly into a function, and also allows both initial and any subsequent (such as recursive) calls to the speculative function to coexist. Upon initial entry, local variables need to be initialized to the same values found in the non-speculative function at forking. For this we fetch the values previously stored within the runtime library and assign them to the corresponding local variables. This process is slightly complicated by the fact that LLVM IR is in SSA form, and so trivially re-assigning register variables is not possible. We thus add a separate *restore* block to assign the local values, and then branch into the actual entry point. Phi nodes are inserted at the beginning of the latter block to distinguish the different versions of the register variables. The program in Figure 3–8 after the fork transformation is presented in Figure 3–10.

3.5.3 Join

At each join point, the LLVM speculator pass adds instructions to check if a thread was speculated on the join point; if so it synchronizes with the speculative thread. This process is encapsulated by `MUTLS_synchronize`, which returns true/false if the speculative thread commits/rollbacks. If true is returned, the synchronization counter and the speculative child thread rank are returned by the arguments c and r, respectively, and control branches to the *synchronization table*. The synchronization table itself is a *switch* LLVM instruction that branches to the code blocks indexed by the synchronization counter. Unlike the speculation table which has $n_{ranks}+1$ entries, the synchronization table has an entry for each possible synchronization block.

```
int work(args){
  int ranks[2]={0};
  S1;
  for(i = 0; i < n; i++){
    MUTLS_get_CPU(ranks, 0, mixed, must, 0);
    if(ranks[0] > 0){
      MUTLS_speculate_local
      work.proxy(args, 1, rank);
    }
    S2;
    a[i] = abs(b[i]);
  }
  S3;
  MUTLS_get_CPU(ranks, 1, mixed, must, 0);
  if(ranks[1] > 0){
    MUTLS_speculate_local
    work.proxy(args, 2, rank);
  }
  S4;
  s = sum(n, a);
  S5;
  printf("%d\n", s);
  S6;
}
```

(a) Non-speculative version

Statement markers
MUTLS library calls
MUTLS local variable library calls
MUTLS speculator pass generated functions
MUTLS speculator pass added blocks

```
int work.speculative(args, counter, rank){
  int ranks[2]={0};
  switch(counter){
    case 0: break;
    case 1: goto join point 1 restore block
    case 2: goto join point 2 restore block
  }
  S1;
  for(i = 0; i < n; i++){
    MUTLS_get_CPU(ranks, 0, mixed, must, rank);
    if(ranks[0] > 0){
      MUTLS_speculate_local
      work.proxy(args, 1, rank);
    }
    S2;
    tmp = MUTLS_load_int32(&b[i], rank);
    MUTLS_store_int32(&a[i], abs(tmp), rank);
  }
  S3;
  MUTLS_get_CPU(ranks, 1, mixed, must, rank);
  if(ranks[1] > 0){
    MUTLS_speculate_local
    work.proxy(args, 2, rank);
  }
  S4;
  s = sum(n, a);
  S5;
  printf("%d\n", s);
  S6;
// restore blocks for join point blocks
}
```

(b) Speculative version

Figure 3–10: Fork Transformation performed by the Speculator Pass

A speculative thread needs to terminate at a synchronization point if it may execute instructions unsafe to perform speculatively, reaches a speculated join point, reaches a barrier point whose corresponding fork/join point has been speculated on at the bottom call stack frame, or if the parent thread is waiting to join with it. The first three cases are enforced by adding a no-return runtime library call `MUTLS_terminate_point`, `MUTLS_sync_parent`, `MUTLS_barrier_point` at each terminate

point block, join point block and barrier point block, respectively. The last case, however, requires polling to determine whether or not the speculative thread needs to stop. This is implemented through calls to `MUTLS_check_point`, which are inserted prior to internal function calls and within inner loops to ensure that the non-speculative thread need not wait overly long. In either case, if validation fails during synchronization, the speculative thread rolls back within these functions; otherwise, it commits and passes in its synchronization counter and rank to indicate its continuation point and to identify itself. Live local variables needs to be saved in order for the non-speculative thread to restore the values after committing. For performance, this is not done before the `MUTLS_check_point` as check points are entered frequently. Instead, the speculator pass adds a *commit block* at each check point, which saves the local variables and calls `MUTLS_commit` to complete the commit process. The program in Figure 3–10 after the join transformation is illustrated in Figure 3–11.

3.5.4 Stack Frame Reconstruction

For simplicity we currently restrict speculative threads from returning from their entry function, but do allow them to call and enter new functions, as was discussed for Figure 3–7. Even this is non-trivial, however, since the non-speculative parent thread may then need to reconstruct equivalent stack frames as part of a successful join, but stack frames are not available at the LLVM-IR level, and may also differ in layout due to the extra parameters added to speculative versions.

We propose a stack frame reconstruction scheme to address this problem. First, we need to explicitly track stack frames as the speculative thread descends into a call chain. At each enter point block, `MUTLS_enter_point` is called to register a new

```
int work(args){
  int ranks[2]={0}, c, r;
  S1;
  for(i = 0; i < n; i++){
    MUTLS_get_CPU(ranks, 0, mixed, must, 0);
    if(ranks[0] > 0){
      MUTLS_speculate_local
      work.proxy(args, 1, rank);
    }
    S2;
    a[i] = abs(b[i]);
    if(ranks[0] > 0){
      MUTLS_validate_local
      if(MUTLS_synchronize(ranks, 0, &c, &r)){
        goto synchronization_table;
      }
    }
  }
  S3;
  MUTLS_get_CPU(ranks, 1, mixed, must, 0);
  if(ranks[1] > 0){
    MUTLS_speculate_local
    work.proxy(args, 2, rank);
  }
  S4;
  s = sum(n, a);
  if(ranks[1] > 0){
    MUTLS_validate_local
    if(MUTLS_synchronize(ranks, 1, &c, &r)){
      goto synchronization_table;
    }
  }
  S5;
  printf("%d\n", s);
  S6;
synchronization_table:
  switch(c){
    case 1: goto synchronization block 1 restore block
    ...
    case 7: goto synchronization block 7 restore block
  }
// restore blocks for synchronization blocks
}
```

(a) Non-speculative version

Statement markers

MUTLS library calls

MUTLS local variable library calls

MUTLS speculator pass generated functions

MUTLS speculator pass added blocks

```
int work.speculative(args, counter, rank){
  int ranks[2]={0};
  switch(counter){
    case 0: break;
    case 1: goto join point 1 restore block
    case 2: goto join point 2 restore block
  }
  S1;
  for(i = 0; i < n; i++){
    if(MUTLS_check_point(3, rank)){
      MUTLS_save_local
      MUTLS_commit(rank);
    }
    MUTLS_get_CPU(ranks, 0, mixed, must, rank);
    if(ranks[0] > 0){
      MUTLS_speculate_local
      work.proxy(args, 1, rank);
    }
    S2;
    tmp = MUTLS_load_int32(&b[i], rank);
    MUTLS_store_int32(&a[i], abs(tmp), rank);
    if(ranks[0] > 0){
      MUTLS_validate_local
      MUTLS_save_local
      MUTLS_sync_parent(1, rank);
    }
  }
  if(ranks[0] > 0 && counter > 0){
    MUTLS_save_local
    MUTLS_barrier_point(4, rank);
  }
  S3;
  MUTLS_get_CPU(ranks, 1, mixed, must, rank);
  if(ranks[1] > 0){
    MUTLS_speculate_local
    work.proxy(args, 2, rank);
  }
  S4;
  s = sum(n, a);
  if(ranks[1] > 0){
    MUTLS_validate_local
    MUTLS_save_local
    MUTLS_sync_parent(2, rank);
  }
  S5;
  MUTLS_save_local
  MUTLS_terminate_point(6, rank);
  printf("%d\n", s);
  S6;
// restore blocks for join point blocks
}
```

(b) Speculative version

Figure 3–11: Join Transformation performed by the Speculator Pass

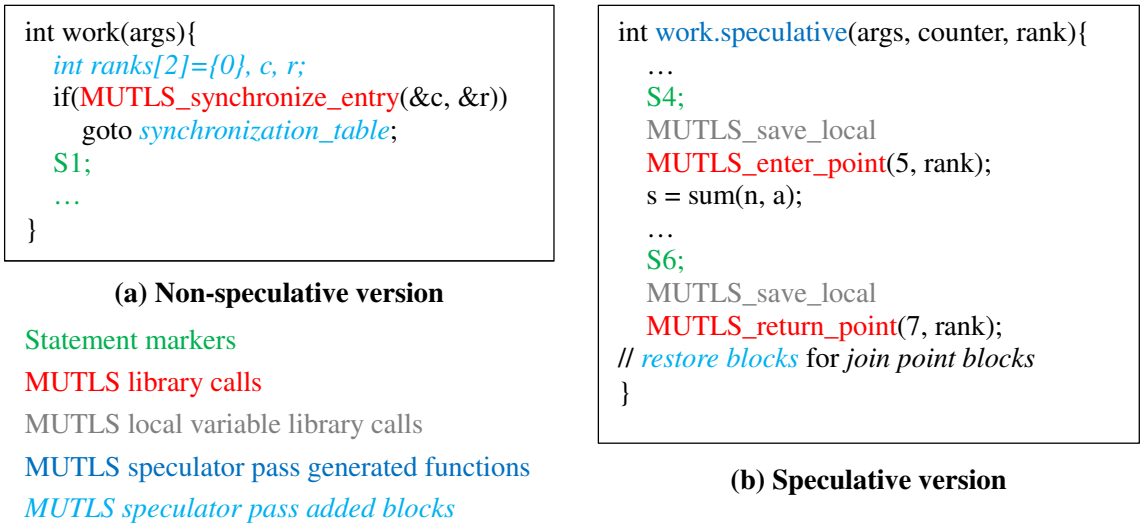


Figure 3–12: Stack Frame Reconstruction Transformation performed by the Speculator Pass

stack frame in the ThreadLocalBuffer object for the nested function call. This is matched at each return point block, where `MUTLS_return_point` is called to pop the stack frame. (Note that this checks to ensure the speculative thread is returning from a nested function call, and not its entry point.) The non-speculative parent must then generate a corresponding call chain, restoring frame data as it descends. This is initiated by the synchronization process, and fully enabled by a library call `MUTLS_synchronize_entry` inserted at the top of each non-speculative function reachable from a speculative one. This function inspects the ThreadLocalBuffer object, recognizes the existence of further frames, and if so restores the current frame data and directs the thread to the correct call point in the current function. This process continues until the non-speculative thread has replicated the entire call chain and

frame state. The program in Figure 3–11 after the stack frame reconstruction transformation is shown in in Figure 3–12. In this figure, common parts with Figure 3–11 are omitted with ellipsis (...).

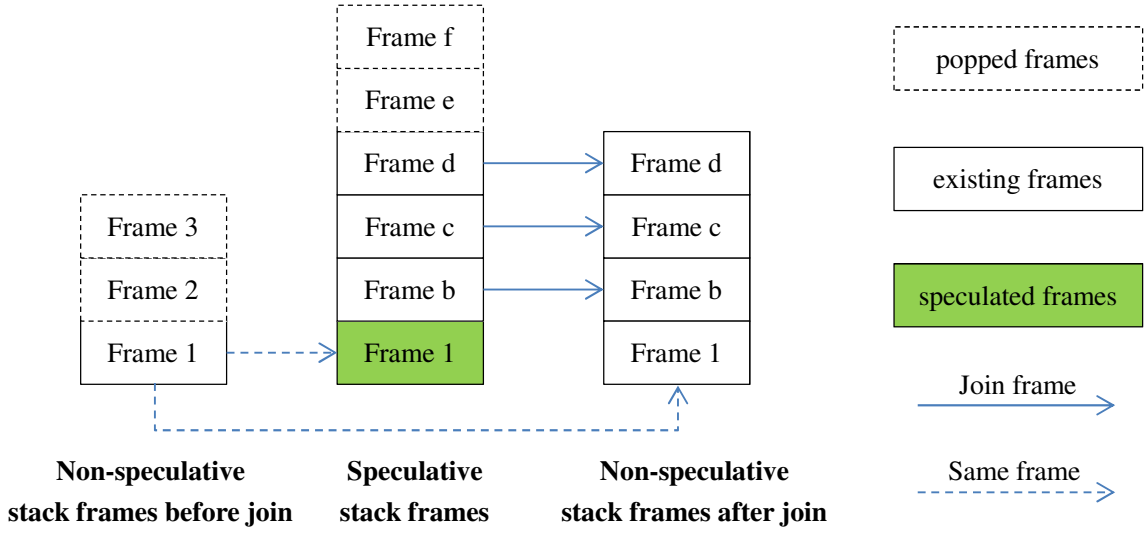


Figure 3–13: Stack Frame Reconstruction Running Example. Frame numbers 1-3 are the non-speculative stack frames and b-f are the speculative stack frames.

An example of the stack frame reconstruction scheme is shown in Figure 3–13. The non-speculative thread forked a speculative thread, and then joined the speculative thread. After forking, the non-speculative thread entered and then returned from frames 2 and 3, and then begins to join the speculative thread. During this time, the speculative thread entered frames b to f, calling `MUTLS_enter_point` each time entering a new frame, and returned from frames e and f, calling `MUTLS_return_point` each time returning from a frame. Then the speculative thread commits and the non-speculative thread merges the stack frames b to d of the speculative thread, calling `MUTLS_synchronize_entry` and jumping to the restore blocks of the corresponding

synchronization blocks each time it enters a new function call frame. Then the non-speculative thread continues execution from the synchronization point block where the speculative thread commits.

3.6 Memory Buffering

The global buffer subsystem buffers variables that multiple threads may read-/write, which include static and heap variables. Stack variables of the non-speculative thread are also buffered in the global buffer subsystem since the non-speculative and speculative threads may simultaneously read/write the stack variables of the non-speculative thread. On the other hand, register and stack variables of speculative threads are buffered in the local buffer subsystem, since only the speculative thread can access its register/stack variables. We thus define static, heap and non-speculative stack variables as *non-local* variables, and register and stack variables of speculative threads as *local* variables, as shown in Figure 3–14.

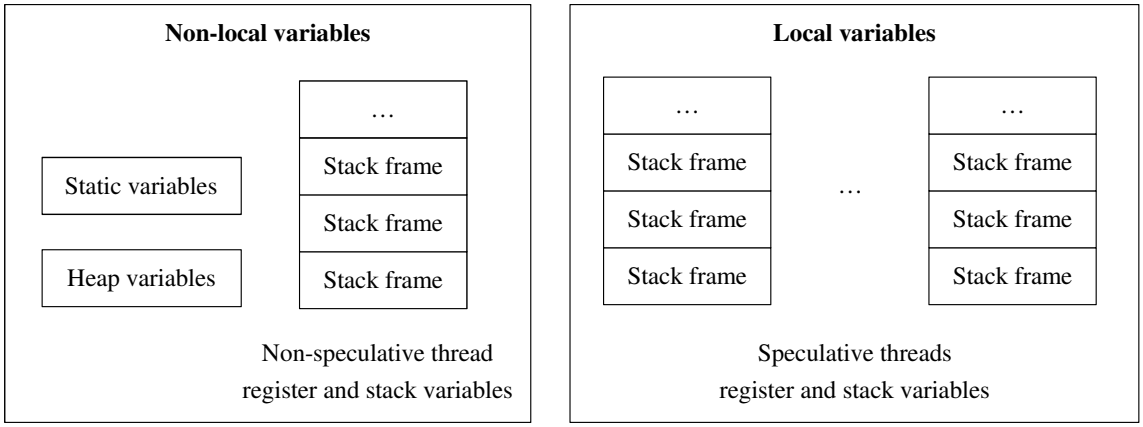


Figure 3–14: Memory Buffering Variables

In the global buffer subsystem, the SharedGlobalBuffer object and the Thread-GlobalBuffer object of each speculative thread are flat since non-speculative and

speculative threads share the same address space (the same addresses for non-local objects). The ThreadLocalBuffer object of each speculative thread is organized as an array of stack frames, with each frame containing a RegisterBuffer and StackBuffer for storing register and stack variables.

3.6.1 Address Space Registration

Memory buffering should guarantee that invalid addresses are not accessed. Also required is to identify whether an address is in the global buffer (non-local variables) or on the speculative stack (speculative stack variables). We solve the problem with an address space registration mechanism. The address space (the start and the end addresses) of each static and heap object is registered in the address space registration module that was discussed for Figure 3–5 during the creation and deletion of the object, that is, at the beginning of program execution for a static object and at memory allocation and deallocation for a heap object. Adjacent spaces can be merged to improve performance. The stack address space of a thread are the addresses between its base and current stack pointers, and is also registered in the address space registration module. A speculative thread is rolled back if it reads/writes an address not in the global and local address spaces.

The current implementation of heap memory registration is to intercept language-specific memory management library calls in LLVM-IR, for example, "malloc" in C and Fortran, and "_Znwm" in C++. We realize that this approach is somewhat ad-hoc and difficult to deal with customized allocators. We plan as future work to solve the problem by hooking into OS system calls or handling page fault signals. As speculative threads may be rolled back, currently we do not allow speculative threads to

allocate/deallocate memory to avoid memory leak, and rollback a speculative thread if it attempts memory allocation/deallocation, which may cause unnecessary performance degradation for some benchmarks. We plan as future work to address the problem by maintaining allocated memory for each thread.

3.6.2 Global Buffer

This section describes the ThreadGlobalBuffer of the global buffer subsystem. The SharedGlobalBuffer will be discussed in the memory buffering optimizations in section 5.2. Each ThreadGlobalBuffer object maintains two maps: a read-set and a write-set, with writes to the global address space redirected into the write-set. Global loads either return the value from the write-set if found there, from the read-set if previously read, or by loading the value from memory and saving it in the read-set (first time).

Conflicts only occur when a speculative thread reads data from an address before the non-speculative thread writes data to the address or other speculative threads commit their write buffer to the main memory. Therefore, the validation process iterates through the read-set of the ThreadGlobalBuffer object, comparing data with the corresponding values in main memory; if they are not equal, then validation fails and the buffer is discarded. Otherwise, validation succeeds and the data in the write-set is committed. After global buffer validation/commit, the speculative thread performs the finalization process to clear the buffer for use by a new speculative thread. As opposed to validation/commit, finalization does not cause critical path delay, as it is performed in parallel with the non-speculative thread execution after thread polling and synchronization that was discussed in section 3.3.

As the addresses of the global read and write operations can be arbitrary, and there may be an arbitrary number of read and write operations, the read- and write-set maps must be efficient. Normal hash maps frequently increase in size as data is inserted, causing dynamic memory allocation and deallocation overhead. Our design is instead to use static memory. The map has a byte array *buffer* of a multiple of the WORD size, a pointer array *addresses* and an integer stack *offsets*, all containing a maximum of N elements, as illustrated in Figure 3–15. The two arrays together implement a hash map while the stack guarantees that validation, commit and finalization operations of threads accessing a small amount of data are fast.

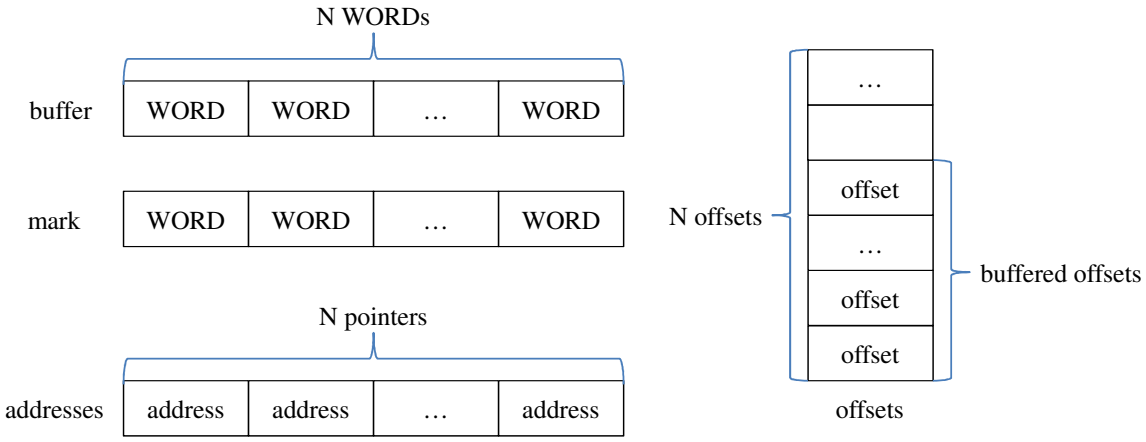


Figure 3–15: Thread Global Buffer Data

The *addresses* are initialized zero at the beginning of program execution. Given an address, the find/insert operation calculates its *offset* in *buffer* as the lower bits of the address, and the array *index* in *addresses* as the offset divided by the WORD size. Then *addresses[index]* is checked; if it is zero, meaning an empty slot, the address is inserted into the *addresses* array, data of WORD size is inserted at *buffer+offset*, and

the offset is pushed onto *offsets*; otherwise, if *addresses[index]* equals the address, meaning the address has been inserted, then the data is accessed in *buffer*, otherwise the hash buffer is conflict. In the buffer conflict case, we store the address and data in a temporary buffer and the speculative thread will wait to be joined at the next check point. If the temporary buffer is used up, the speculative thread rolls back, although this is rare since check points are entered frequently. During validation, commit and finalization, *offsets* is traversed to find addresses and data accessed by the speculative thread.

Different size data accesses can be encountered. Assuming a read/write of *data* of *size* size at address p, the MUTLS memory buffering supports read/write operations if *size* is larger than, equal to, or smaller than WORD, given that one of *size* and WORD is a multiple of the other, and that p is aligned by *size*. If *size* is larger than WORD, we split the address into several WORD pointers and split *data* before write or reconstruct *data* after read operations. To support the case that *size* is smaller than WORD, a byte array *mark* with the same size as *buffer* is needed. First, a normalized address *np* is calculated by making the lowest WORD bits of p 0. If *addresses[index]* equals *np*, then the data is read/written at *buffer+offset*, and *size* bytes from *mark+offset* are set to `0xFF` if it is a write; otherwise, if it is an empty slot, then WORD bytes of data are read from *np* and written to the buffer and *size* bytes from *mark+offset* are set to `0xFF` if it is a write; otherwise, it is a buffer conflict. Validation of the read-set validates all read data, while commit of the write-set only commits data marked by *mark*. An optimization for commit is that if WORD size

data of *mark* is -1, then the WORD bytes of data in *buffer* can be committed all at once.

3.6.3 Local Buffer

The local buffer subsystem is used to transfer local (register and stack) variables between parent and child threads during fork and join though `MUTLS_(set|get)_(regvar|stackvar)_*` library calls. At the preparation transformation step (4) of section 3.5.1, the speculator pass assigns an offset for each register and stack variable. `MUTLS_(set|get)_regvar_*` passes the offset and the register value to the RegisterBuffer object, which in turn stores/loads the register value in a static array. If there are too many variables and the assigned offset exceeds the array size, the speculator pass reports an error and speculation fails. `MUTLS_(set|get)_stackvar_*` is a similar case for stack variables, except that they pass the address and size of the stack variable, and copy the stack data.

The above is complicated by the potential presence of pointers to stack variables. If such a pointer is used in a speculative context, and the thread commits, the pointer will be invalid since the speculative version of the stack variable no longer exists. Instead, it should point to the non-speculative version. We propose a pointer mapping mechanism to solve the problem, as illustrated in Figure 3–16.

During commit of pointers through `MUTLS_get_(regvar|stackvar)_ptr` calls, the value of the pointer is checked and if it is in the stack address space of the speculative thread, it is mapped to point to the corresponding variable in the non-speculative thread. Since the non-speculative and speculative functions may have different stack

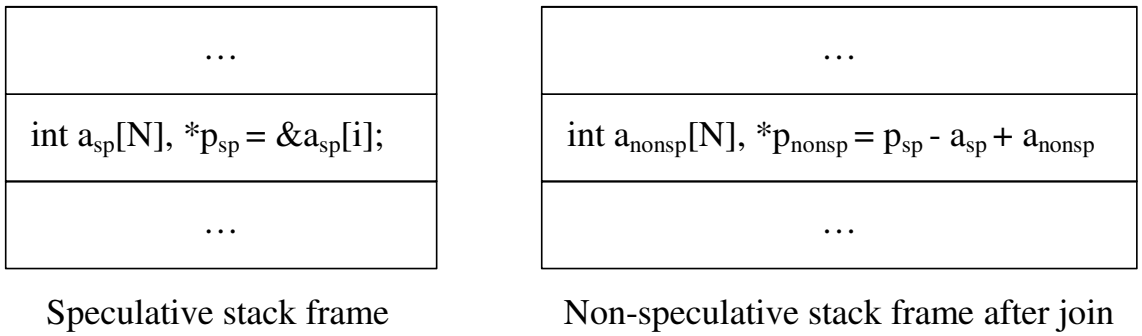


Figure 3–16: Pointer Mapping Mechanism

layouts, we cannot use a constant offset for mapping of all variables. The implementation thus records stack variable addresses of the non-speculative and the committing speculative threads in a hash map during `MUTLS_set_stackvar_*` calls. Then during `MUTLS_get_(regvar|stackvar)_ptr` calls, for each local pointer, the non-speculative thread searches the stack variable in the hash map and calculates the non-speculative pointer value based on the address of the non-speculative stack variable and the offset of the pointer value within the stack variable.

A complication to this occurs when type-casts between pointers and integers are present in a function—the pointer mapping mechanism may be unsafe as integer values may be used in various instructions including I/O. Our current implementation is to disallow pointer-integer type-casts unless the value is inside the global address space that is not mapped. Before type-cast instructions between pointers and integers, `MUTLS_ptr_int_cast` library calls are inserted which barrier the speculative thread if the pointer/integer value is not in the global address space.

Stack variable loads/stores in nested frames of a speculative function directly access the function's stack, since they do not affect the non-speculative thread and

the stack acts as buffer itself. These variables are committed (copied) to the stack of the non-speculative thread by `MUTLS_get_stackvar_*` calls. Stack variables at the bottom frame are accessed as non-local data, the non-speculative version of which are buffered in global buffer subsystem during `MUTLS_(load|store)_*` calls.

3.6.4 Register Variable Validation

In the `MUTLS_speculate_regvar_{int32, ptr,` etc} library calls generated in speculation blocks as was discussed in section 3.5.2, local register variables live at the beginning of the join point block are initialized as they would be when normal execution reaches the join point. Induction variables and expressions can be made live by code transformations such as hoisting/sinking (moving to blocks before the fork point/after the join point). If the variable is still not live at the fork point, then it should be predicted by the forking thread, otherwise the speculative child thread retrieves an uninitialized value. When the forking thread reaches the join point, it should validate that the live local variables were correctly speculated, which is implemented by the `MUTLS_validate_regvar_{int32, ptr,` etc} library calls. The speculative thread will rollback if this validation fails.

The `MUTLS_speculate_regvar_*` and `MUTLS_validate_regvar_*` library calls implement the register variable value prediction. We use the offset assigned in the preparation step (4) of section 3.5.1 as well as the fork point id to identify each register variable. The `MUTLS_speculate_regvar_*` function calls set the register variable values to be retrieved by the `MUTLS_get_regvar_*` calls of the speculative thread. The `MUTLS_speculate_regvar_*` functions also have a `predictor_type` integer argument to

specify the value predictor to be used for the register variable. Currently, we implement the last value (LV) [98], stride (S) [72] and 2-delta stride (2DS) [151] value predictors. Support for more advanced predictors is planned for future work. For live register variables, the speculator pass sets the `predictor_type` argument to NULL so that they are not predicted. The `MUTLS_validate_regvar_*` library calls save the actual values of the register variables to the runtime library, which can then be used by future `MUTLS_speculate_regvar_*` calls for value prediction based on the history register variable values.

3.7 Stack Frame Optimization

In the baseline stack frame reconstruction design shown in Figure 3–12, each time a speculative thread enters/returns from a nested function call, it saves the local variables into the MUTLS runtime library, which incurs unnecessary overhead for returned nested function call frames since the saved local variables are not used. This is an especially important problem for recursive applications as nested functions are entered and returned frequently. We in turn propose the stack frame optimization to address the issue.

The stack frame optimization is illustrated in Figure 3–17. The approach is to save/restore local variables only at thread joining time. When the speculative thread commits at a check point, to construct the nested call frames in the runtime library, instead of exiting as is the case in 3–12, it calls the `MUTLS_commit_frame` library function, passing the synchronization point counter, the current function counter, and the speculative thread rank, and then returns from the current function call. Then `MUTLS_commit_frame` function pushes a new stack frame in the runtime library

```
int work(args){
  int stackptr, c, r;
  S1;
  for(i = 0; i < n; i++){
    if(MUTLS_get_CPU(&stackptr, 0, mixed, must, 0)){
      MUTLS_speculate_local
      work.proxy(args, 1, rank);
    }
    S2;
    a[i] = abs(b[i]);
    if(MUTLS_get_speculated_child(&stackptr, 0, 0)){
      MUTLS_validate_local
      if(MUTLS_synchronize(&stackptr, 0, &c, &r)){
        goto synchronization_table;
      }
    }
  }
  S3;
  if(MUTLS_get_CPU(&stackptr, 1, mixed, must, 0)){
    MUTLS_speculate_local
    work.proxy(args, 2, rank);
  }
  S4;
  s = sum(n, a);
  if(MUTLS_get_speculated_child(&stackptr, 1, 0)){
    MUTLS_validate_local
    if(MUTLS_synchronize(&stackptr, 1, &c, &r)){
      goto synchronization_table;
    }
  }
  S5;
  printf("%d\n", s);
  S6;
synchronization_table:
  switch(c){
    case 1: goto synchronization block 1 restore block
    ...
    case 7: goto synchronization block 7 restore block
    case 8: goto synchronization block 8 restore block
  }
// restore blocks for synchronization blocks
}
```

(a) Non-speculative version

```
void work.stub(rank){
  args = MUTLS_restore_local(rank)
  counter = MUTLS_restore_local(rank)
  v = work.speculative(args, counter, rank);
  MUTLS_save_local(v, rank)
  MUTLS.commit_stub(8, rank);
}
```

(b) Proxy and stub functions

Statement markers

MUTLS library calls

MUTLS local variable library calls

MUTLS speculator pass generated functions

MUTLS speculator pass added blocks

```
int work.speculative(args, counter, rank){
  int stackptr;
  switch(counter){
    case 0: break;
    case 1: goto join point 1 restore block
    case 2: goto join point 2 restore block
  }
  S1;
  for(i = 0; i < n; i++){
    if(MUTLS_check_point(3, rank)){
      MUTLS_save_local
      MUTLS_commit_frame(3, counter, rank);
      return;
    }
    if(MUTLS_get_CPU(&stackptr, 0, mixed, must, rank)){
      MUTLS_speculate_local
      work.proxy(args, 1, rank);
    }
    S2;
    tmp = MUTLS_load_int32(&b[i], rank);
    MUTLS_store_int32(&a[i], abs(tmp), rank);
    if(MUTLS_get_speculated_child(&stackptr, 0, rank)){
      MUTLS_validate_local
      MUTLS_save_local
      MUTLS_sync_parent(1, rank);
    }
  }
  if(counter > 0 && MUTLS_get_speculated_child(&stackptr, 0, rank)){
    MUTLS_save_local
    MUTLS_barrier_point(4, rank);
  }
  S3;
  if(MUTLS_get_CPU(&stackptr, 1, mixed, must, rank)){
    MUTLS_speculate_local
    work.proxy(args, 2, rank);
  }
  S4;
  if(MUTLS_check_point(5, rank)){
    MUTLS_save_local
    MUTLS_commit_frame(5, counter, rank);
    return;
  }
  s = sum(n, a);
  if(MUTLS_is_committing(rank)){
    MUTLS_save_local
    MUTLS_commit_frame(5, counter, rank);
    return;
  }
  if(MUTLS_get_speculated_child(&stackptr, 1, rank)){
    MUTLS_validate_local
    MUTLS_save_local
    MUTLS_sync_parent(2, rank);
  }
  S5;
  MUTLS_save_local
  MUTLS_terminate_commit_frame(6, rank);
  return;
  printf("%d\n", s);
  S6;
// restore blocks for join point blocks
}
```

(c) Speculative version

Figure 3–17: Stack Frame Optimization performed by the Speculator Pass

and returns if the function counter is 0 (it is in a nested function call) and notifies the non-speculative thread to join the stack frame if the counter is non-0 (it is in the speculative entry stack frame). To be able to continue returning and constructing the stack frames in the runtime library until the speculative thread reaches the entry stack frame, after each nested function call the runtime library function `MUTLS_is_committing` is called to check whether the speculative thread is committing, and if so the speculative thread saves local variables, calls `MUTLS_commit_frame` to commit the current frame and returns from the current function call. The speculative thread may also return from the bottom frame to reach the stub function frame, in which case the stub function saves the return value of the speculative function call in the runtime library and calls `MUTLS_commit_stub` with a new synchronization counter. The synchronization table of the non-speculative thread also adds an entry to get the return value of the speculative thread function and return from the current function.

Another optimization we apply is to remove the need of the *ranks* array for speculation/synchronization. This is beneficial because each time a speculative thread enters a nested function call with fork/join points, the *ranks* array is initialized 0, even if speculation is disabled for the stack frame. Without the *ranks* array, we still need to determine whether a child thread has been speculated at the join point when a thread reaches a join point. We propose a new thread naming mechanism to solve the problem. The idea is to use both the speculated stack frame and the fork/join point id to identify each speculative thread instead of just the fork/join point id, so

that each actively running speculative thread can be uniquely identified by its parent thread. Then we call the library functions `MUTLS_get_CPU(stackptr, id, model, hint, arg)` at fork points and `MUTLS_get_speculated_child(stackptr, id, rank)` at join points and perform thread forking/joining if the functions return true.

3.8 Thread Task Optimization

In the baseline MUTLS virtual CPU design that was described in section 3.5.2, threads are bound to virtual CPUs, which are usually set to no more than the number of physical CPUs of the running machine to avoid performance degradation caused by threads representing sequentially later execution competing with the CPU time of sequential earlier threads. This design however, may unnecessarily limit thread work coverage of the software-TLS system, as demonstrated by the example in Figure 3–18.

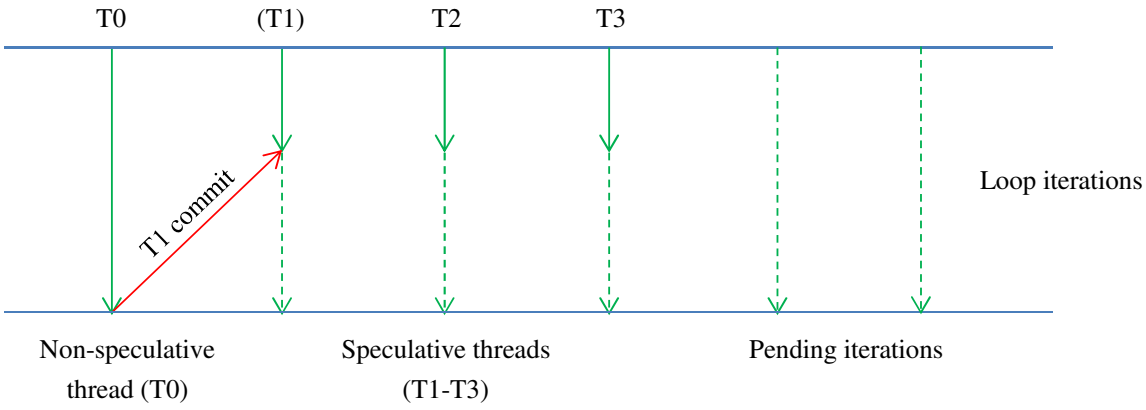


Figure 3–18: Thread Coverage Problem of Baseline Virtual CPU Framework Design. After T1 commits, only 3 threads are running.

A loop with N iterations is speculatively parallelized with P virtual CPUs ($N > P$). In the figure, P is 4 and the non-speculative thread T0 in-order speculated threads T1, T2 and T3. When T0 completes its iteration, it joins T1 and T1 commits, after which there are only 3 threads running. As the speculative threads need buffering and thus are slower than the non-speculative thread, more time is required before T3 reaches the end of the iteration to speculate a new thread, resulting in less than optimal thread coverage.

We propose the thread task optimization to solve the problem. Instead of binding each thread to a virtual CPU that always corresponds to an operating system (OS) thread, the optimization associates each thread with a thread task, which can be in either running or pending state, as illustrated in Figure 3–19.

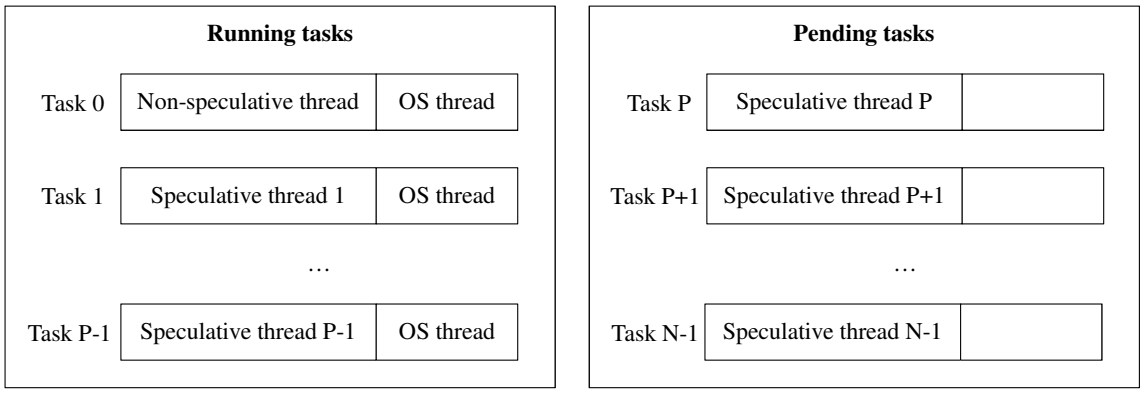
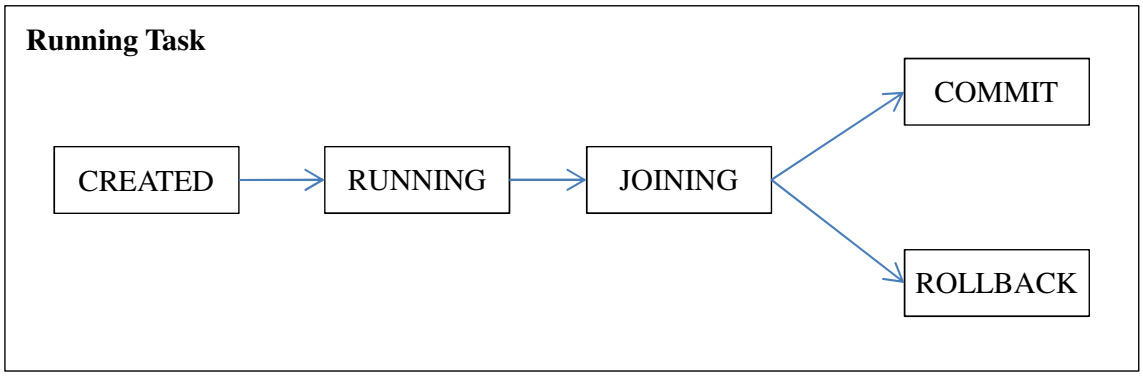


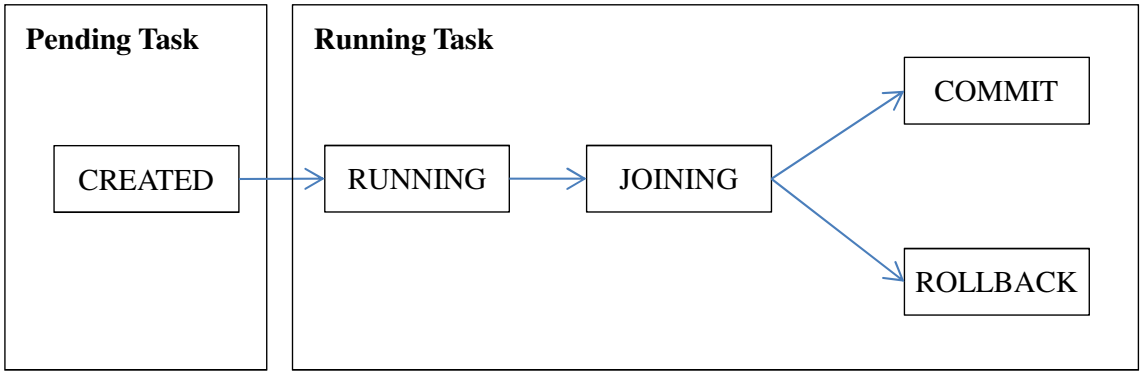
Figure 3–19: Running and Pending Thread Tasks

A running task has a corresponding virtual CPU (OS thread) while a pending task does not. Therefore, though running tasks should generally be no more than physical CPUs, the total number of running and pending tasks can be larger than the number of physical CPUs. After a running task exits (its speculative thread

commits or rolls back), if there is a pending task, the pending task is assigned its OS thread and becomes a running task, otherwise its OS thread is returned to the underlying threading implementation. Though the number of tasks N is larger than the number of virtual CPUs P, we note that N can be a bounded value with respect to P, in particular, $N < 2P$. This is because each speculative thread needs to create at most one pending task. Moreover, if using in-order forking model, then only the most-speculative thread needs to create a pending task, and thus $N = P + 1$.



(a) OS Thread Available at Fork Point



(b) OS Thread Unavailable, Pending Task Available at Fork Point

Figure 3–20: State Transition with Thread Task Optimization

The state transition of the MUTLS framework with the thread task optimization is illustrated in Figure 3–20. When a thread reaches a fork point, there are three possibilities: a virtual CPU (OS thread) is available, no virtual CPU but a pending task is available, and no pending task is available. If a virtual CPU is available, a thread task data structure is registered and set to running state, and a child thread is speculated on the thread task executing on the virtual CPU. If no virtual CPU is available at the fork point, yet a thread task data structure is available, the thread task data structure is registered and set to pending state, and a child thread is created on the thread task. When a virtual CPU is available again, the pending thread task is set to running state and scheduled on the virtual CPU for execution. If no pending task is available, the fork point is ignored and no child thread is speculated.

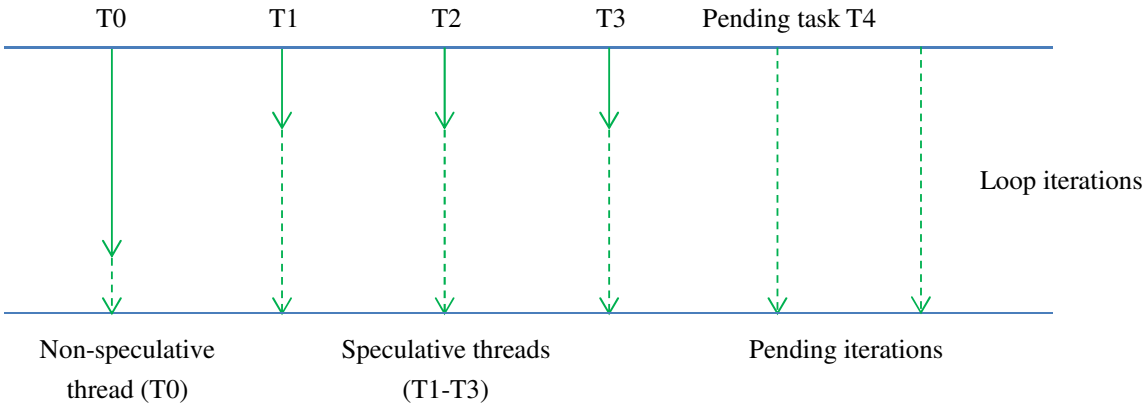


Figure 3–21: Thread Task Optimization - Normal Execution

The thread task optimization for the example of Figure 3–18 is demonstrated in Figures 3–21 and 3–22. After threads T1, T2 and T3 are in-order speculated, there are no available OS threads, and therefore T3 speculates a pending thread task T4. After T1 commits, T4 is assigned the OS thread of T1 and scheduled to run on a

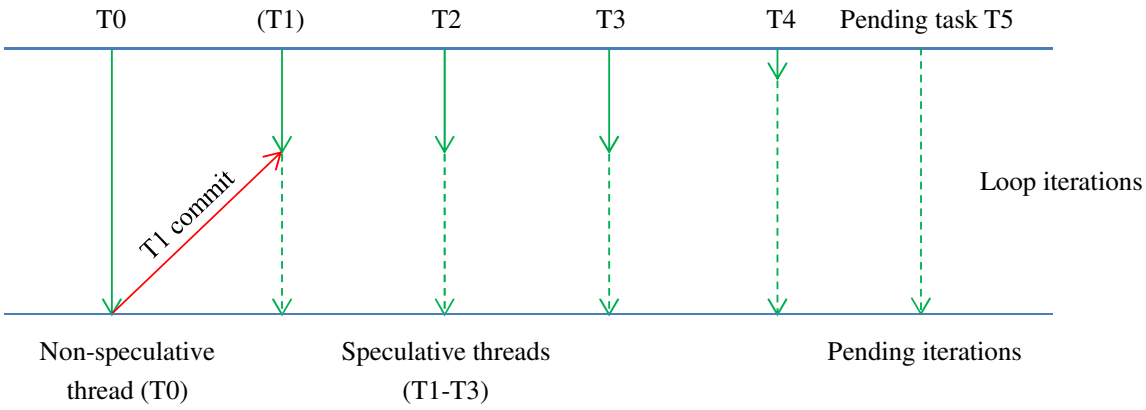


Figure 3–22: Thread Task Optimization - Thread Joining

physical CPU. T4 then reaches a fork point and speculates a new pending thread task T5. In implementation, T5 and T1 can share the same rank and thread status and task data.

3.9 Chapter Summary

In this chapter, we presented the Mixed-model Universal software Thread-Level Speculation (MUTLS) framework that is purely based on the LLVM intermediate representation, and thus is capable to support all LLVM front-end source languages and back-end target architectures. With a mixed forking model, MUTLS can exploit more parallelism from tree-form recursion applications. We first described how to implement fork point annotation in the GCC/DragonEgg LLVM front-end, and then presented detailed design and implementation of the back-end, including the speculator LLVM transformation pass and the MUTLS runtime library. For the back-end, we first presented an overview of the MUTLS runtime library architecture, state transition and forking models, then described the LLVM-IR transformation for each step of preparation, fork, join and stack frame reconstruction, next put forward

various memory buffering issues of address space registration, global/local buffer, local pointer mapping and register variable validation, and finally proposed the stack frame optimization to reduce stack frame maintenance overhead and the thread task optimization to increase thread work coverage.

CHAPTER 4
Experimentation of MUTLS

In this chapter, we experiment with the MUTLS framework design. The experiment environment and the benchmarks in this and following chapters are presented in section 4.1. In section 4.2 we compare the speedups of baseline MUTLS framework design with the stack frame and thread task optimizations, and cluster the applications into high speedup, mediocre speedup and slowdown categories according to their performance. In order to see how well the MUTLS software-TLS framework performs with respect to the theoretically highest speedups, we show the performance ratio of the speculatively parallelized benchmarks against the OpenMP manually parallelized versions in section 4.3. As we will see that the performance is not always ideal, for further improvement we analyse the performance characteristics of the benchmarks in section 4.4, including critical path efficiency, speculative path efficiency, power efficiency and the runtime breakdown. Afterwards, we compare the strengths and weaknesses of different forking models in section 4.5 and finally experiment with the rollback sensitivity performance characteristics of the benchmarks in section 4.6.

4.1 Experiment Environment and Benchmarks

We implement the MUTLS[1] system on LLVM 3.5 with the GCC-4.6.4 and DragonEgg front-end. For experimentation we use an AMD Opteron 6274 machine with 64 2.2GHz processor cores (4×16-core, 8×2MB L2 cache) and 64GB memory. The operating system is 64-bit Red Hat Enterprise Linux.

Table 4–1: Benchmarks

Benchmark	Description	Source
bh	Barnes-Hut N-body simulation	Lonestar [136]
raytracing	ray tracing rendering	c-ray [2]
smallpt	path tracing global illumination rendering	smallpt [27]
sparsematmul	sparse matrix times vector	SciMark [26]
bwaves	blast waves simulation	SPEC CPU2006 [81]
lbm	incompressible fluids simulation	SPEC CPU2006
3x+1	3x+1 problem in number theory	MUTLS [42]
mandelbrot	mandelbrot fractal generation	mandelbrot [11]
md	3D molecular dynamics simulation	md [13]
fft	recursive Fast Fourier Transform	MUTLS
matmult	block-based matrix multiplication	MUTLS
nqueen	N-queen problem	MUTLS
tsp	travelling sales person (TSP) problem	MUTLS
lavaMD	3D hierarchical particle simulation	Rodinia [48, 49]
streamcluster	online stream data clustering	Rodinia
kmeans	k-means clustering	Rodinia
srad	speckle reducing anisotropic diffusion imaging	Rodinia
cfd	3D fluid computational dynamics	Rodinia
heartwall	heart wall image processing	Rodinia
myocyte	cardiac myocyte modeling	Rodinia

[1] MUTLS is available online at `http://www.sable.mcgill.ca/~zcao7/mutls`

Table 4–2: Benchmark Characteristics

Benchmark	Problem Size	Language
bh	12800 bodies	C++
raytracing	192 objects, 800×600 resolution	C
smallpt	4 ray samples, 800×600 resolution	C++
sparsematmul	2M×2M matrix, 100M non-zero elements	C
bwaves	train run	Fortran
lbm	train run	C
3x+1	40M integers (enumerate)	C/Fortran
mandelbrot	512×512 image, maximum 80000 iterations	C/Fortran
md	512 particles, 400 iteration steps	C/Fortran
fft	2^{20} doubles	C
matmult	2048×2048 matrices	C
nqueen	14 queens	C
tsp	12 cities	C
lavaMD	10 boxes in each dimension	C
streamcluster	65536 points, 256 dimensions, 1000 clusters	C++
kmeans	494020 points	C
srad	609×590 image, 0.5 saturation	C
cfd	97046 elements	C++
heartwall	104 609×590 images	C
myocyte	1024 instances	C

The benchmarks we experiment with are summarized in Tables 4–1 and 4–2. These benchmarks are typical workloads used in a range of application domains, and were selected because they exhibit a variety of workload characteristics while exposing significant opportunities for parallelism. We select the Rodinia [48, 49] benchmark suite as it includes important scientific computing applications and provides OpenMP manual parallelization implementations which we can compare with. We use 7 but not all of its benchmarks because others are not suitable for TLS, e.g. I/O bound. The SPEC CPU2006 [81] benchmark suite comprises representative application workloads; however, only 2 of its benchmarks are selected because

other benchmarks require compiler analysis/transformation currently not available in LLVM to expose the parallelism opportunities for TLS. We select the bh and sparsematmul benchmarks from Lonestar [136] and SciMark [26] because they are widely-used applications most suitable for TLS automatic parallelization: TLS can effectively parallelize these benchmarks as memory dependencies do not exist in the speculatively parallelized program, but static automatic parallelization approaches cannot as the absence of memory dependencies cannot be proved by static compiler analysis.

We also include stand-alone benchmarks for better evaluation of the MUTLS software-TLS system. 3x+1 is a well-known number theory problem that avoids memory access during the computation, and thus serves as an idealized benchmark for our software-TLS system. The mandelbrot and md benchmarks are computation intensive scientific applications and usually serve as synthetic benchmarks. We select raytracing and smallpt as they are important computer graphics applications with workload characteristics interesting for MUTLS: they have recursive function calls and are neither too computation intensive nor too memory intensive. Though most of the benchmarks are parallelized on loop iterations, we include 4 tree-form recursion applications to evaluate the mixed forking model, including two divide-and-conquer benchmarks fft and matmult, and two depth-first-search (DFS) benchmarks nqueen and tsp. Matmult is a block-based matrix multiplication like Strassen's algorithm.

These benchmarks give us a mix of CPU- and memory-intensive computation. We consider 3x+1, mandelbrot, md and myocyte to be computation intensive, bh, raytracing, smallpt, bwaves, fft, matmult, nqueen, tsp, lavaMD, kmeans, srad and

heartwall as locality memory intensive, and sparsematmul, lbm, streamcluster and cfd non-locality memory intensive. Note that the computation/memory intensiveness is not characterized by the total memory used, but by the memory access frequency (density), defined as the number of read/writes divided by the program runtime $\rho = N_{rw}/T$. Thus, although matmult has time complexity $O(N^3)$ and space complexity $O(N^2)$, it is still considered memory intensive. We use train run for bwaves and lbm because it takes long time to run all versions of the ref run data sets.

4.2 Speedup

Given the execution time of a parallelized program on N CPU cores T_N, and of the original sequential program T_s, the absolute speedup is defined as T_s/T_N. The speedups of the benchmarks are presented in Figures 4–1 to 4–6. The geometric mean of the speedups of all benchmarks are shown in Figure 4–6(d). The "baseline" results are speedups of the baseline MUTLS framework design that was described from section 3.1 to section 3.6. The "stack frame", "thread task" and "stack frame & thread task" are the baseline design with the stack frame optimization of section 3.7, the thread stack optimization of section 3.8 and both optimizations, respectively.

As expected, the stack frame optimization significantly improves the performance of recursive benchmarks with frequent function calls, such as bh, raytracing, smallpt, nqueen and tsp, resulting from reduction in the stack frame maintenance overhead. The reason that fft and matmult are not improved much is that they are memory intensive benchmarks and the dominant overhead is memory buffering. On the other hand, applications with many loop iterations such as raytracing, smallpt and mandelbrot benefit significantly from the thread task optimization, as a result

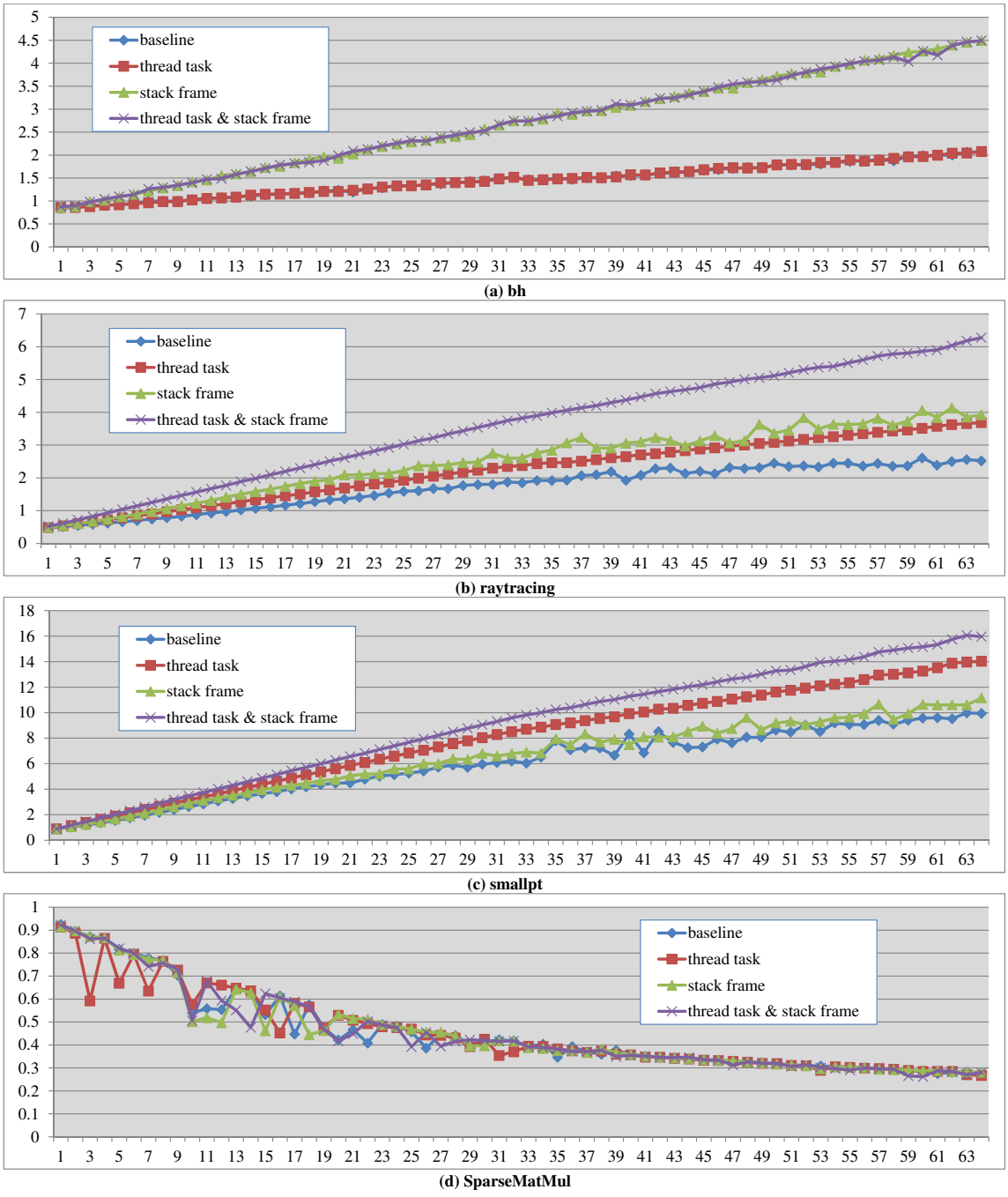
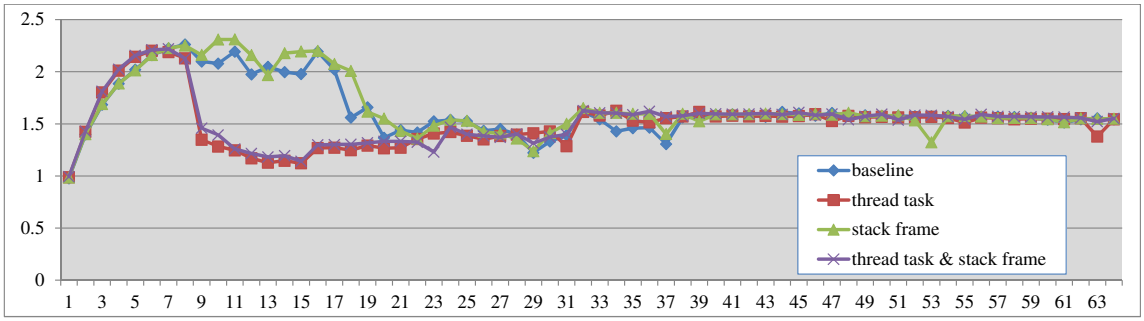
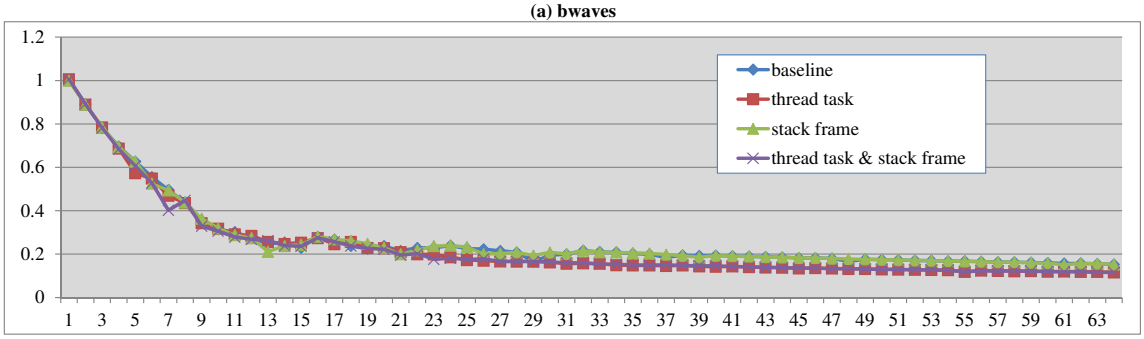


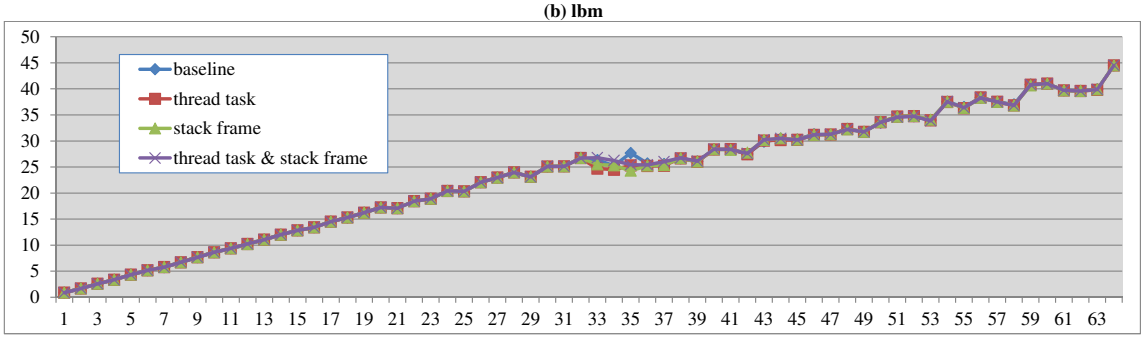
Figure 4–1: Speedup versus number of CPUs; higher is better (1/6)



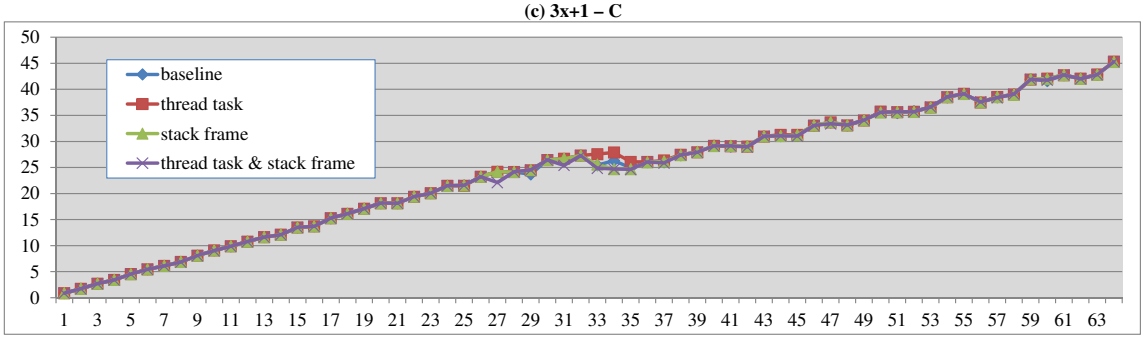
(a) bwaves



(b) lbm

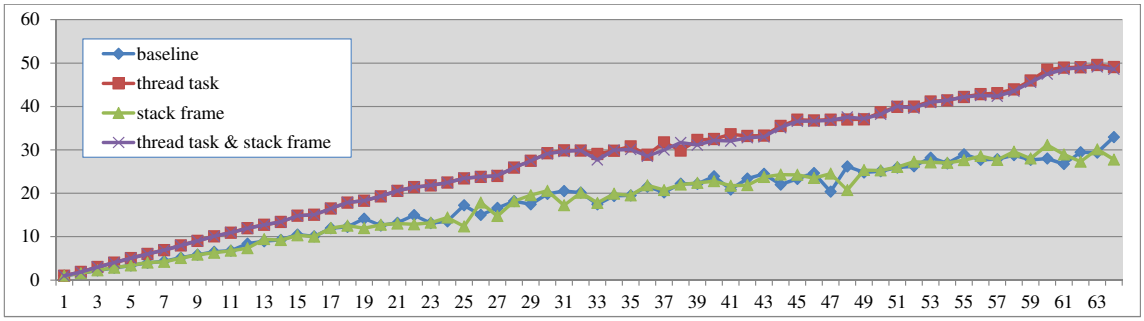


(c) 3x+1 – C

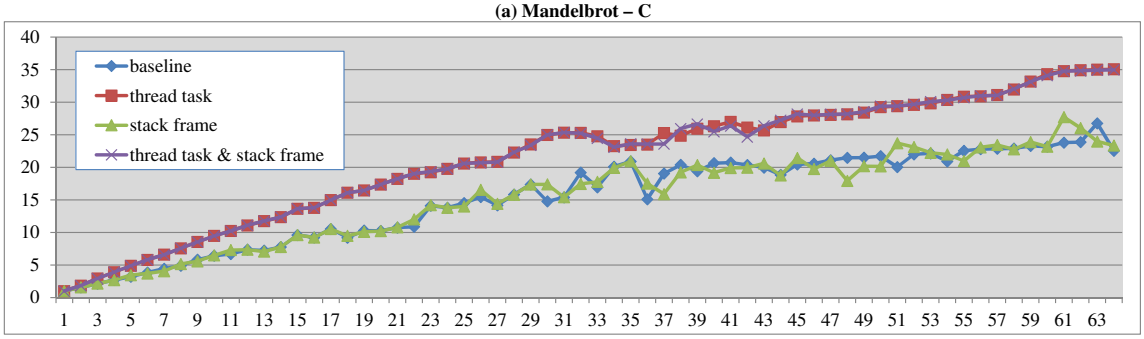


(d) 3x+1 – Fortran

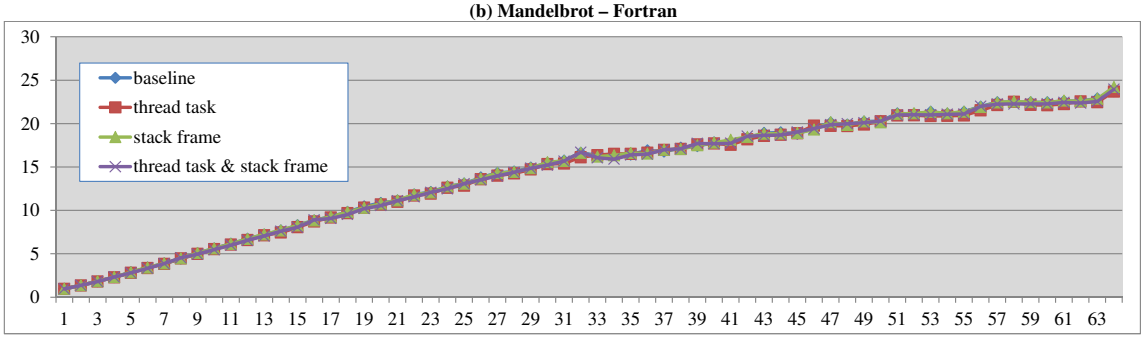
Figure 4–2: Speedup versus number of CPUs; higher is better (2/6)



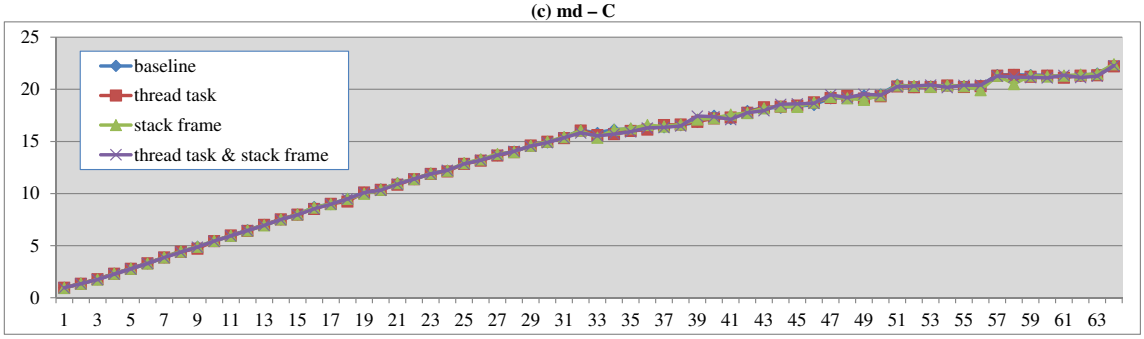
(a) Mandelbrot – C



(b) Mandelbrot – Fortran



(c) md – C



(d) md – Fortran

Figure 4–3: Speedup versus number of CPUs; higher is better (3/6)

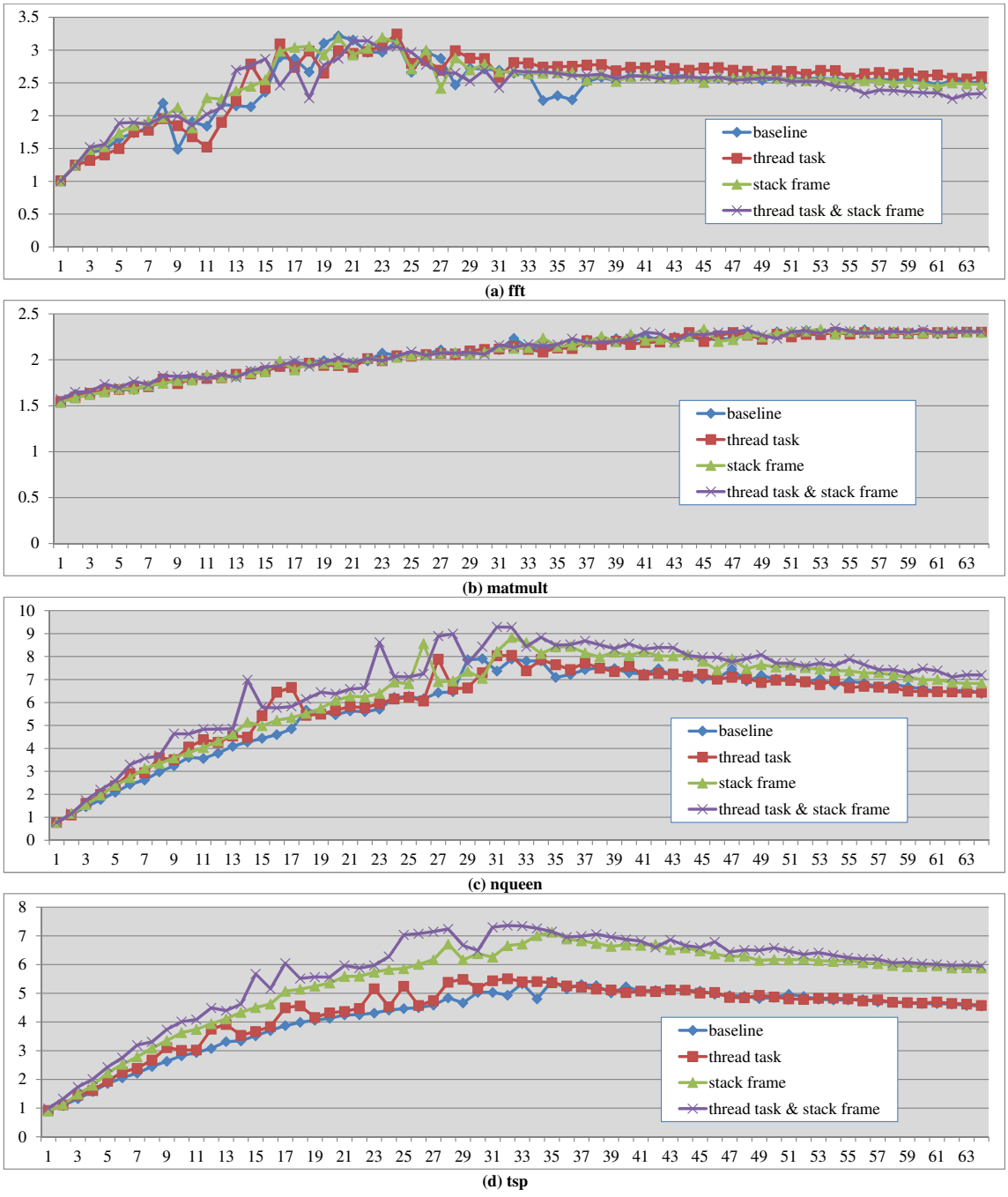


Figure 4–4: Speedup versus number of CPUs; higher is better (4/6)

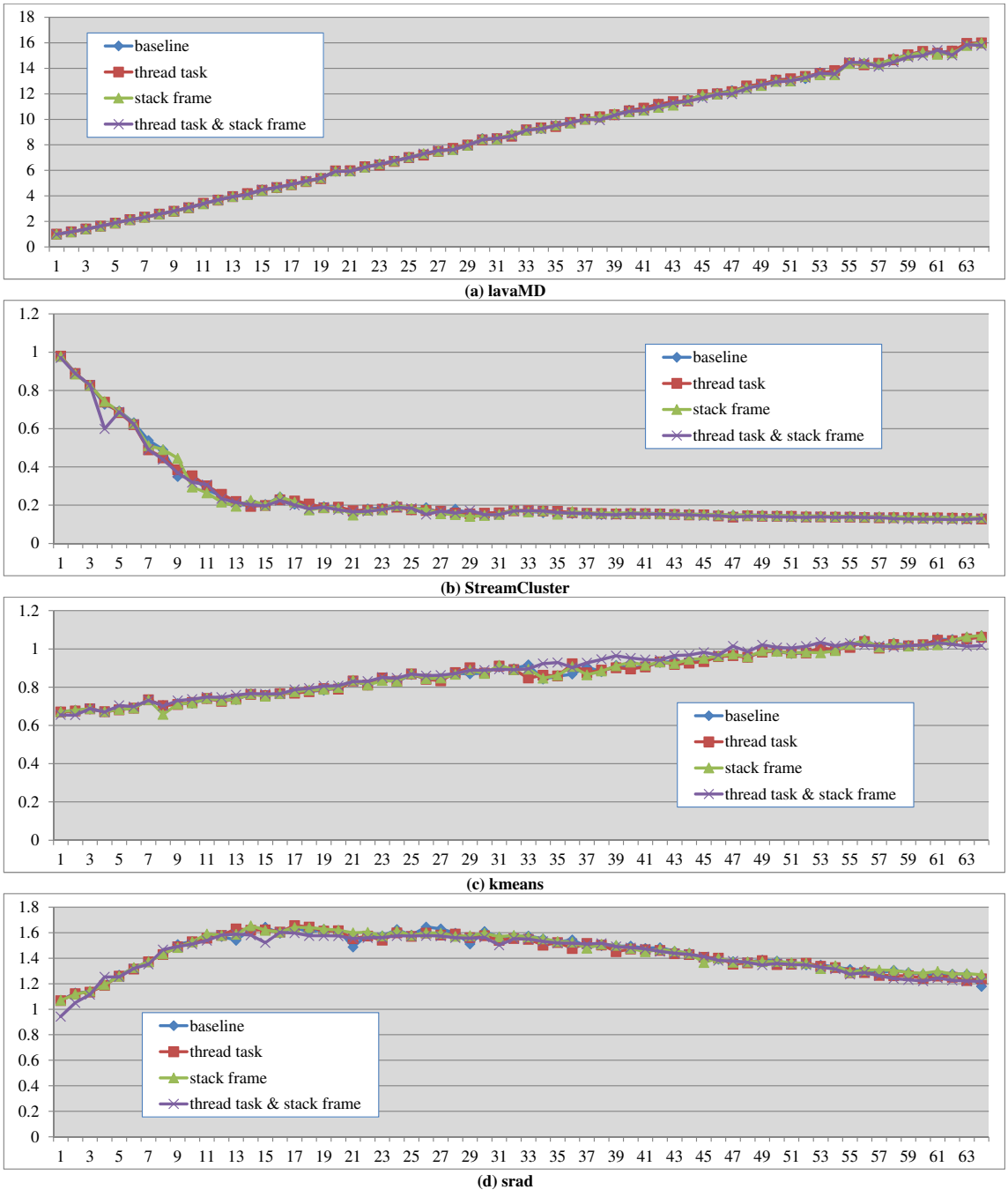


Figure 4–5: Speedup versus number of CPUs; higher is better (5/6)

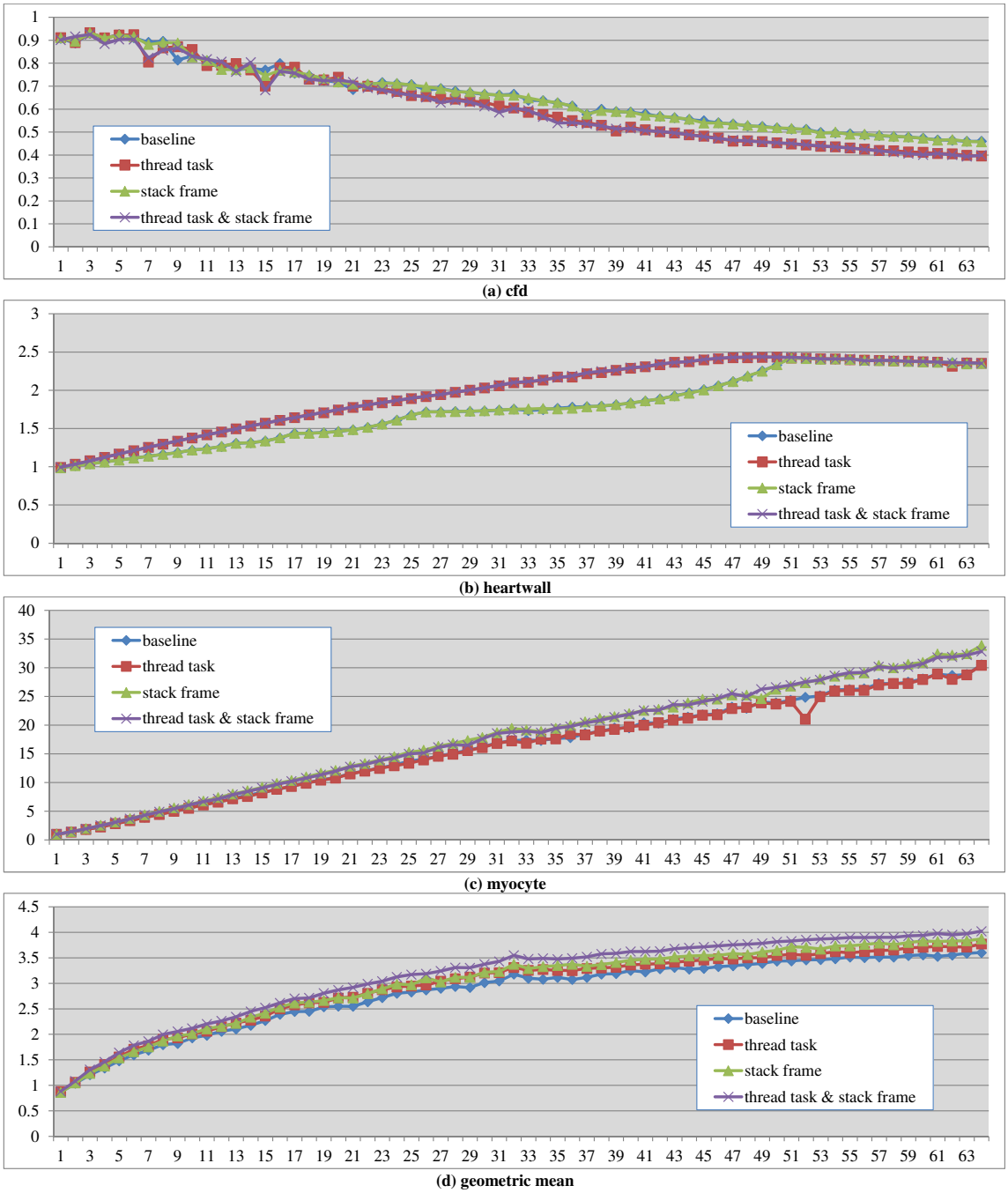


Figure 4–6: Speedup versus number of CPUs; higher is better (6/6)

of more parallel thread work coverage. Other benchmarks such as bh, 3x+1, md and lavaMD are not improved by the optimization because we employ the "blockize" transformation that statically distributes the loop iterations to the total number of CPU cores and thus more thread tasks than the number of CPU cores are not beneficial. For bwaves, the thread task optimization is beneficial from 2 to 6 cores, but detrimental from 7 to 27 cores. This is because more speculative threads result in more speculative thread work coverage that reduces critical path work time, but this also results in more validation/commit time that increases the critical path thread joining time. With more cores, the thread joining time of the benchmark becomes dominant as will be discussed in section 4.4, and thus the runtime is slower. For the geometric mean of all the benchmarks, the "stack frame", "thread task" and "stack frame & thread task" versions improve speedups of the baseline version from 3.60 to 3.77, 3.87 and 4.02 respectively, at 64 cores.

The following experiments are thus performed on the highest speedup "stack frame & thread task" framework design. The performance of the "stack frame & thread task" versions of the benchmarks with high speedups, mediocre speedups and slowdowns are summarized in Figures 4–7, 4–8 and 4–9, respectively.

As can be seen from Figure 4–7, the high speedup applications generally have computation-intensive workloads and exhibit linear speedups. The lower scalability of the Fortran versions of the mandelbrot and md benchmarks is mainly because of their additional memory buffering overhead, e.g., the shapes of arrays being allocated on the stack. On the other hand, the mediocre speedup benchmarks in Figure 4–8 can be sorted into three categories: (1) nqueen and tsp that peek speedups at around

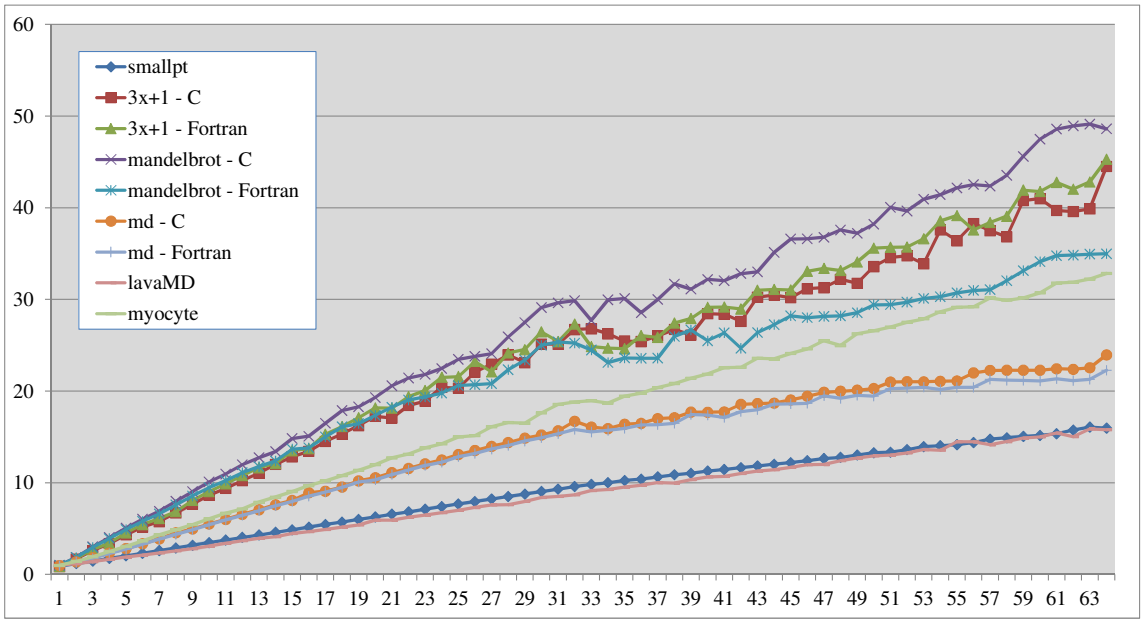


Figure 4–7: Performance Comparison - High Speedup

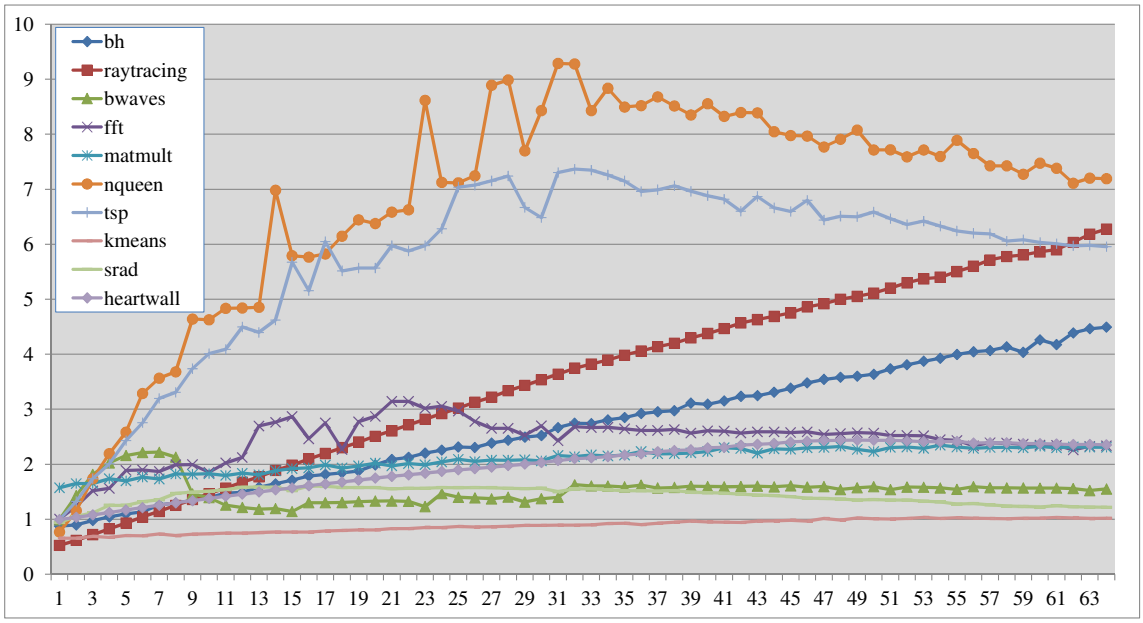


Figure 4–8: Performance Comparison - Mediocre Speedup

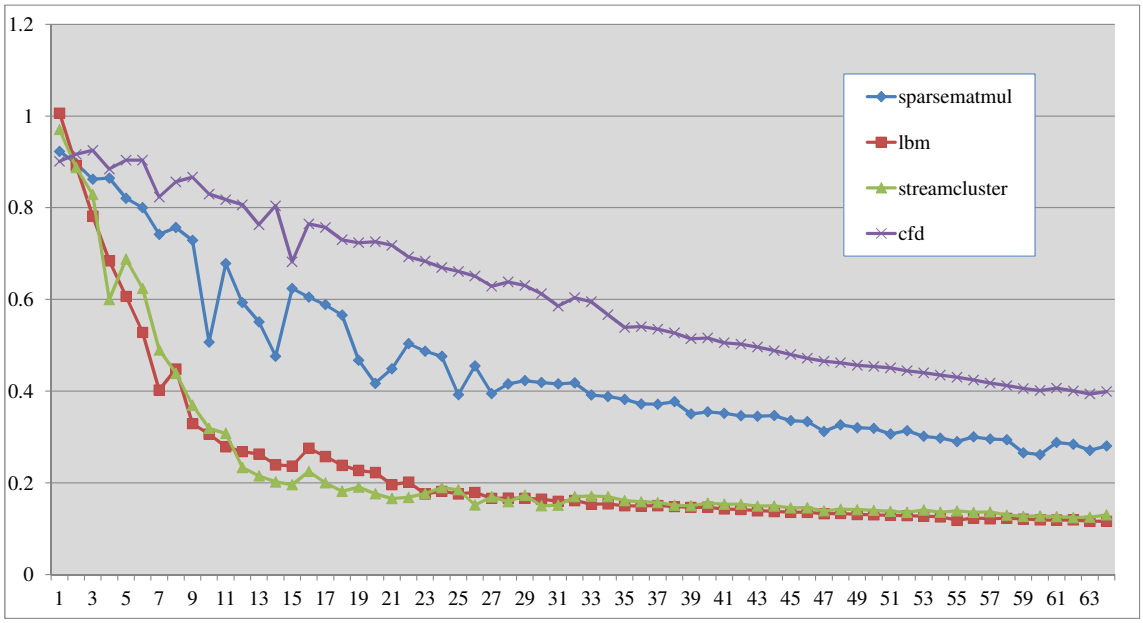


Figure 4–9: Performance Comparison - Slowdown

32 CPU cores due to insufficient parallelism in deeper recursive calls; (2) raytracing and bh that have linear but slowly increasing speedups due to memory buffering time slowing the speculative threads; (3) fft, matmult, bwaves, kmeans, srad and heartwall that are relatively memory intensive whose speedups are saturated at few CPU cores. For fft and matmult, the small threads speculated in deeper recursive calls also cause significant amount of idle time, as will be discussed in section 4.4 below. Larger problem sizes may relieve the problem, although a larger amount of data also requires larger memory buffers for speculation not to overflow, which would result in longer startup times for the programs. We can see from Figure 4–9 that the benchmarks with slowdowns are generally highly memory intensive, with performance from 0.12 (lbm) to 0.40 (cfd) at 64 CPU cores; this is due to the domination of validation/commit time.

4.3 Theoretically Ideal Performance

To understand how well the MUTLS framework performs in a relative sense, we compare the runtime of the MUTLS speculatively parallelized versions with the OpenMP manually parallelized versions, which serve as the theoretically ideal performance of the MUTLS software-TLS system. On N CPU cores, given the execution time of the speculatively parallelized program $T_{N-mutls}$, and the runtime of the OpenMP parallelized version T_{N-omp}, the MUTLS/OpenMP runtime ratio is defined as $T_{N-mutls}/T_{N-omp}$. The MUTLS/OpenMP runtime ratios of the benchmarks are presented in Figures 4–10 to 4–13.

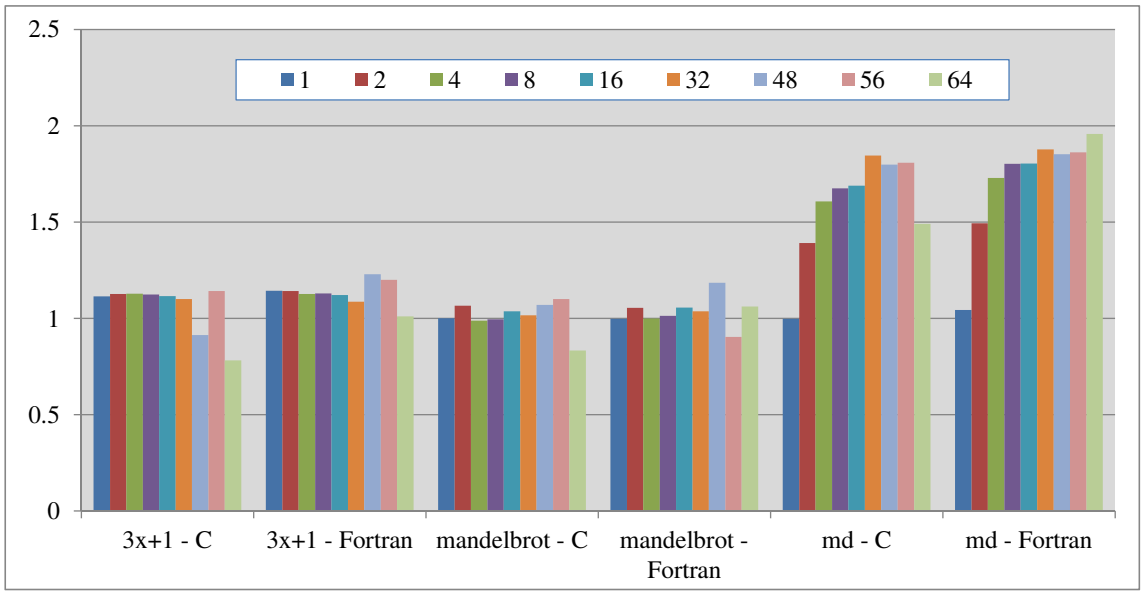


Figure 4–10: MUTLS/OpenMP Runtime ratio; higher is worse (1/4)

We can see that computation intensive benchmarks such as 3x+1, mandelbrot, md and myocyte generally achieve ideal speedups, which is expected. On the other

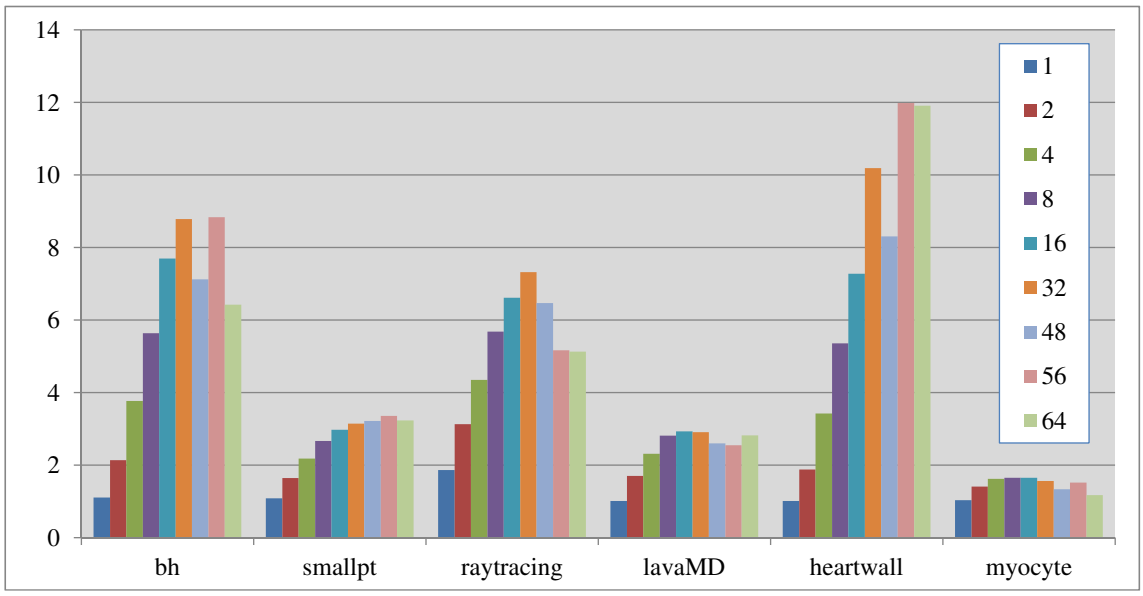


Figure 4–11: MUTLS/OpenMP Runtime ratio; higher is worse (2/4)

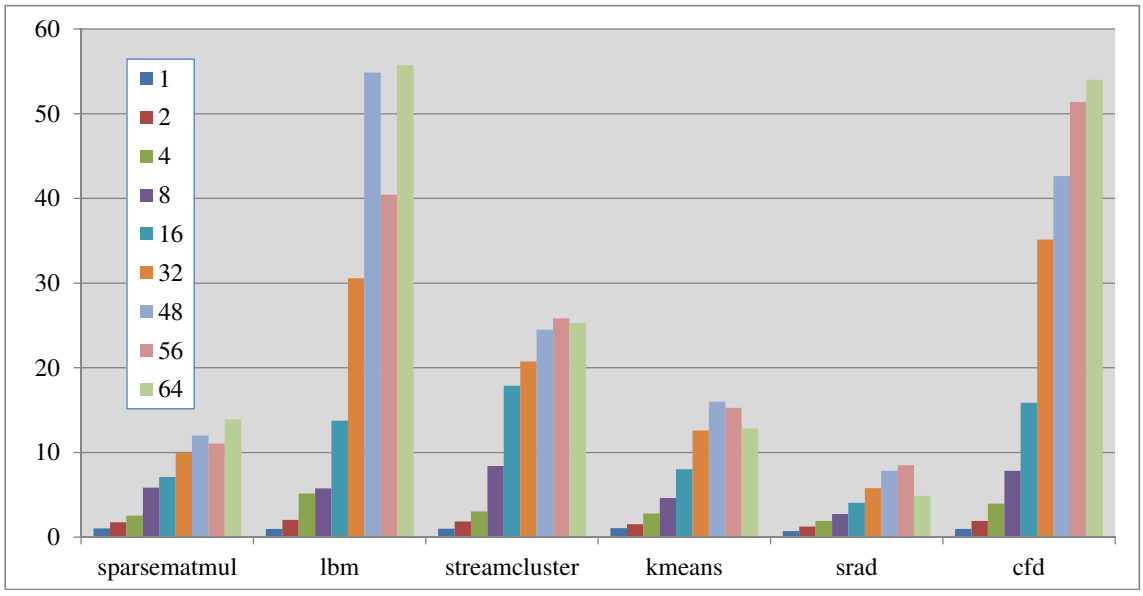


Figure 4–12: MUTLS/OpenMP Runtime ratio; higher is worse (3/4)

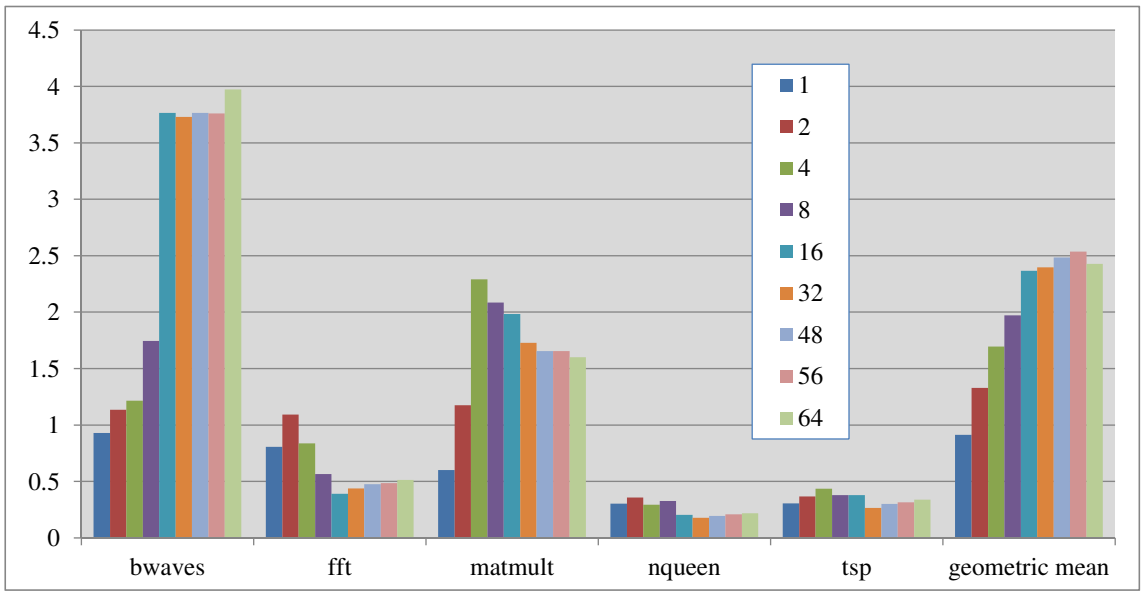


Figure 4–13: MUTLS/OpenMP Runtime ratio; higher is worse (4/4)

hand, the extremely memory intensive benchmarks lbm, streamcluster and cfd perform much worse than the OpenMP versions. It is also interesting to note that the tree-form recursion benchmarks fft, nqueen and tsp achieve significantly higher speedups than the OpenMP version, as a result of the efficient mixed forking model implementation. For other loop-based benchmarks, generally less memory intensive workloads have better performance ratios with respect to the OpenMP versions. On average, the benchmarks have a factor of around 2.37 to 2.54 slowdowns (39% to 42% performance) with respect to the OpenMP manually parallelized versions using 16 to 64 cores.

4.4 Analysis of Parallel Execution

We are more concerned with the non-speculative thread since (1) it is on the critical path, (2) it does not rollback, and (3) it is the fastest thread, since it does

not have speculative buffering overhead. As a result, we define two indicators: *critical path efficiency* which is the useful work time divided by the runtime of the non-speculative thread $\eta_{crit} = T_{worktime\_nonsp}/T_{runtime\_nonsp}$, and *speculative path efficiency* which is the sum of the work time divided by the sum of the runtime of the speculative threads $\eta_{sp} = T_{worktime\_sp}/T_{runtime\_sp}$. The two efficiencies are illustrated in Figures 4–14 and 4–15.

We can see that the benchmarks can be clustered into 3 categories with respect to critical path efficiency: (1) bh, raytracing, smallpt, 3x+1, mandelbrot - C, matmult, nqueen, tsp, lavaMD, heartwall and myocyte that maintain high critical path efficiency ($\eta_{crit} > 0.88$) with any number of CPU cores; (2) mandelbrot - Fortran and md and with slowly decreasing critical path efficiency; (3) fft, bwaves, kmeans, srad, sparsematmul, cfd, streamcluster and lbm that sharply decrease critical path efficiency at few cores and then gradually stabilize to a low efficiency. The geometric mean of the critical path efficiency over all benchmarks is 0.48, which we will improve by reducing the critical path serial validation/commit time in Chapter 5.

According to speculative path efficiency, the benchmarks can also be sorted into 3 categories: (1) bh, raytracing, smallpt, 3x+1, mandelbrot, md, nqueen, tsp, lavaMD, kmeans, heartwall and myocyte that have high speculative path efficiency ($\eta_{sp} > 0.70$) for more than 8 cores; (2) sparsematmul, bwaves, fft, matmult and srad that stabilize to medium speculative path efficiency ($0.32 < \eta_{sp} < 0.51$); (3) streamcluster, cfd and lbm with low efficiency ($\eta_{sp} < 0.14$) at 64 cores. The many benchmarks have lower speculative path efficiency with few (less than 8) cores is because of the thread task optimization: when the non-speculative thread joins a

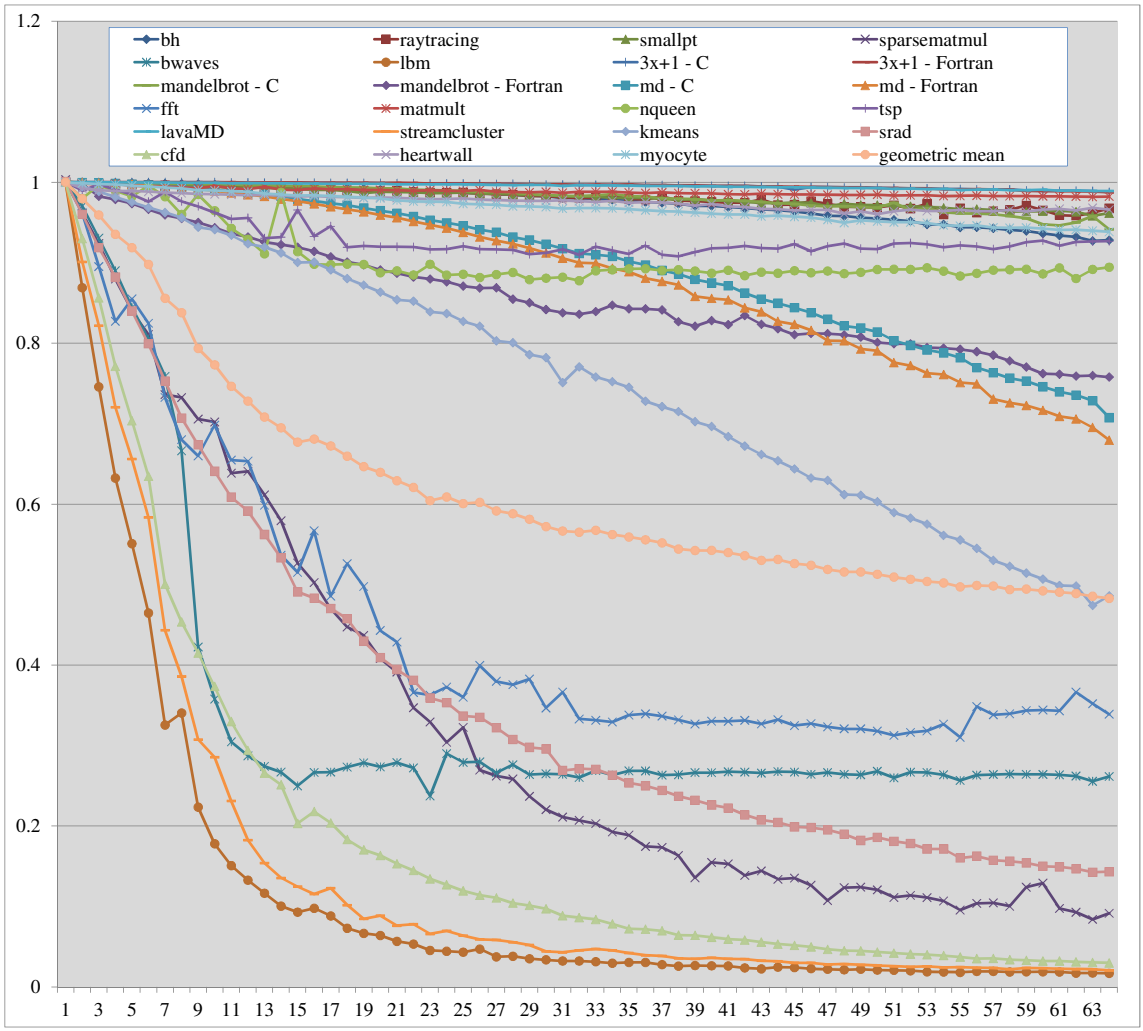


Figure 4–14: Critical Path Execution Efficiency; higher is better

speculative thread, if there is a pending thread task after the last loop iteration, the pending task then forks a thread executing the continuation of the loop, which is always idle at the loop barrier. The geometric mean of the speculative path efficiency for all benchmarks is 0.81 at 64 cores.

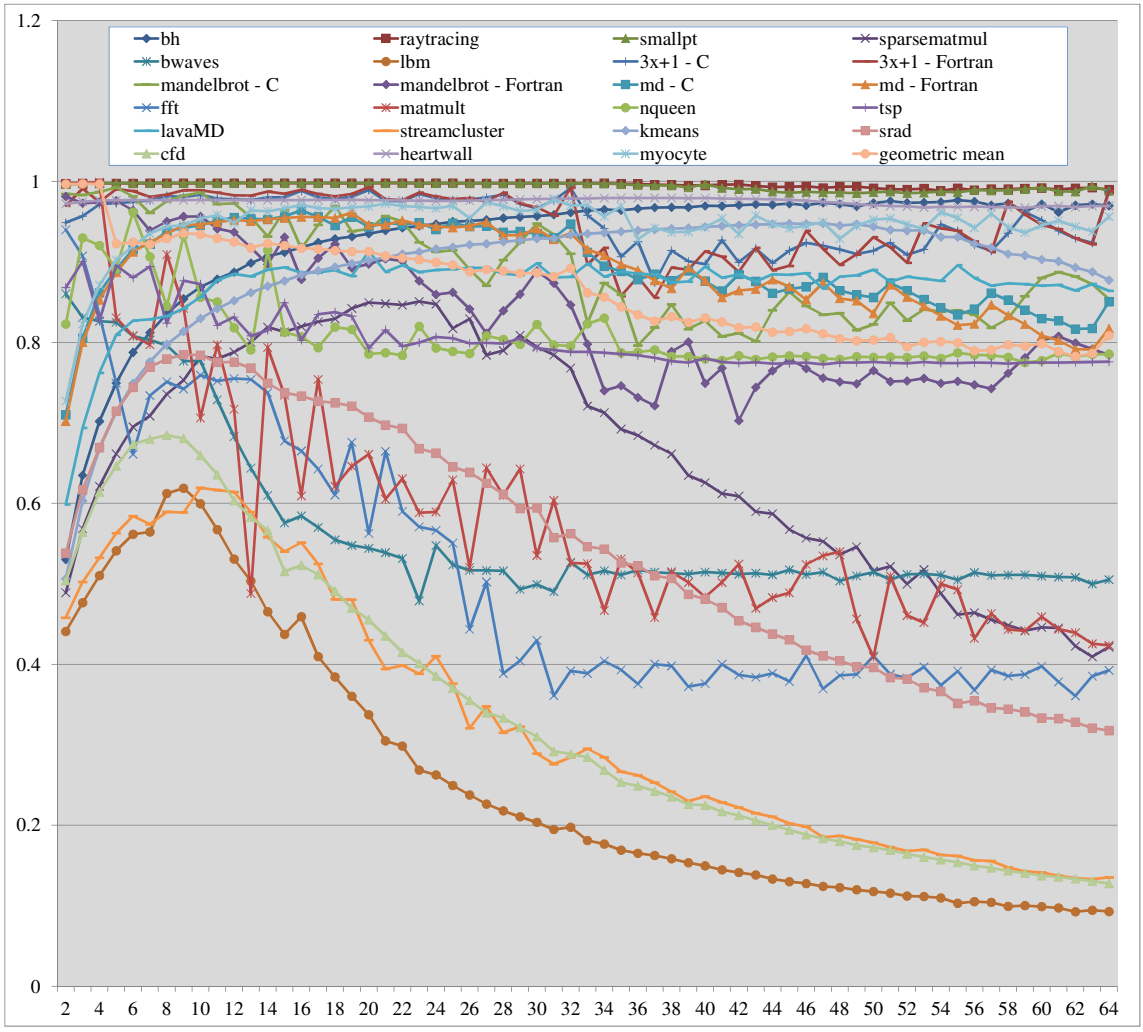


Figure 4–15: Speculative Path Execution Efficiency; higher is better

As expected, all the benchmarks achieving relatively high (larger than 4) speedups, including bh, raytracing, smallpt, 3x+1, mandelbrot, md, nqueen, tsp, lavaMD and myocyte, have both high critical and speculative path efficiency, while the slowdown benchmarks cfd, streamcluster and lbm have both low critical and speculative path efficiency. Other benchmarks with low critical path efficiency have low speedups,

such as fft, bwaves, kmeans and srad, or slowdown, such as sparsematmul. The matmult benchmark has high critical path efficiency (larger than 0.98), illustrating the benefit of data reuse (spatial locality) of efficient memory buffering using the block-based approach. However, its medium speculative path efficiency weakens its speedups.

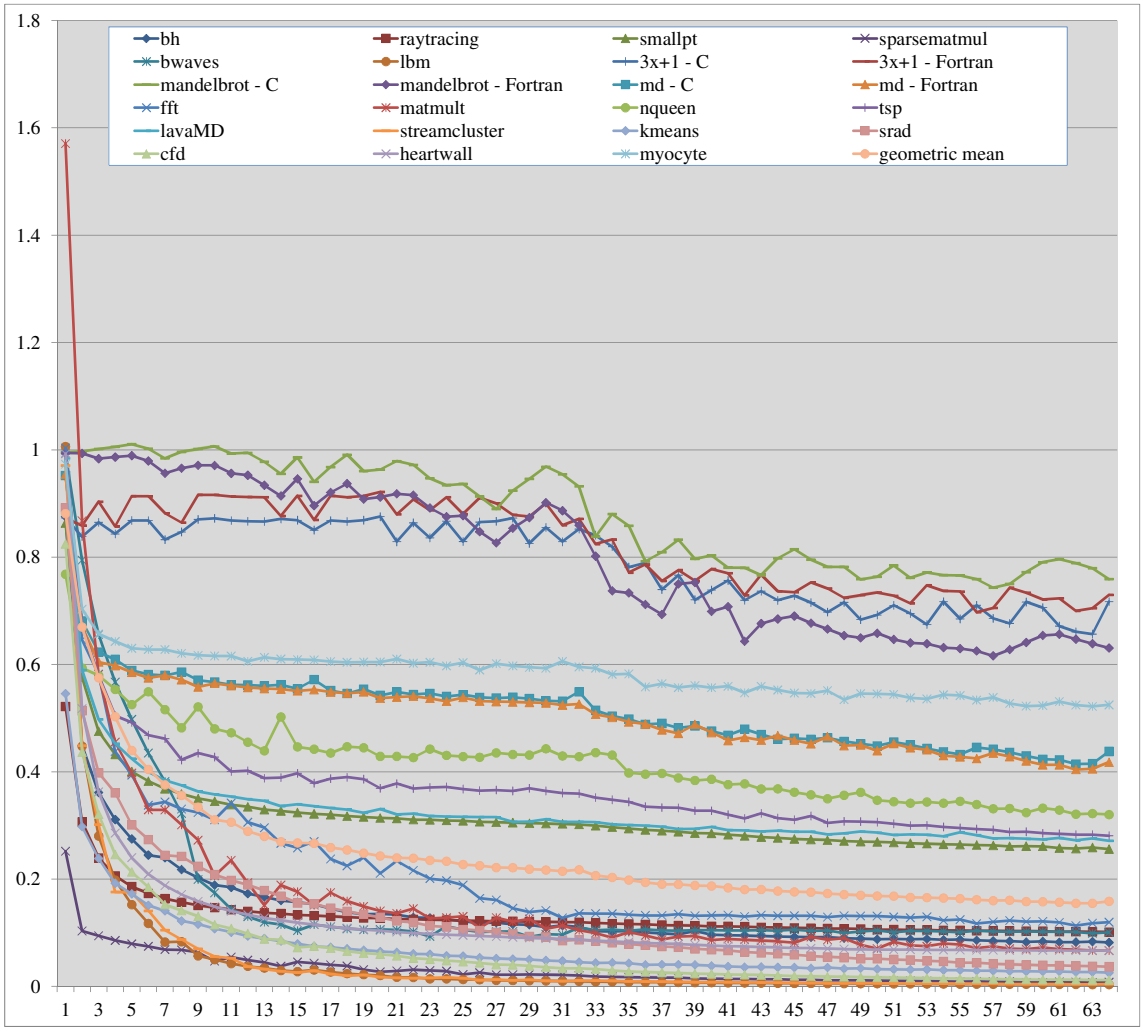


Figure 4–16: Power Efficiency; higher is better

Power consumption is a major concern in modern systems as well. We approximate power *efficiency* as the sequential runtime divided by the sum of the runtime of the non-speculative and all speculative threads $\eta_{power} = T_s/(T_{runtime\_nonsp} + T_{runtime\_sp})$, giving us an inverse measure of relative waste. This is shown in Figure 4–16. It can be seen that applications with higher speedups usually have higher power efficiency. The 3x+1 and mandelbrot benchmarks have highest power efficiency of 0.63 to 0.76 at 64 cores, while slowdown ones have the lowest of 0.0035 to 0.012. The geometric mean of all benchmarks is 0.16, which we will improve in Chapter 5. The matmult benchmark has a power efficiency of 1.57 at 1 core, showing that the TLS transformation can sometimes improve the performance and power efficiency for sequential programs.

The parallel execution coverage is defined as the sum of the speculative path runtime divided by the critical path runtime $C = T_{runtime\_sp}/T_{runtime\_nonsp}$. All our benchmarks have significant parallel execution coverage of 14.4 to 61.1, as expected of the mixed forking model.

To further understand the overhead, we breakdown the executions of some typical benchmarks from each category of the critical and speculative path efficiency. The results are presented in Figures 4–17 and 4–18. The execution time percentages of the "average" results in Figure 4–17(f) and 4–18(f) are computed as the arithmetic mean of the corresponding percentage of all the benchmarks. We cannot use geometric mean as the execution time percentage of one part would be eliminated if the percentage is 0 in only one benchmark.

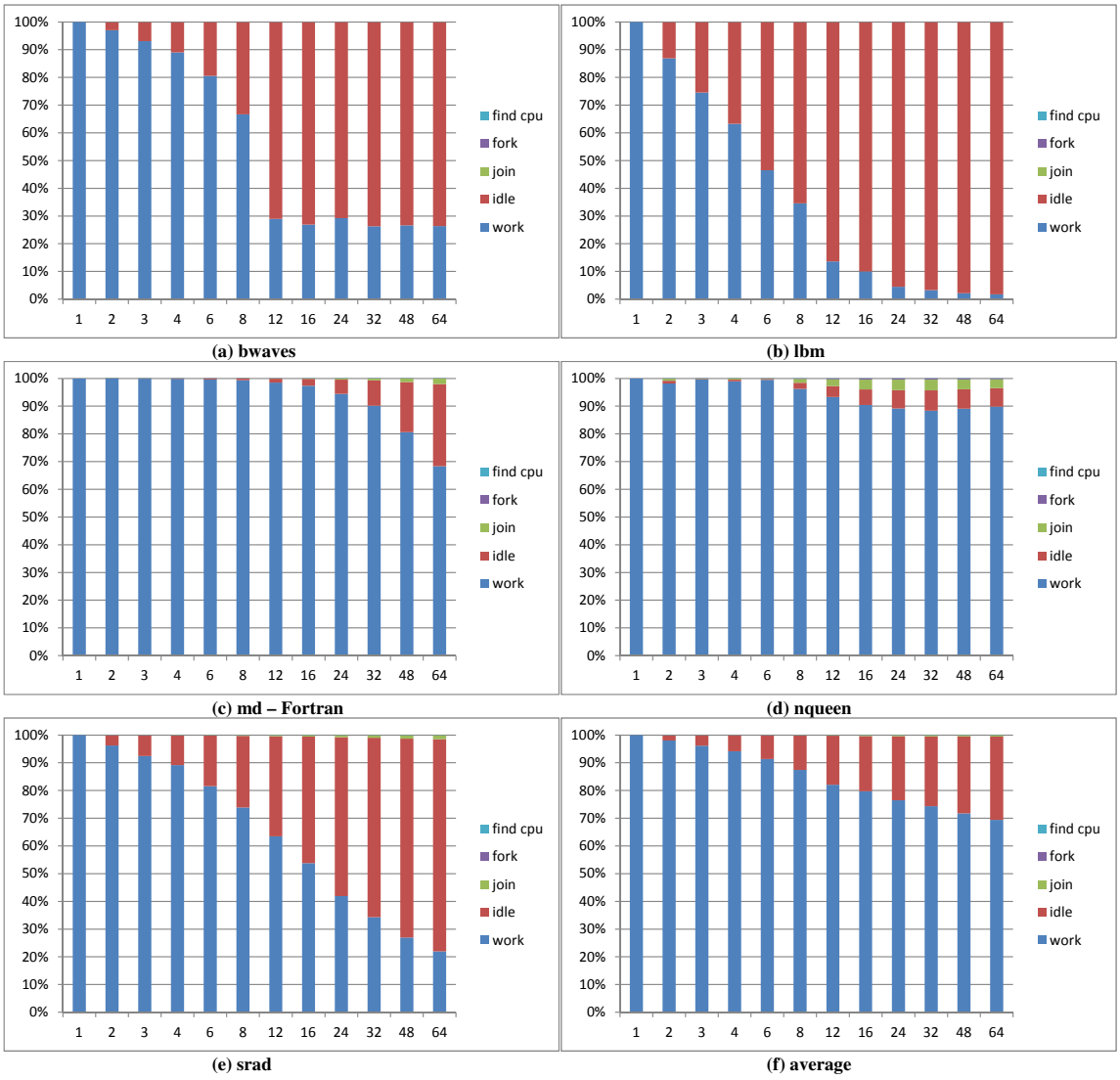


Figure 4–17: Critical Path Runtime Breakdown

It can be seen that almost all overhead of the critical path can be attributed to the "idle" time spent on polling and synchronizing with the speculative threads as was discussed in section 3.3, mainly waiting for them to validate and commit their memory buffers. The "join" time is the thread joining time after the thread

polling synchronization, mainly including the thread status maintenance time of section 3.4 and the stack frame reconstruction time of section 3.7. We can see that most benchmarks have small "find CPU", "fork" and "join" time, demonstrating the thread fork/join and stack frame reconstruction implementations are efficient, which is encouraging. For the nqueen benchmark, a significant amount of time is spent on thread joining, for two reasons: (1) a significant number of threads are speculated as a result of the mixed forking model, and (2) the thread joining of the benchmark usually reconstructs deep levels of recursive nested call stack frames. The md - Fortran and srad benchmarks also have notable thread joining time, since they speculatively parallelize loops within an outer loop iteration and thus fork/join large numbers of threads.

The speculative path is more interesting. For the loop workloads bwaves, lbm and kmeans, the thread task optimization causes significant thread idle time even with few cores, as was discussed for Figure 4–15. The reason that the percentage of idle time first decreases with more CPU cores for lbm and kmeans is that we blockize the loops to distribute the loop iterations evenly to the cores for these benchmarks, and thus the last iteration execution time is smaller with more cores. Then with more cores, the idle time increases, due to the fact that some speculative threads complete execution and wait to join the non-speculative thread while other speculative threads are validating/committing during the serial commit process. We could not speculate more threads to use the idle cores as the loop continuation usually has dependency with the loop iterations. For the divide-and-conquer fft benchmark, idle time also accounts for most of its time, up to 56% at 64 cores; this is partly

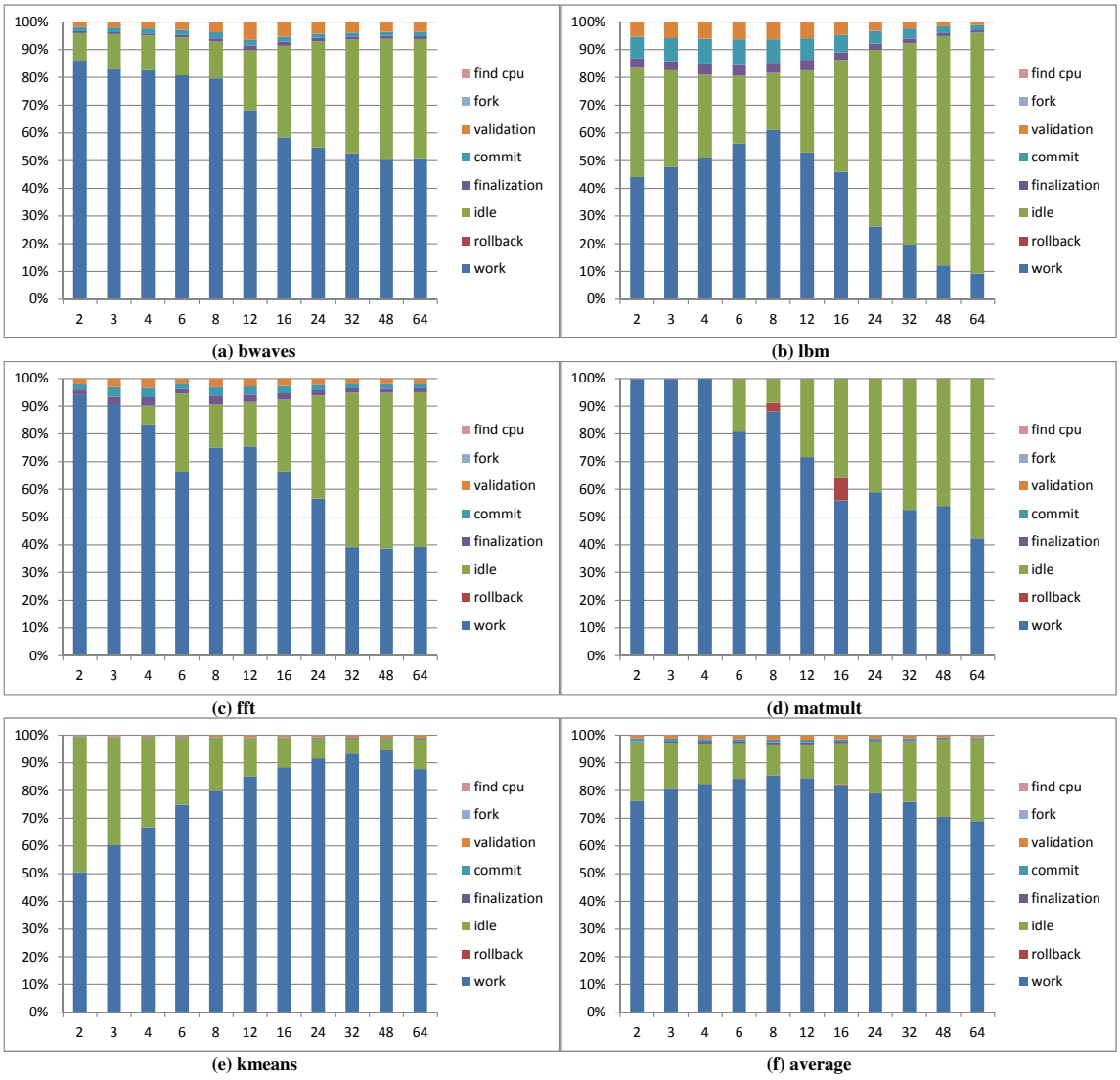


Figure 4–18: Speculative Path Runtime Breakdown

because we fork a thread to execute the second recursive call and barrier it after the call, preventing it from accessing parent data and causing unnecessary rollbacks. In our experiments, matmult is the only benchmark that exhibits rollbacks: although we split the computation into 4 sub-tasks each multiplying one sub-matrix, if the

sub-tasks split their own sub-tasks, then different "sub-sub-tasks" may read/write the same data and cause rollbacks. Like fft, though, idle time still dominates, again due to speculative execution barriers. It can also be seen that more memory-intensive benchmarks generally have larger validation/commit/finalization time.

4.5 Comparison of Forking Models

A comparison for different forking models is illustrated in Figure 4–19, normalized to the speedup of the full mixed model. We enable full optimization of the MUTLS framework design (stack frame and thread task optimizations, as discussed in section 4.2) for all these forking models.

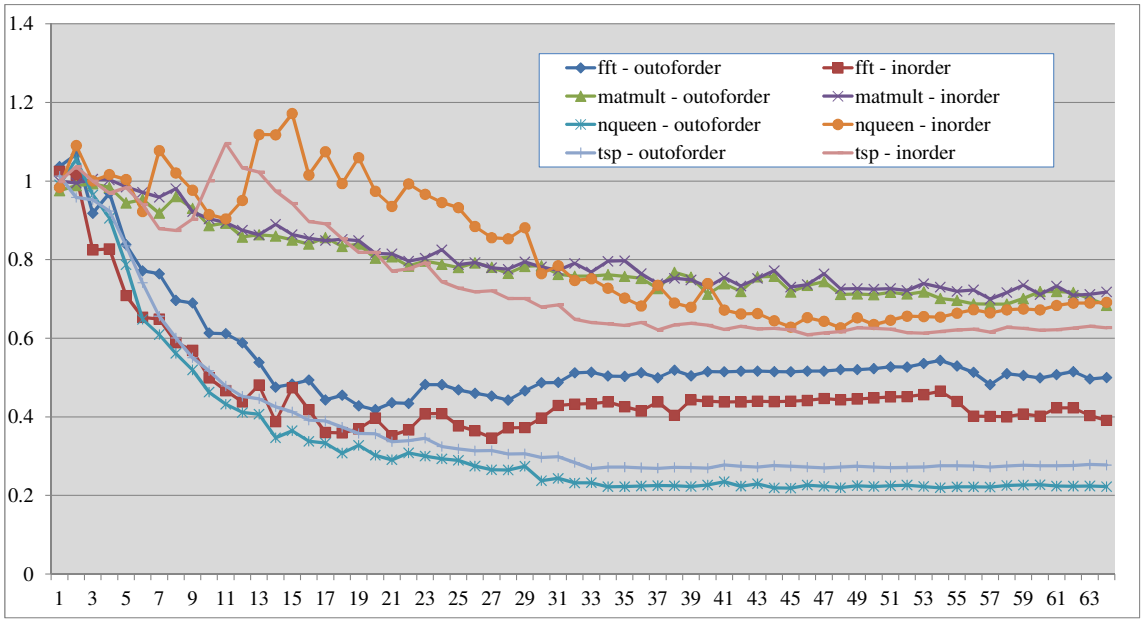


Figure 4–19: Comparison of Forking Models

With more than 4 cores, the mixed model beats in-order and out-of-order models for almost all benchmarks. The only exceptions are in-order nqueen from 5 to 19

cores and tsp from 10 to 13 cores. With no more than 4 cores, the fork models generally have limited effect on the performance of the benchmarks. It can also be seen that for divide-and-conquer benchmarks with few branches such as fft, the out-of-order forking model usually outperforms the in-order model, demonstrating the effectiveness of out-of-order model for method-level speculation. On the other hand, for benchmarks with significant amount of parallelism at the top frame level such as the depth-first-search (DFS) benchmarks nqueen and tsp, the in-order model is usually more beneficial.

4.6 Rollback Sensitivity

Thread rollback is always a concern in TLS systems. As the parallel execution of a rolled back speculative thread has to be executed sequentially by the non-speculative thread instead, rollback generally causes program performance degradation in a way we would expect from Amdahl's law: $Speedup = 1/(B + (1 - B)/P)$, where B is the percentage of the sequential execution time and P is the speedup factor of the parallel execution. Given the same parallel speedup factor $P > 1$, program performance would degrade more significantly if rollback causes a larger proportion of sequential execution B. On the other hand, for programs with similar non-rollback and rollback sequential execution proportions B, those with higher speedup factors P suffer from more severe performance degradation.

Since most of our benchmarks (other than matmult) do not generate rollacks due to their embarrassingly parallel properties, here we intentionally make the MUTLS system randomly cause rollbacks with specific probabilities in order to see the effect on performance. We do not validate a speculative thread if it is probability rolled

back, as validation usually takes a small amount of time if a speculative has memory dependencies. We characterize this as Rollback Sensitivity, which is the relative slowdown with respect to the non-rollback scenarios. The rollback sensitivity is experimented using 64 CPU cores, and computed as the geometric mean over 10 random runs. The results are shown in Figures 4–20 to 4–23.

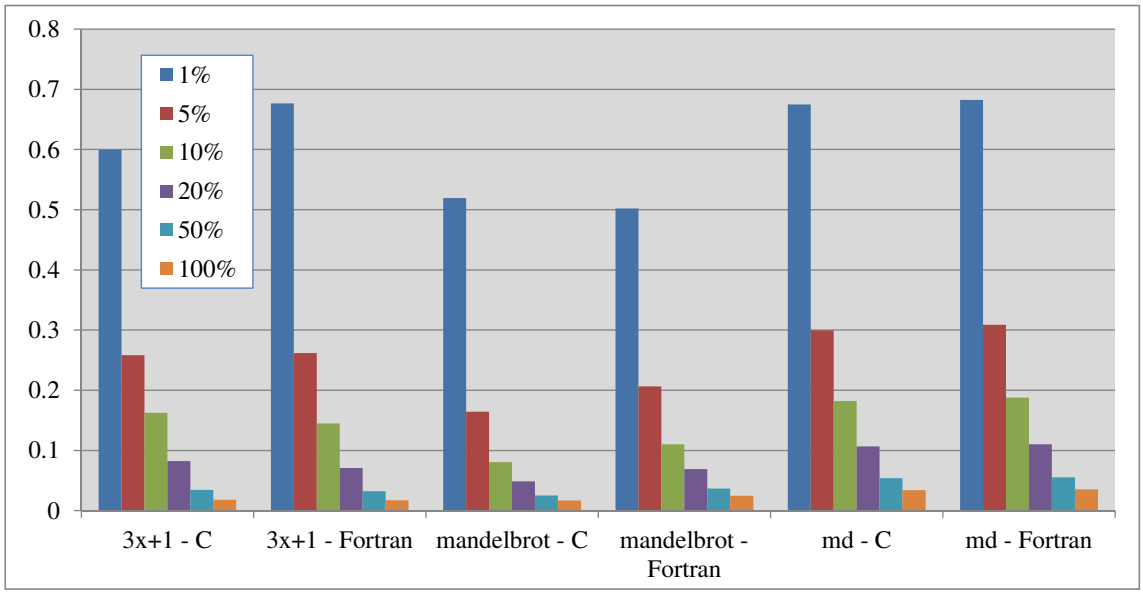


Figure 4–20: Rollback Sensitivity on 64 CPUs (1/4)

It can be seen that loop benchmarks with linear speedups are more sensitive to rollbacks with low rollback probabilities. On the other hand, the performance of tree-form recursion applications degrade significantly only with high probability rollbacks. This is as expected, since just one thread rollback in a loop speculative region would almost double the execution time of the speculative region, due to the fact that after rollback the non-speculative thread has to complete the current loop iteration before reaching the fork point to speculate another group of parallel

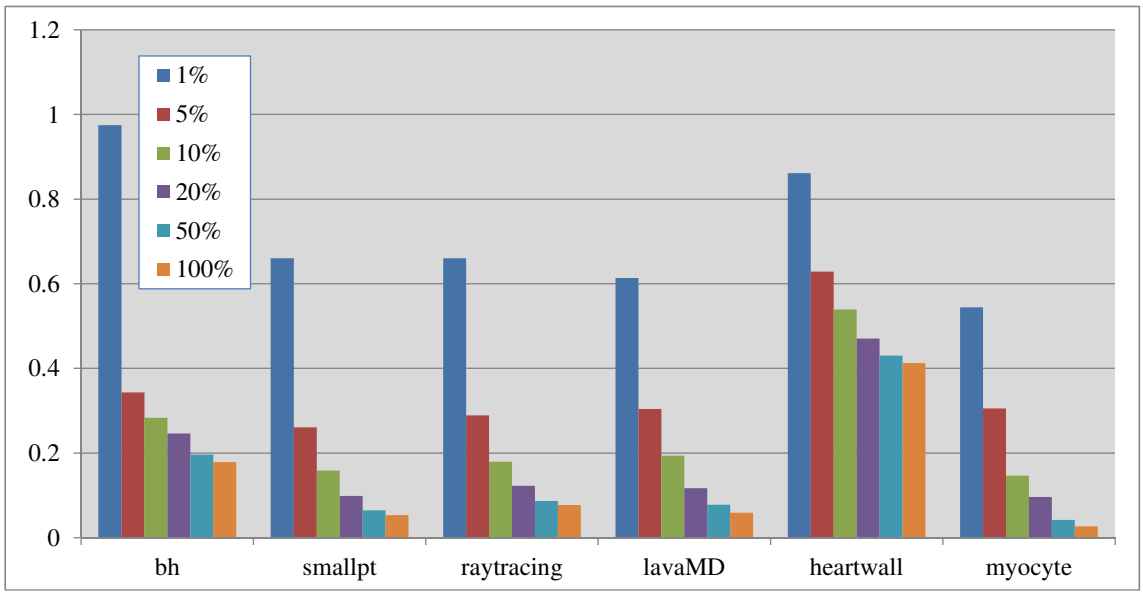


Figure 4–21: Rollback Sensitivity on 64 CPUs (2/4)

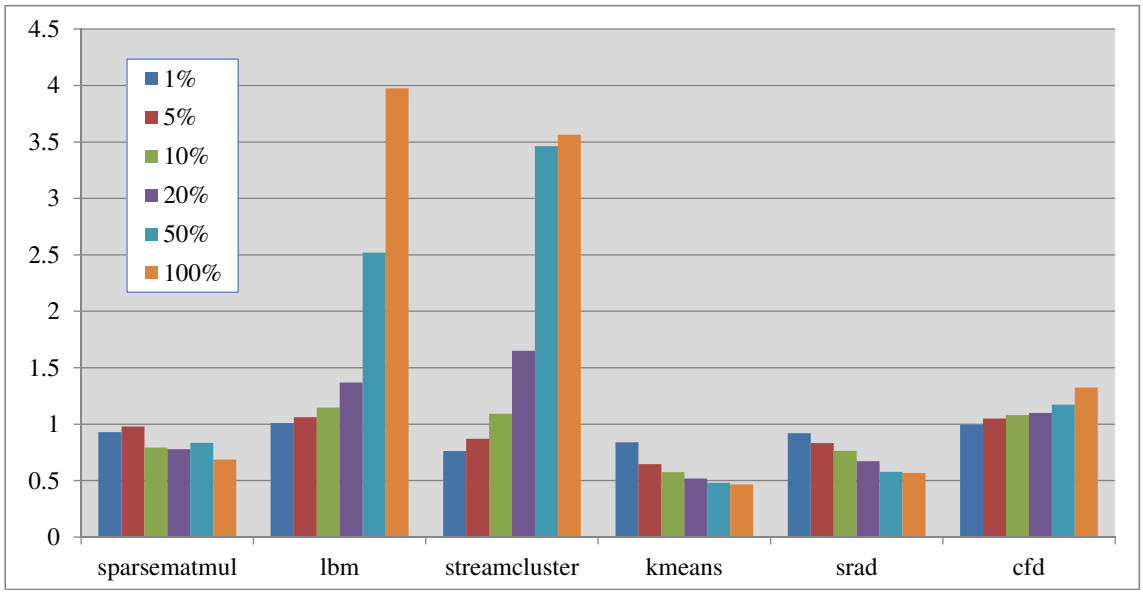


Figure 4–22: Rollback Sensitivity on 64 CPUs (3/4)

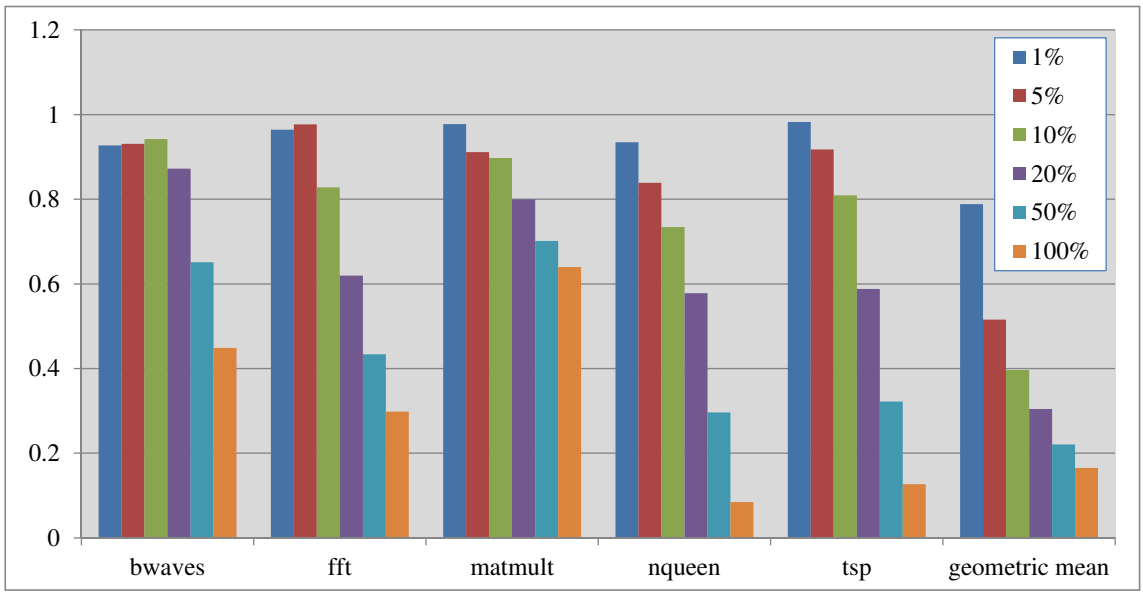


Figure 4–23: Rollback Sensitivity on 64 CPUs (4/4)

threads, which essentially splits the speculative parallel region to two disjoint parallel regions, causing significant load imbalance and/or parallel under-coverage. On the other hand, for tree-form recursion applications, even if a few speculative threads are rolled back, the performance would not degrade severely, as there are still significant parallelism opportunities in deeper recursive calls thanks to the mixed forking model. We can also see that programs with less speedups are generally less sensitive to rollbacks, which is expected from Amdahl's law.

The rollback overhead of the MUTLS framework is relatively low: the thread global buffer is just discarded and buffer finalization does not cause critical path delay, as was discussed in section 3.6.2. This is demonstrated in the figures: programs with low speedups do not have significant performance degradation even with 100%

rollbacks. Also, for some benchmarks with slowdowns, rollbacks can improve performance, as a result of reduced validataion/commit time delaying the critical path. On average, these benchmarks preserve 79% and 52% of the speedups with 1% and 5% rollbacks, respectively.

4.7 Chapter Summary

In this chapter, we first described the experiment environment and the benchmarks used in this and following chapters. Then we compared the speedups of the MUTLS framework design with the stack frame and the thread task optimizations, with the "stack frame & thread task" version achieving 22.3 to 49.1, 1.0 to 15.8 and 0.12 to 0.40 for computation intensive, locality and non-locality memory intensive benchmarks, respectively. Then we compared speedups categorizing the benchmarks into high speedups, mediocre speedups and slowdowns, which roughly corresponds respectively to the computation intensive, locality and non-locality memory intensive except for lavaMD and smallpt. For better understanding of the MUTLS software-TLS framework, we compared the performance of the MUTLS and OpenMP parallelized benchmarks, showing that the MUTLS versions have on average 41% performance of the OpenMP versions, and that memory intensive benchmarks do not achieve theoretically ideal performance and thus can be improved with further optimizations. We then analyzed the parallel execution and power efficiencies, and broke down the runtime to study the overhead of the MUTLS system. Afterwards, we compared different forking models, demonstrating that the mixed forking model usually achieves higher speedups for tree-form recursion applications. Finally, we

experimented with the rollback sensitivity of the benchmarks speculatively parallelized using the mixed forking model on the MUTLS system, showing that rollback performance characteristics can generally be expected from Amdahl's law.

CHAPTER 5
Memory Buffering Optimization

As can be seen from the results in Chapter 4, the MUTLS framework design achieves ideal performance for computation intensive benchmarks but not for memory intensive ones. The main cause is the memory buffering overhead, which both incurs high thread joining idle time that delays the critical path, and increases memory access cost that slows down the speculative path. We will address the memory buffering issue in this chapter.

Two approaches to software TLS memory buffering have been proposed: lazy version management (deferred updates) and eager version management (in-place updates) [125, 179]. Most software TLS systems, such as SableSpMT [133], SpLSC [124], Lector [179] and the MUTLS [45] design in Chapter 3 adopt the lazy version management approach, which buffers data accessed by speculative threads and employs a serial commit phase to validate/commit the speculative buffer to main memory. This design, however, has also been blamed for limiting scalability [125]. The large buffering costs inherent in the design of software isolation and validation mechanisms result in significant memory traffic, impacting cache performance and resulting in long validation/commit times. If there are a significant number of threads running, the serial commit phase may become a scalability bottleneck for memory intensive applications due to it delaying the critical path, something we do observe in our benchmarks.

Recent software TLS systems such as SpLIP [125] and MiniTLS [179] apply eager version management to address the problem, allowing speculative threads to directly access main memory and thereby eliminating the serial commit phase. In this approach, history versions of accessed data are maintained in shadow buffers and are used to recover the main memory state when rolling back offending threads. This comes with a trade-off, however, and despite the advantage of higher scalability, eager version management has the disadvantage of incurring expensive rollback overhead if dependency occurs, as well as causing rollbacks for all RAW, WAR and WAW dependencies. While lazy version management just discards the buffer if RAW dependency is detected.

Since lazy and eager version management have complementary weaknesses and strengths, it would be more effective to integrate them into one system to get the strengths of both [125]. Here we propose the first such software TLS buffering solution which can automatically determine which version management buffering to apply to which variables. This approach is integrated with and complemented by a number of other optimizations that reduce buffer size and improve buffer management. More specifically, in this chapter we describe the following contributions.

- To improve lazy version management, we propose a per-thread page-table memory buffering scheme that enables direct parallelization of the validation/commit (V/C) operations themselves. Our parallelized V/C takes advantage of extra processors still available once scalability has been saturated in order to reduce the overhead of one of the more important overhead concerns in lazy

buffering schemes. We also use vector processing to accelerate both address-space checking and memory-buffering V/C through common SIMD instructions, demonstrating application of both coarse and fine-grain parallelism as a means of helping the TLS system itself, and indicating an interesting optimization point exists in balancing the parallel resources applied to the base program with the resources used to implement that parallelism.

- For eager version management, we describe a shared address-owner memory buffering design where the space overhead is bounded by a constant factor of the program data size, as well as a buffer preserving optimization that reduces the buffer metadata clearing overhead at the beginning of each speculative region. The size of shadow buffers is a significant concern in eager approaches, and can limit the ability to exploit parallelism at larger granularity. Our design allows buffers to be allocated sufficiently large to enable speculation of any granularity without causing buffering overflow. Previous eager buffering approaches frequently clear/re-initialize buffering metadata, including at the beginning of each speculative region that incurs significant overhead. We propose a buffer preserving optimization to reduce the overhead.
- Significant reductions to data management costs are also possible if we know data is not changed at runtime, and so does not require temporary buffer space or need to participate in validation. Pure readonly data is uncommon and difficult to find in programs, but in a TLS context opportunities are improved by the fact that we only require variables be readonly during actual periods of overlapping speculative executions, and this has been the basis of previous

approaches that accurately find readonly variables with the help of profiler support [60] or through manual programmer specifications [124]. We describe a design that automates the process, using page markers to identify and optimize readonly and independent memory pages on-the-fly, giving us a less precise but low overhead and fully dynamic means of finding readonly data that can be applied to either buffering strategy.

- Finally, we propose a buffering integration mechanism that can automatically select the appropriate buffering technique for each variable. It can quickly identify variables as independent (without RAW, WAR, WAW dependencies, *i.e.* readonly or access disjoint memory) or not, and apply the optimized address-owner buffering for independent variables and the page-table buffering for dependent ones. In this way, we can benefit from the higher scalability of eager version management for independent variables, while still enabling TLS in the presence of dependent ones in a speculative region. This design includes adaptive buffering selection heuristics to dynamically choose the appropriate buffering based on the program execution characteristics.

In sections 5.1 to 5.3, we describe our optimized designs for lazy and eager buffering approaches and their adaptive integration. First, we present the page-table memory buffering for lazy buffering, which allows us to exploit coarse and fine grain parallelism in the validation/commit phase through the parallelized V/C and SIMD acceleration optimizations. Next, we describe the shared address-owner buffering that enables higher scalability and reduced buffer overflow in an eager design. For the eager buffering we also propose a buffer preserving optimization to reduce the

buffer metadata clearing/re-initialization overhead when entering each speculative region. Readonly data detection and adaptive buffering heuristics are related in their integration, and so are described together in section 5.3.

Note that we adopt page-based designs to both the buffering integration mechanism and the page-table thread memory buffering implementation. These pages, however, are independent of each other and have different characteristics: the former need only improve performance in the most common cases and thus is not required to be accurate, while the latter is expected to support a wide range of applications as long as they do not exhibit true RAW dependencies, and hence accurate, one-to-one mapping of each byte from the main memory to the buffering is important. The two page-based designs also serve different purposes. The former is to reduce the TLS system overhead by maintaining optimization meta-data at a coarser granularity with page-based data structures. The latter exposes coarse and fine grain parallelism to reduce validation/commit time for the thread memory buffering.

5.1 Lazy Per-Thread Page-Table Buffering

The basic MUTLS buffering approach does not buffer the main, non-speculative thread, following a lazy version management approach [45]. Page-table thread memory buffering implements this same behaviour, but organizing data to facilitate parallelization in the final V/C stage of each speculative thread. Note that these page-table buffer are thread-specific, and thus synchronization between speculative threads is not needed.

The page-table thread memory buffering implementation is detailed in Figure 5–1. The buffering maintains a page table (`table`, `pages`), a read-set (`markR`, `bufR`) and

```
1   class ThreadBuffer{
2     int rank;
3     char* table[PAGE_NUM];
4     array<size_t, PAGE_NUM> pages;
5     char markR[SIZE], bufR[SIZE];
6     char markW[SIZE], bufW[SIZE];
7   private:
8     size_t find_page(size_t addr){
9       size_t n = PAGE_NUM / K;
10      size_t index = (addr / PAGE) % n;
11      for(; index < PAGE_NUM; index += n){
12        if(table[index] == addr) return index;
13        if(table[index] == NULL){
14          table[index] = addr;
15          pages.push_back(index);
16          return index;
17        }
18      }
19      rollback(OVERFLOW, rank);
20    }
21  public:
22    template<typename T>
23    T load(T* addr);
24    template<typename T>
25    void store(T* addr, T data);
26    void commit();
27    bool validation();
28  };
```

rank: the rank of the thread (1 to N-1)
K: associativity of the hash mapping (4)
SIZE: the size of the thread buffer
PAGE: the size of each page (4KB)
PAGE_NUM = SIZE / PAGE

```
29  template<typename T>
30  T load(T* addr){
31    size_t p = find_page((size_t)addr);
32    size_t ofs = (p * PAGE) + (addr % PAGE);
33    if(*(T*)(markW + ofs) == (T)all_one)
34      return *(T*)(bufW + ofs);
35    if(*(T*)(markW + ofs) != 0)
36      rollback(PART_ACCESS, rank);
37    if(*(T*)(markR + ofs) == (T)all_one)
38      return *(T*)(bufR + ofs);
39    if(*(T*)(markR + ofs) != 0)
40      rollback(PART_ACCESS, rank);
41    *(T*)(markR + ofs) = (T)all_one;
42    *(T*)(bufR + ofs) = *addr;
43    return *addr;
44  }
45  template<typename T>
46  void store(T* addr, T data){
47    size_t p = find_page((size_t)addr);
48    size_t ofs = (p * PAGE) + (addr % PAGE);
49    *(T*)(markW + ofs) = (T)all_one;
50    *(T*)(bufW + ofs) = data;
51  }
52  bool validation(){
53    for(each p in pages){
54      if(!validate_page(p))
55        return false;
56    }
57    return true;
58  }
59  void commit(){
60    for(each p in pages)
61      commit_page(p);
62  }
```

Figure 5–1: Lazy Per-Thread Page-Table Memory Buffering

a write-set (`markW`, `bufW`) (Line 3–6). A memory store directly inserts the address-value pair to the write-set, while a memory load returns the buffered data from its write-set or read-set if found, and otherwise inserts the address and the read data to the read-set and returns the data. During validation, if the buffered data in the read-set is not equal to the main memory version, then RAW dependencies are detected

and the thread is rolled back; otherwise, validation succeeds and the thread commits all buffered data in the write-set to main memory.

The thread memory buffering can buffer at most *SIZE* bytes or *SIZE/PAGE* pages. The offset *ofs* of each byte of the read-set and write-set can be calculated as $ofs = i * PAGE + o$ given page index i and the offset o within the page. When buffering an address `addr`, the page index and offset within a page are calculated using hash mapping by the `find_page` method. To support the case that multiple addresses are hashed to the same page, `find_page` implements a K-way associative hash mapping, which considers the buffer to comprise K consecutive blocks and each address can be hashed to any of the K blocks. The `load` method calls `find_page` to find the page index (Line 31) and then computes the buffer offset of the address (Line 32). If the address is fully buffered in the write-set or read-set, the data is returned (Line 33–34, 37–38). If part is buffered, it means the program uses aliasing of different data types (unions), which we consider rare and in which case we simply rollback the thread (Line 35–36, 39–40). If the data is not buffered, it is inserted into the read-set and returned (Line 41–43).

In addition to exposing different granularities of parallelism, the page-table memory buffering has other advantages over previous software TLS buffering implementations. First, the memory buffer can accurately track dependencies with mixed load/store data types. Different threads accessing adjacent memory locations will not cause thread rollbacks, even for those accessing different integers or characters within the same 8-byte boundary on a 64-bit machine. Second, the K-way associative hash mapping can effectively reduce rollbacks caused by hashing conflicts

for programs with many variables such as memory allocation/deallocation intensive applications. However, we also note that these advantages come at costs. To support accurate dependency tracking, the average buffering space overhead for each word is $2 + \epsilon$ words for read-only or write-only variables and $4 + \epsilon$ for read-write variables, where ϵ is the paging overhead, compared to 2 (a buffered word and an address) for SpLSC and Lector. The K-way hash mapping also incurs performance overhead. Nevertheless, we find the benefits outweigh the overhead, as the page-table buffering is utilized in the fallback path expected to support general cases.

5.1.1 Parallelized V/C

To enable parallelized V/C and/or SIMD acceleration, it should be guaranteed that validation/commit within and/or across pages are independent and not subject to sequential ordering, which is the case for the page-table memory buffering. Other software-TLS approaches such as SpLSC [124] and Lector [179] do not enable these optimizations since in their designs multiple writes to the same address by a speculative thread are all stored in the write-set, and therefore must be committed serially to guarantee the last write is the last to commit. Also, the addresses in the write-set are not necessarily adjacent.

If a speculative thread accesses a large number of memory buffering pages, we can validate/commit the memory buffering pages in parallel to reduce the critical path idle time, which helps to achieve scalable speedups for memory intensive applications. Our current implementation needs additional CPUs other than those of the speculation runtime system for parallelized V/C, which is not an important problem for many-core machines. Also, power consumption issues may prevent all cores from

being utilized all the time, and this is an especially suitable scenario for the parallelized V/C optimization. For machines with fewer cores, dedicated V/C cores may be detrimental to performance, in which case we can use the same speculation cores as V/C cores while setting higher priority to V/C threads to prevent more threads delaying the critical path.

5.1.2 SIMD acceleration

Vector processing of data on consecutive memory locations can be accelerated by SIMD instructions such as SSE2, SSE4 and AVX2, and this can be applied to validation/commit and address space checking in our design, as the page-based validation/commit exposes fine grain parallelism opportunities for SIMD vectorization. Validation and commit of 16-byte data on a page using SSE4 intrinsics are demonstrated in Figure 5–2. SSE2 and AVX2 implementations are similar. One issue is that `_mm_maskmoveu_si128` needs a memory fence after buffering commit to ensure cache coherency, and this overhead may cause SIMD commit to be unprofitable.

```
bool compare_data(__m128i* p, __m128i *q,
      __m128i* mark){
   __m128i va = _mm_load_si128(mark);
   if(_mm_testz_si128(va, va)) return true;
   __m128i vp = _mm_load_si128(p);
   __m128i vq = _mm_load_si128(q);
   __m128i vc = _mm_cmpeq_epi8(vp, vq);
   return _mm_testc_si128(vc, va);
}
```

```
void commit_data(__m128i* p, __m128i* q,
      __m128i* mark){
   __m128i va = _mm_load_si128(mark);
   if(_mm_testz_si128(va, va)) return;
   __m128i vp = _mm_load_si128(p);
   _mm_maskmoveu_si128(vp, va, (char*)q);
}
```

Figure 5–2: Validation/Commit of 16-byte Data Using SSE4 Intrinsics

SIMD can also help improve the performance of MUTLS address space checking. Before buffering a memory load/store, MUTLS checks that the address is valid to

avoid segmentation fault. The addresses and sizes of global and heap variables as well as the non-speculative thread stack are registered and merged to form disjoint address space intervals, on which address space checking can be vector processed by SIMD instructions.

5.2 Eager Shared Address-Owner Buffering

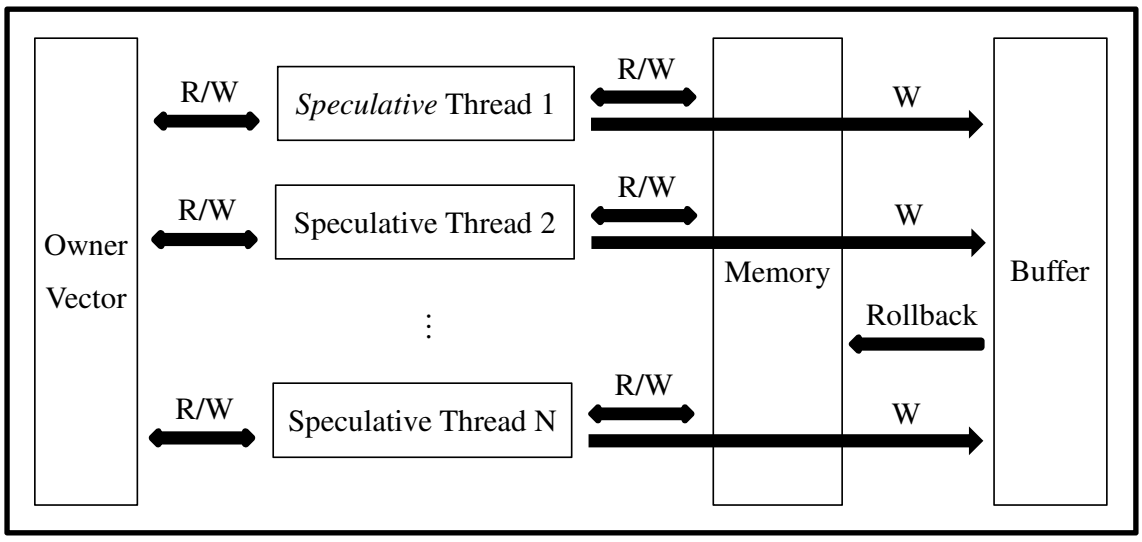


Figure 5–3: Shared Address-Owner Memory Buffering - Architecture

The address-owner buffering is an eager version management memory buffering with the property that a single buffer is shared by all threads. The architecture of the buffering is illustrated in Figure 5–3, and should be compared with the typical multiple buffer eager buffering design shown in Figure 2–9. Notably, our design uses a single shadow buffer for all threads, and its space complexity is proportional to the amount of data accessed, irrespective of the number of memory stores, as opposed to the multiple buffer designs of other software-TLS systems such SpLIP [125] and MiniTLS [179]. This ensures that the space overhead of the buffering is

bounded by a constant factor of the amount of program data, and further enables optimizations that can generally assume the buffering is sufficiently large that it has a one-to-one mapping of the main memory for any parallelism granularity, since allocating more memory by a constant factor is usually not a severe problem. As a result, K-way hash mapping implementation, and thus the page table, is not needed. As a trade-off, however, we need to include the non-speculative thread (Thread 1, with "Speculative" italicized) in the buffering design, in order to be able to detect interference between speculative threads and the non-speculative thread.

The granularity of the buffering is WORD; different threads accessing the same WORD with at least one writing are considered to have dependencies. WORD can be set as the native word of the machine, or smaller to achieve higher precision of dependency tracking. By our design, using smaller WORD such as 8, 16 or 32 bit is still as efficient as using larger WORD such as 64 bit for large word accesses, though smaller WORD overflows earlier as more threads are speculated and thus requires more frequent buffer flushes. However, for relatively large WORD such as 32-bit, overflow occurs once every 2^{31} threads, which is rare. By comparison, the SpLIP and MiniTLS systems need to flush buffer once every 256 and 32 threads, respectively.

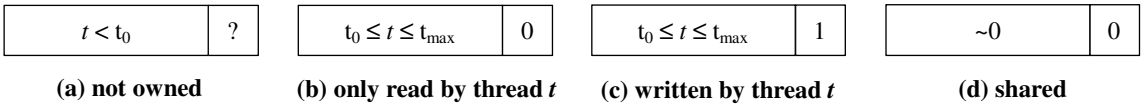


Figure 5–4: Shared Address-Owner Memory Buffering - Owner Number

For each WORD, the buffering maintains an *owner* for dependency tracking and a shadow memory copy for rolling back the WORD if dependency occurs. The encoding of the owner WORD is illustrated in Figure 5–4: the last bit of the owner

WORD is 1 if the buffering WORD has been written by a thread and thus is exclusively owned by the thread and 0 otherwise. Higher bits denote the thread accessing the WORD (the owner thread number of the WORD), or an all-one value if it is read by more than one thread (shared), in which case the last bit is 0. The non-speculative thread has the lowest thread owner number t_0. If the owner number t of the WORD is smaller than t_0, the WORD is not owned by any actively running thread (case (a), accessed by committed threads or not accessed by any thread); if t is the owner number of a running thread, the WORD is owned by the thread (case (b) or (c), non-exclusively or exclusively owned by the thread depending on whether the last bit is 0); if t is $\sim$0, the WORD is shared (case(d), can be read by all running threads).

To identify owner threads, speculative threads need globally unique thread owner numbers, for which the flexibility of arbitrary point speculation that is used by MUTLS and was discussed in section 2.2 imposes its specific difficulties. As fork/join points can be inserted anywhere in a function, it is not possible to use loop iteration numbers to identify threads, as pure loop-level speculation systems such as SpLIP and MiniTLS do. In addition, the non-speculative thread needs to update its thread owner number after joining speculative threads so that the non-speculative thread can access the WORDs owned by committed threads. We propose a thread owner numbering mechanism to resolve these issues. We maintain a global owner counter and for each thread an owner number. Each time a thread is speculated, we increment the counter and assign it to be the owner number of the speculative thread. After the non-speculative thread joins a speculative thread, the non-speculative thread

is assigned the owner number of the speculative thread. Since the non-speculative thread should have the lowest thread owner number of all actively running threads as was discussed for Figure 5–4, the joining speculative thread should have the lowest thread owner number of all running speculative threads. In order to guarantee this property, we use the in-order forking model of the MUTLS system, in which only the most speculative thread can fork a new thread.

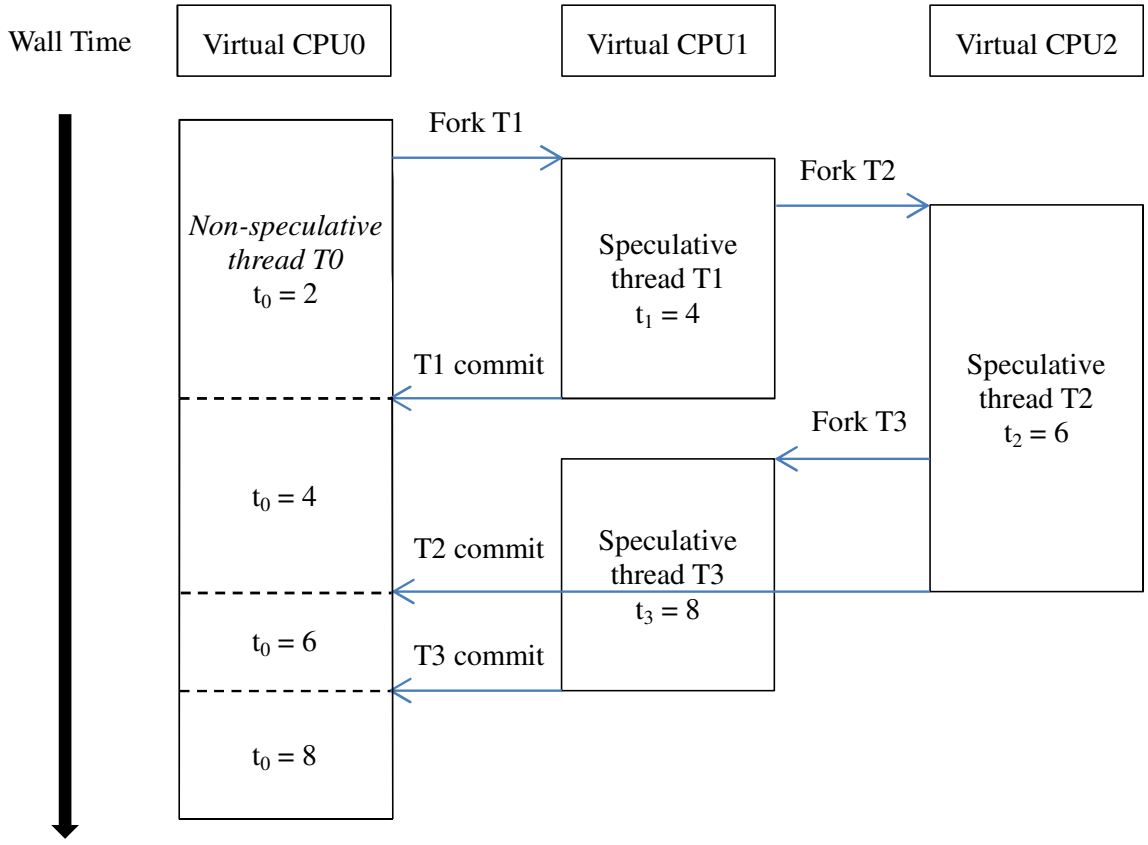


Figure 5–5: Shared Address-Owner Memory Buffering - Timeline

A running instance is demonstrated in Figure 5–5. The non-speculative thread T0 runs on CPU 0. The speculative threads T1, T2 and T3 are in-order speculated

running on virtual CPU 1, 2 and 1, respectively. The thread owner numbers are incrementally allocated to each thread as they are created, and therefore threads representing sequentially later execution have larger thread owner numbers. Since the non-speculative thread T0 then joins the speculative threads in the same order as they are speculated, it always joins the speculative thread with the lowest thread owner number. As can be noted in the figure, the non-speculative thread T0 always has the lowest owner number among the running threads (horizontally) anywhere in the (vertical) wall time-line.

The first time an address is written, the memory data is copied to the shadow buffer, so it can be restored to main memory if the speculative thread rolls back. It should also be guaranteed that the data *only* be copied the first time it is accessed, since we only maintain one version of each buffering WORD and should ensure the buffered data be the original memory version not written by any speculative thread. This enables the shared address owner buffering to use a single global shadow buffer with constant size and larger hash space, as opposed to the multiple buffer architecture shown in Figure 2–9.

The shared address-owner memory buffering implementation is presented in Figure 5–6. Thread owner numbers of speculative threads are assigned by `register_start_thread` (Line 26) and the non-speculative thread owner number `thread_owners[0]` is updated by `register_join_thread` (Line 27). The owner and shadow memory copy WORDs are stored in `owners` and `buf`, respectively. The `register_load` method returns true to indicate the load is valid if the buffering WORD of the address `addr` is shared or is already owned by the thread (Line 8). Otherwise it registers

the owner if the buffering WORD is not exclusively owned by another thread (Line 9–12); if it fails, then another thread is simultaneously registering the owner, in which case it tries to register the owner as shared if the buffering WORD is not registered as exclusively owned by another thread (Line 13). If it cannot register a valid owner, false is returned (Line 15) and rollback is initiated. The `register_store` method returns true if the buffering WORD is already exclusively owned by the thread (Line 19). Otherwise, it tries to register its owner (Line 20) and copy the memory value to the shadow buffer (Line 21). No other threads can register ownership of the buffering WORD again after the thread registers itself as an exclusive owner, thus guaranteeing the copy-only-once requirement of the buffering.

If a thread detects dependencies in the shared buffering, all speculative threads are rolled back, and then the buffering data is restored to the main memory. The `rollback_page_nonsp` method rolls back a buffering page of size `size` for address `addr`. For each buffering WORD in the page, if its owner number is larger than the non-speculative thread owner number and it is written, the shadow memory copy is restored to the main memory, and then the owner WORD is reset to 0 (Line 30–33). If a speculative thread is rolled back not as a result of the shared buffering dependency, e.g. attempting to call I/O functions, other speculative threads are not required to rollback since they cannot access variables written by the rolling back thread, although speculative threads representing sequentially later execution may need cascade rollback. When such a speculative thread is rolling back, it calls the `rollback_page_sp` method whose implementation is similar to `rollback_page_nonsp`, except that it determines whether to rollback a WORD by checking whether the

```
1   class AddressOwnerBuffer{
2     static const intptr_t SHARED = ~1LL;
3       char owners[SIZE], buf[SIZE];
4       WORD thread_owners[N], counter;
5   public:
6       bool register_load(WORD* addr, int rank){
7           WORD* p = (WORD*)(owners + (addr & SIZE)), owner = *p, t = thread_owners[rank];
8           if(owner == SHARED || (owner & SHARED) == t) return true;
9           if((owner & 1) == 0 || owner < thread_owners[0]){
10              T new_owner = (owner < thread_owners[0] ? t : SHARED);
11              T o = __sync_val_compare_and_swap(p, owner, new_owner);
12              if(o == owner) return true;
13              if((o & 1) == 0 && (__sync_val_compare_and_swap(p, o, SHARED) & 1) == 0) return true;
14          }
15          return false;
16      }
17      bool register_store(WORD* addr, int rank){
18          WORD* p = (WORD*)(owners + (addr & SIZE)), owner = *p, t = thread_owners[rank];
19          if(owner == (t | 1)) return true;
20          if((owner < thread_owners[0] || owner == t) &&__sync_bool_compare_and_swap(p, owner, t | 1)){
21              *(WORD*)(buf + ofs) = *addr;
22              return true;
23          }
24          return false;
25      }
26      void register_start_thread(int rank){ counter += 2; thread_owners[rank] = counter; }
27      void register_join_thread(int rank){ thread_owners[0] = thread_owners[rank]; }
28      void rollback_page_nonsp(WORD* addr, size_t size){
29          WORD* p = (WORD*)(owners + (addr & SIZE)), *q = (WORD*)(buf + (addr & SIZE));
30          for(size_t i = 0; i < size / sizeof(WORD); i++){
31              if(p[i] > thread_owners[0] && (p[i] & 1) == 1) addr[i] = q[i];
32              p[i] = 0;
33          }
34      }
35      void rollback_page_sp(WORD* addr, size_t size, int rank){
36          WORD* p = (WORD*)(owners + (addr & SIZE)), *q = (WORD*)(buf + (addr & SIZE));
37          for(size_t i = 0; i < size / sizeof(WORD); i++){
38              if((p[i] & SHARED) == thread_owners[rank]){
39                  if((p[i] & 1) == 1) addr[i] = q[i];
40                  p[i] = 0;
41              }
42          }
43      }
44  };
```

SIZE: the size of the shared address owner buffer

Figure 5–6: Shared Address-Owner Memory Buffering - Implementation

WORD thread owner number is equal to the speculative thread owner number (Line 38).

5.2.1 Buffer Preserving Optimization

Before entering each speculative region, the eager buffering should be cleared/re-initialized to avoid unnecessary rollbacks caused by the threads of the new speculative region accessing stale buffering metadata. This requires resetting the parts of the `owners` array of all global variables to 0, which has significant overhead, especially for benchmarks frequently entering speculative regions, for example, those iteratively computing parallel regions such as lbm.

We can reduce the buffer clearing/re-initialization overhead by the buffer preserving optimization. Before entering a speculative region, we do not reset the `owners` array but just call `register_start_thread(0)` to set the non-speculative thread owner to be higher than the owner numbers of previous speculative threads of the speculative region, which can simply be achieved by increasing the owner number of the non-speculative thread. From the discussion of Figure 5–4 we can see that this essentially sets all non-shared owners to be case (a) (not owned by any threads that will be speculated in the speculative region). The shared owner of case (d) is not affected as it does not consider the owner number. However, shared owners generally cause rarer unnecessary rollbacks as speculative regions usually access common shared data.

5.3 Readonly-Page Optimization & Buffering Integration

Speculative regions often contain readonly variables, and this can include larger data structures such as matrices, trees, and graphs that consume significant buffer

space and thus validation time. Aggressively pruning out readonly data is thus worthwhile, and can be efficiently done at the page level. Our readonly-page optimization maps the address of each read to its address and optimizes the read by not buffering it if the page address is marked readonly. If a thread then attempts to write an address whose page is marked readonly, all speculative threads are rolled back.

Variable Definition	Memory Intervals	Variable Pages
char a[10000];	global variable id 1: [0x16000, 0x18710)	variable id 1 pages: [0x16, 0x18]

void insert_tree_node(tree* root, int data){
tree* node = (tree*)malloc(sizeof(tree));
…
}
	heap variable id 2: [0x3108, 0x3128)
	heap variable id 2: [0x5770, 0x5790)
	heap variable id 2: [0x20350, 0x20370)
	…
		variable id 2 pages:
		0x3, 0x5, 0x20, …

Figure 5–7: Variable Pages

In order for the optimization to not cause excessive rollbacks, we register the page addresses of each global and heap variable in the TLS runtime library, as demonstrated in Figure 5–7. If one page address of a variable is written by a thread, we mark *all* pages of the written variable as not-readonly, which can usually find all readonly and not-readonly pages within short amount of time after entering a loop speculative region. One issue to be considered is pointer-based heap data structures. If we consider different nodes to be different variables, the optimization would be ineffective: if a tree is not readonly, writing a node could only mark the single node not-readonly and each write would cause rollbacks. We solve this problem by treating each heap allocation call instruction as a single heap variable since the nodes of a data structure are usually allocated by the same instruction.

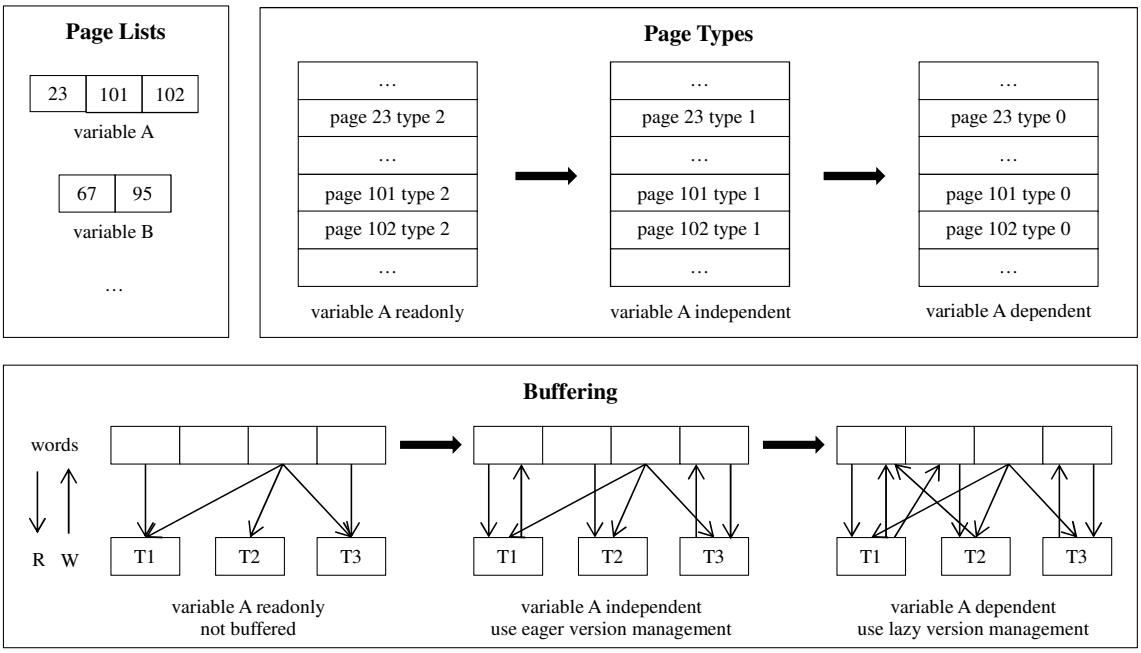


Figure 5–8: Buffering Example. Threads T1, T2 and T3 read/write variable A, which has pages 23, 101 and 102.

Our readonly-page optimization actually builds on a more general buffering mechanism that identifies pages as either readonly (2), independent (1), or dependent (0), using this distinction to help drive the choice of eager or lazy buffering. Before entering a speculative region, all pages are optimistically set to the default type (2, or readonly). If an address causes rollback and its page type is not at 0 (dependent), all pages of the variable that contains the address are reduced to a lower type. Figure 5–8 includes an example: the variable A spans three pages 23, 101 and 102, which are initialized to 2 before entering the speculative region. After entering the speculative region, the non-speculative thread T1 and 2 speculative threads T2 and T3 read A, and as long as none write A the TLS runtime system assumes the

variable is readonly and does not buffer it. If A is written the speculative threads are rolled back and restarted with A set to be independent and buffered using eager version management buffering. This continues as long as T1, T2 and T3 only read each given word or read/write different words of A. When different threads write the same word of A, speculative threads again rollback and restart with A set as dependent using the lazy version management buffering.

To avoid unnecessary rollbacks, we apply the page type preserving optimization that does not reset the page types if the fork point id of the speculative region is the same as that of the last entered speculative region whose fork point is not disabled. This alleviates performance degradation for applications that iteratively compute speculative regions, such as the md and heartwall benchmarks. To prevent the eager buffering from slowing down the non-speculative thread within a non-parallelizable region, we use the sequential region optimization that the non-speculative thread checks at each memory access if it is the only running thread, and if so then skips the buffering integration mechanism and directly accesses main memory.

The memory buffering integration mechanism, which is also the top-level implementation of the memory buffering, is present in Figure 5–9. To efficiently find the variable of an address and all pages of a variable/memory allocation instruction, we maintain the address and size of each variable and the allocated variables of each memory allocation instruction by registering the variable pages in the buffering (Line 14–16). For each page, we also register all variables accessing the page. We reset the all pages before entering speculative regions (Line 18). When a speculative thread accessing an address of a non-0 type page causes rollback, it sets the `rbk_addr` to

```
1   class MemoyBuffering{
2     int page_types[PAGE_NUM], rbk_page_type;
3     char* page_addresses[PAGE_NUM], *rbk_addr;
4     array<size_t, PAGE_NUM> pages;
5     ThreadBuffer threads_buffer[N];
6     AddressOwnerBuffer owner_buffer;
7   private:
8     size_t get_page(char* addr){ return (addr & SIZE) / PAGE; }
9     int get_page_type(char* addr){
10      size_t p = get_page(addr), paddr = addr & ~(PAGE – 1); return (page_addresses[p] == paddr) ? page_types[p] : 0;
11    }
12  public:
13    void register_variable_node(char* addr, size_t size){
14      size_t p = get_page(addr), pe = get_page(addr + size);
15      for(; p <= pe; p++, addr += PAGE)
16        if(page_addresses[p] == NULL){ page_addresses[p] = addr & ~(PAGE – 1); pages.push_back(p); }
17    }
18    void reset(){ for(each p in pages) page_types[p] = 2; }
19    template<bool sp, typename T>
20    T load(T* addr, int rank){
21      if(is_self_stack_address(addr, rank)) return *addr;
22      int t = get_page_type(addr);
23      if(t == 2) return *addr;
24      if(t == 1){
25        if(owner_buffer.register_load(addr, rank)) return *addr;
26        if(sp) rollback_self_sp(addr, 0, rank); else rollback_self_nonsp(addr, 0);
27      }
28      return sp ? threads_buffer[rank].load(addr) : *addr;
29    }
30    template<bool sp, typename T>
31    void store(T* addr, T data, int rank){
32      if(is_self_stack_address(addr, rank)){ *addr = data; return; }
33      int t = get_page_type(addr);
34      if(t == 2) sp ? rollback_self_sp(addr, 1, rank) : rollback_self_nonsp(addr, 1);
35      else if(t == 1){
36        if(owner_buffer.register_store(addr, rank)) *addr = data;
37        else if(sp) rollback_self_sp(addr, 0, rank); else rollback_self_nonsp(addr, 0);
38      }
39      else if(sp) threads_buffer[rank].store(addr, data); else *addr = data;
40    }
41    void register_start_thread(int rank){ owner_buffer.register_start_thread(rank); }
42    void register_join_thread(int rank){ owner_buffer.register_join_thread(rank); }
43    void check_valid(){ if(rbk_addr == NULL) return; rollback_self_nonsp(addr, rbk_page_type); rbk_addr = NULL; }
44    void rollback_self_sp(char* addr, int type, int rank){
45      rbk_addr = addr; rbk_page_type = type; rollback(INVALID_PAGE, rank);
46    }
47    void rollback_self_nonsp(char* addr, int type){
48      rollback_all_speculative_threads(); var = find_variable(addr);
49      for(each p in var.get_pages()){ page_types[p] = type; owner_buffer.rollback_page(page_addresses[p], PAGE); }
50    }
51  };
```

SIZE: the size of the shared address owner buffer; PAGE: the size of each page (4KB); PAGE_NUM = SIZE / PAGE

Figure 5–9: Buffering Integration Implementation

the address and the `rbk_page_type` to the appropriate lower type and then rolls back itself (Line 45). When the non-speculative thread accesses an invalid page, it aborts all speculative threads, rolls back the eager address-owner buffering and resets the page types of the variable that contains the address (Line 48–49). Since rolling back each page of a variable is independent, the loop can be parallelized, and as there are no speculative thread running when the non-speculative thread rolls back the eager buffering, the parallelized loop can be scheduled to run on the speculative computation cores. The non-speculative thread also frequently calls `check_valid` and initiates thread rollbacks if it finds that a speculative thread has accessed invalid pages (Line 43).

5.3.1 Thread Stopping Optimization

To rollback and restart the speculative threads for the readonly-page optimization and the buffering integration mechanism, such as the example shown in Figure 5–8, we have different design choices. In the MUTLS framework design of Chapter 3, all speculative threads/tasks can be rolled back. When the non-speculative thread reaches a fork point again, a child thread is then forked to continue speculative parallel execution.

An example of this design is illustrated in Figure 5–10. The non-speculative thread T0 in-order speculates threads T1, T2, T3 and the pending task T4. Then a thread writes a readonly page and initiates speculative threads/tasks rollback-/restart, which causes all speculative threads/tasks to rollback. This design has the drawback of reduced parallel thread work coverage: after the speculative thread tasks are rolled back, the non-speculative thread has to complete the rest of its iteration

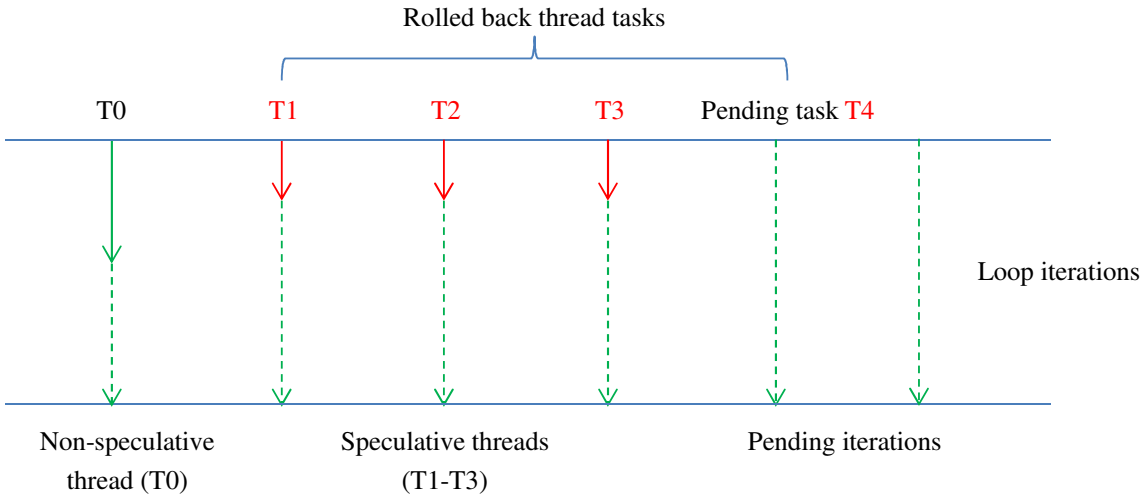


Figure 5–10: Speculative Threads/Tasks Rollback/Restart without Thread Stopping Optimization

before speculating a child thread again, resulting in unnecessarily longer program execution time.

We propose the thread stopping optimization to resolve this issue. When the speculative threads need to be restarted, the non-speculative thread initiates speculative threads/tasks stop/rollback/restart, which stops its direct children, and then indirect children are cascadingly rolled back. Then after rolling back the eager buffering and resetting the variable page types, the non-speculative thread restarts the stopped child thread tasks.

The example of Figure 5–10 with the thread stopping optimization is demonstrated in Figure 5–11. When the speculative threads/tasks T1 to T4 need to restart, the speculative thread T1 is stopped, which in turn cascadingly rolls back T2, T3 and T4. Then as there are no speculative threads running, the non-speculative thread T0 can maintain the buffering metadata and global main memory. Afterwards, it

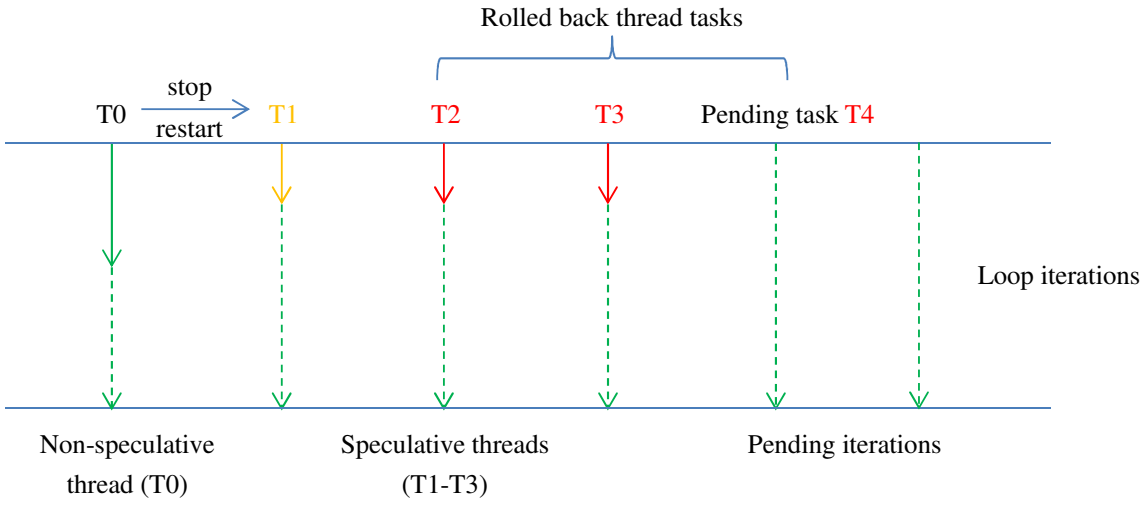


Figure 5–11: Speculative Threads/Tasks Stop/Rollback/Restart with Thread Stopping Optimization

restarts the stopped speculative thread task T1, which in turn re-speculates child threads/tasks T2, T3 and T4.

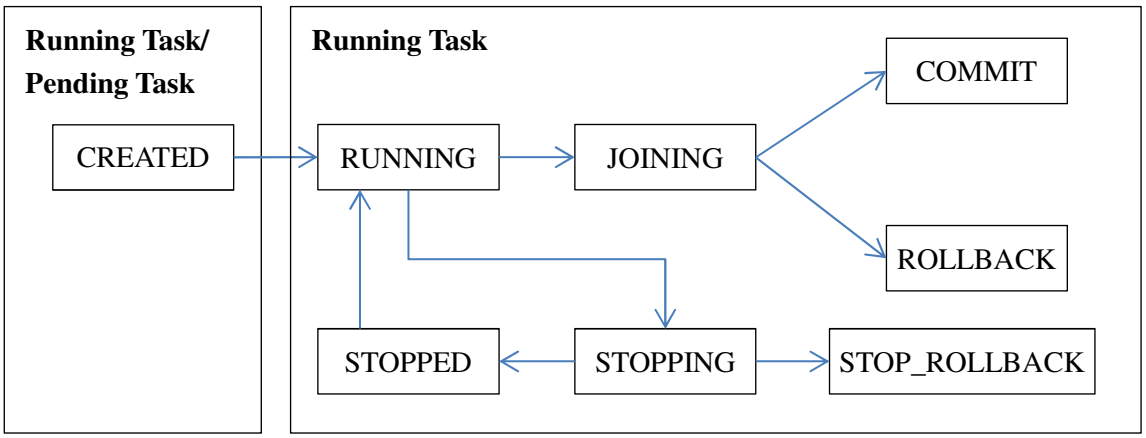


Figure 5–12: State Transition of Thread Stopping Optimization Design

The state transition of the MUTLS framework design with the thread stopping optimization is illustrated in Figure 5–12. When a speculative thread has an invalid

buffering memory access such as writing a readonly page or reading/writing an independent WORD and needs to restart, it waits for its state to be STOPPING. When the non-speculative thread has an invalid buffering memory access or finds one of a speculative thread in the `check_valid` call as was discussed for Figure 5–9, it sets the states of the speculative threads/tasks to STOPPING. If a speculative thread finds its state is STOPPING in a synchronization point as was discussed in section 3.5.3 or during waiting after an invalid buffering memory access, there are two cases, depending on whether the speculative thread is a direct child of the non-speculative thread: if it is, it sets its state to STOPPED and exits; otherwise, it transits to STOP_ROLLBACK and calls the `rollback_self_sp` method of Figure 5–9 to rollback itself. The reason that indirect child threads do not transit to the ROLLBACK state is that, instead of calling `rollback_self_sp` of Figure 5–9, a thread normally rolling back from the ROLLBACK state calls the `rollback_page_sp` method of Figure 5–6 for all its accessed pages, as was discussed for the figure. After the non-speculative thread rolls back the eager buffering and resets the buffering metadata, it resets the states of the stopped direct child threads to RUNNING and restarts the threads.

5.3.2 Adaptive Buffering Selection Heuristics

Since eager version management buffering has higher scalability, the buffering integration mechanism defaults to selecting the eager shared address-owner buffering whenever possible. However, when there are few processor cores, or the validation/-commit time is small, the way eager buffering delays the non-speculative thread

becomes the dominant overhead, and it can be more efficient to use the lazy page-table buffering. We thus propose adaptive buffering selection heuristics to address this problem.

The adaptive buffering selection heuristics is similar to the adaptive fork heuristics [44] that will be discussed in Chapter 6 in that it dynamically profiles the parallel program execution and adapts to the appropriate buffering on-the-fly. At the beginning of program execution, the lazy page-table buffering is the default selection since it does not cause unnecessary rollbacks for dependent variables. The runtime system records the number of memory accesses m, work time T_{work} (the time from being speculated to the start of validation/commit), and validation/commit time T_{vc} of each speculative thread. When a speculative thread commits, the expected runtime of lazy and eager buffering are estimated as follows. Assume each speculative memory access has overhead C cycles and the eager buffering delays each thread by a constant factor of K: if there are $N-1$ speculative threads, the work time speedup of the N threads for the lazy buffering is $S = 1 + (N-1) * (T_{work} - C * m)/T_{work}$. For an assumed workload of L loop iterations, the estimated runtime of the lazy and eager buffering is then $t_{lazy} = L * T_{work}/S + L * T_{vc}$ and $t_{eager} = L * T_{work} * K/N$, respectively. If $t_{lazy}/t_{eager} = N/K * (1/S + T_{vc}/T_{work}) > 1$, then the eager buffering is considered more efficient and should be enabled. In our experiments, we use parameters $C = 20$ and $K = 8$.

5.4 Experiments

We perform experiments of the buffering optimizations using the MUTLS framework design with the highest speedups that enables the stack frame and thread task

optimizations, as shown in section 4.2. Since the eager shared address owner buffering requires the in-order forking model, as discussed in section 5.2, we perform two groups of experiments for each measurement in the following subsections, according to whether the eager buffering is used.

As more parallel thread work generally requires more validation/commit (V/C) time, we use more cores for parallelized V/C as more cores are used for speculative parallelization. For the experiment, we use N_{pvc} =1, 1, 2, 3, 4, 6, 8, 8 and 8 cores for parallelized V/C, when there are N =1, 2, 4, 8, 16, 32, 48, 56 and 64 available CPU cores, thus leaving for working threads N_{work} =1, 2, 3, 6, 13, 27, 41, 49 and 57 cores, respectively. The reason that $N_{pvc} + N_{work} = N + 1$ is because the non-speculative thread can also be used for parallelized V/C and thus save one dedicated V/C core. We use SSE4 instructions for SIMD acceleration.

5.4.1 Speedup of Lazy Page-Table Buffering

We first experiment with the effectiveness of the lazy per-thread page-table buffering optimizations. The speedups are presented in Figures 5–13, 5–14 and 5–15. The *noopt* curves show the speedups of the non-optimized lazy buffering of the MUTLS framework design (with the stack frame and thread task optimizations) in Chapter 3. The *simd*, *pvc* and *ro* mean the SIMD acceleration, parallelized V/C and readonly-page optimization, respectively.

It can be seen that the readonly-page optimization is highly efficient and effective for benchmarks with large readonly variables, significantly improving the

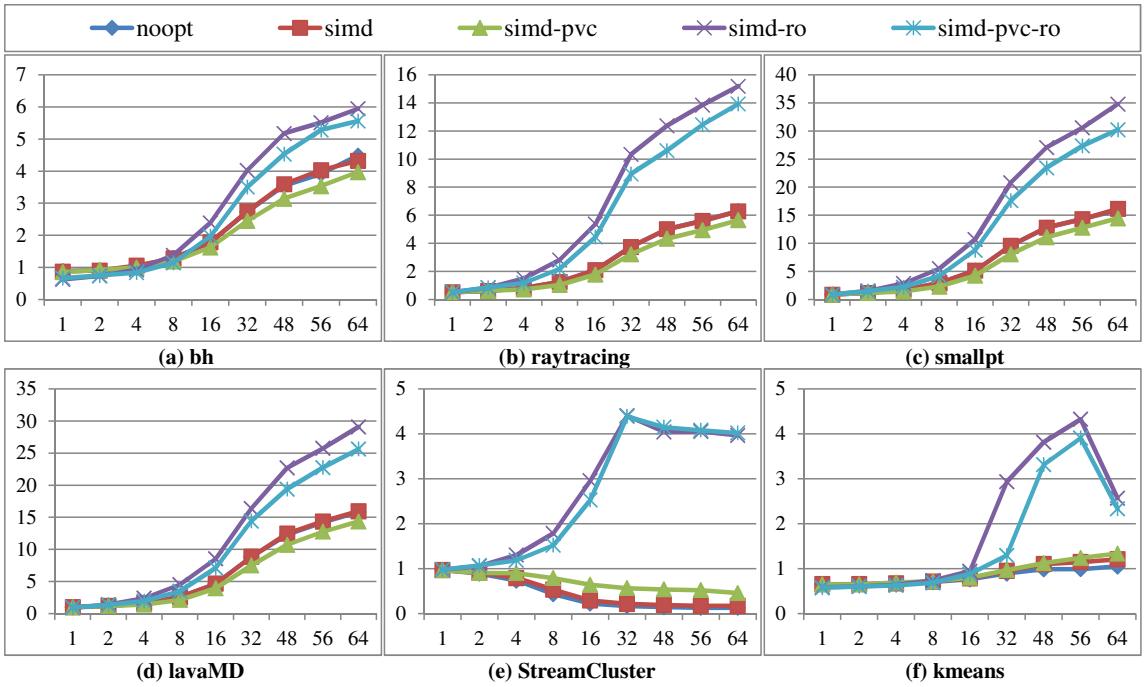


Figure 5–13: Speedups of Lazy Buffering; higher is better (1/3). The readonly-page optimization is most effective for benchmarks with large readonly variables.

performance of the bh, raytracing, smallpt, lavaMD, streamcluster, kmeans, sparsematmul, bwaves, srad, cfd and heartwall benchmarks. It also shows improvement for other benchmarks with generally negligible overhead.

The SIMD acceleration optimization improves performance considerably for bwaves, fft, srad and md, and moderately for others. The parallelized V/C optimization significantly benefits more memory intensive benchmarks such as sparsematmul, bwaves, lbm, fft, srad, cfd and streamcluster, as a result of reduced critical path delay, while degrading the speedups of computation intensive benchmarks such as bh, raytracing, smallpt, 3x+1, mandelbrot, lavaMD and myocyte due to less thread resources allocated for speculative parallel computation. The bwaves, fft, srad and

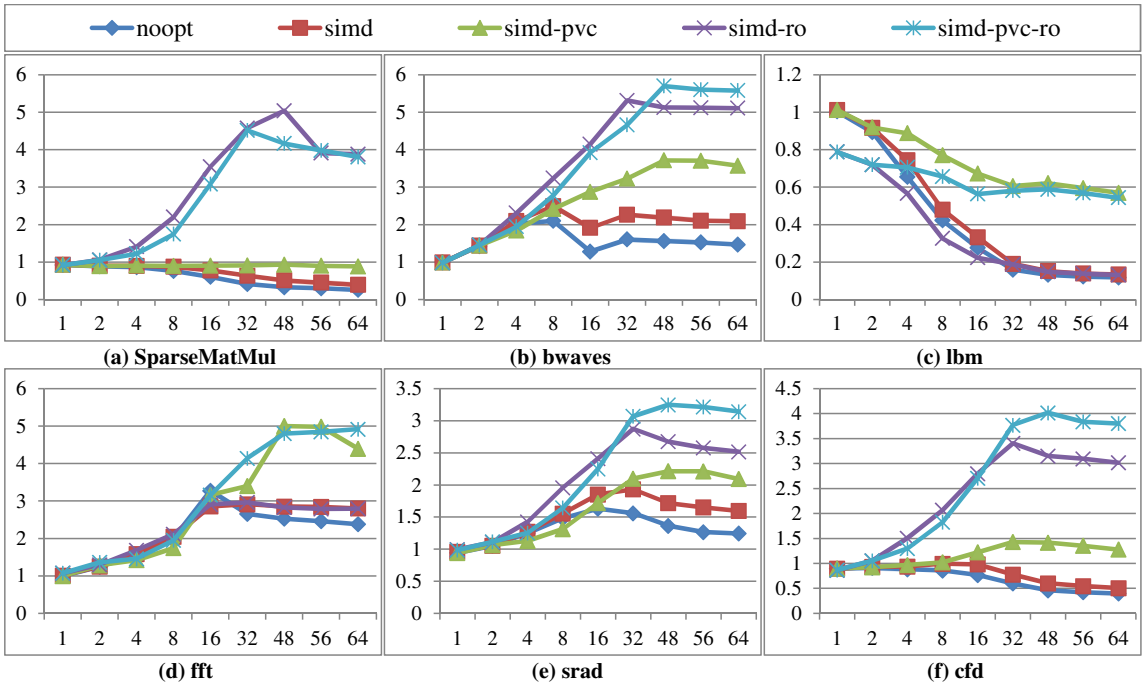


Figure 5–14: Speedups of Lazy Buffering; higher is better (2/3). The parallelized V/C and SIMD acceleration optimizations benefit memory-intensive benchmarks.

cfd benchmarks also show the interesting behaviour of the parallelized V/C optimization, which degrades the speedups with fewer cores but has benefits with more cores, demonstrating the trade-off between the speculative execution time and the validation/commit time.

As are exhibited in the geometric mean and most benchmarks, generally few or no dedicated parallelized V/C cores are needed on machines with no more than 8 cores. On average, the *simd-pvc-ro* version improves the geometric mean of the speedups by a factor of 1.59 from 4.06 to 6.46 at 64 cores.

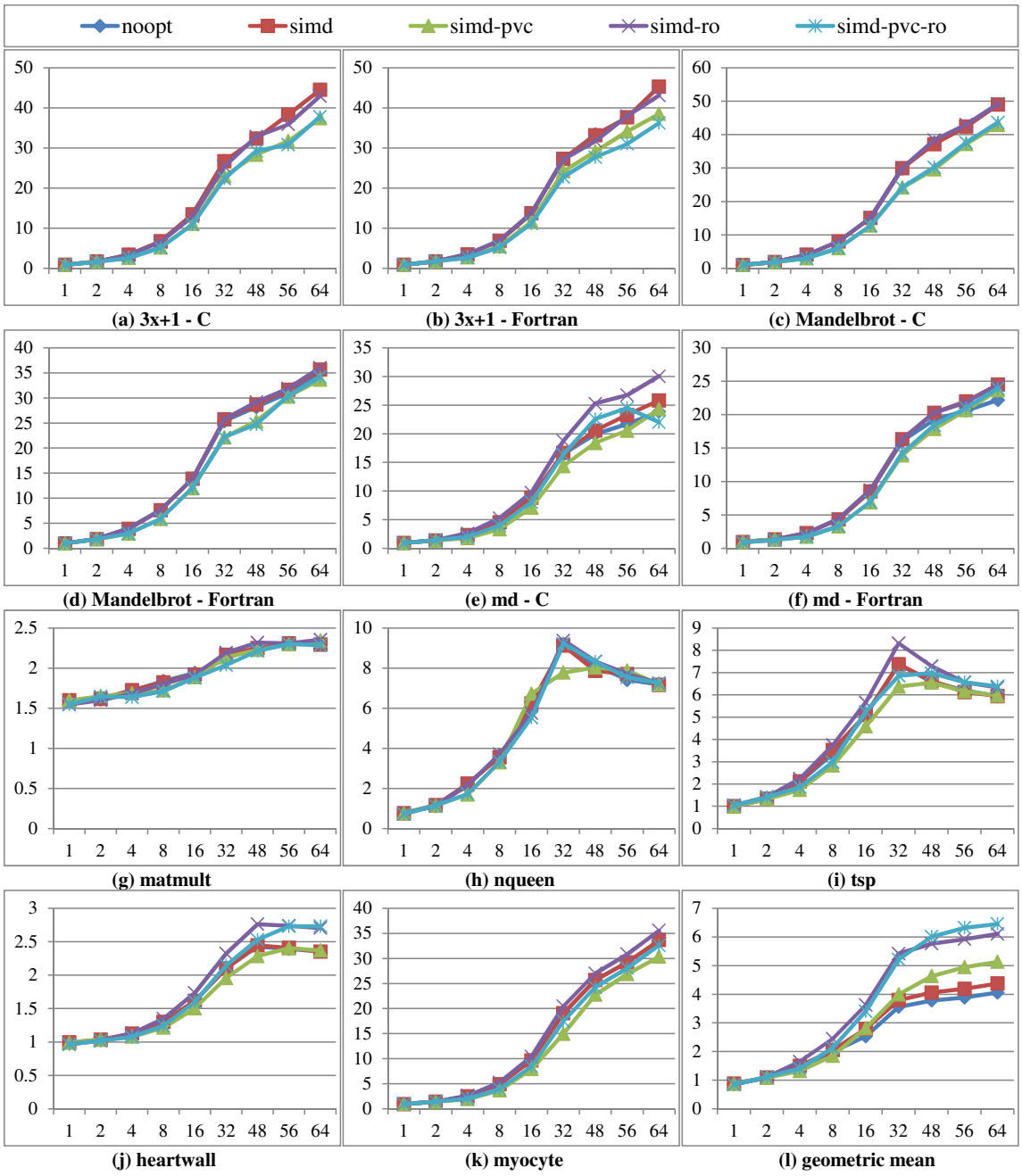


Figure 5–15: Speedups of Lazy Buffering; higher is better (3/3). The geometric mean performance is 59% faster with lazy buffering optimizations.

5.4.2 Speedup of Eager Address-Owner Buffering

We then experiment with the eager shared address-owner buffering optimizations. The results are shown in Figures 5–16, 5–17 and 5–18. The *eager-nolazy* version uses the eager buffering and rolls back threads with RAW, WAR or WAW dependencies. The *simd-eager* tries to use the eager buffering for independent global variables and falls back to the *simd* lazy buffering for dependent ones, and *simd-eager-ro* also enables the readonly-page optimization. We enable the thread stopping and buffer preserving optimizations for these *eager* versions and will experiment with the two optimizations in section 5.4.3. We also show the speedups of the *simd-pvc-ro* version for comparison with the optimized lazy buffering.

It can be seen that the readonly-page optimization also significantly benefits the eager buffering, demonstrating its effectiveness and low overhead. It significantly improves the speedups of bh, raytracing, smallpt, kmeans, srad and cfd, bwaves, streamcluster and sparsematmul, and moderately improves others. The only benchmark with slower performance is heartwall, which has independent variables that are written late in loop iterations and thus causes long wasted executions for the readonly-page optimization.

As expected, for most loop-based benchmarks such as bh, raytracing, smallpt, lbm, 3x+1, md, lavaMD, kmeans, cfd, heartwall, myocyte, mandelbrot and srad, the integrated buffering version *simd-eager-ro* demonstrates higher scalability and speedups with more cores than the optimized lazy buffering only version *simd-pvc-ro*, as a result of the higher scalability of the eager buffering, yet lower with few cores due to the eager buffering slowing down the non-speculative thread. In addition, other

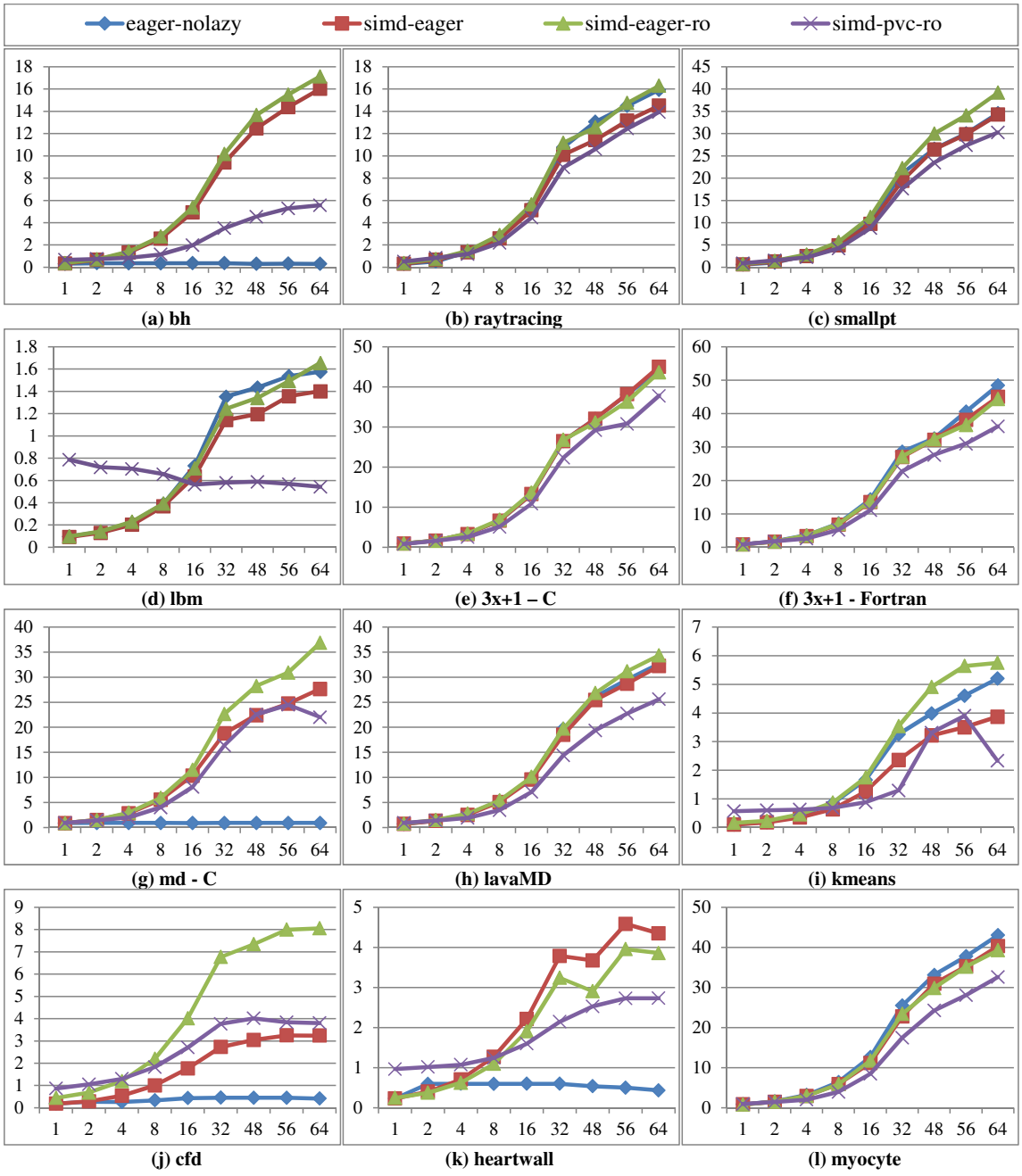


Figure 5–16: Speedups of Eager Buffering; higher is better (1/3). The eager buffering has higher scalability and speedups with more cores for most loop-based benchmarks.

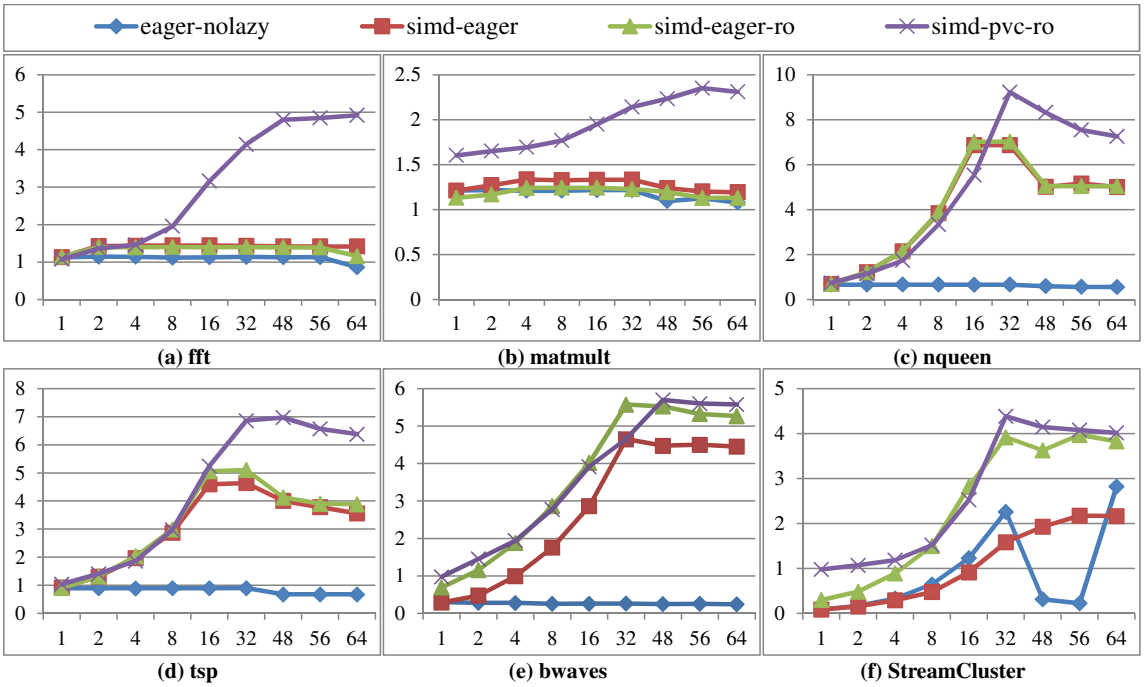


Figure 5–17: Speedups of Eager Buffering; higher is better (2/3). The eager buffering is less scalable and slower for tree-form recursion benchmarks due to the required use of in-order forking model.

than for cfd and srad, even without the readonly-page optimization, the integrated buffering implementation *simd-eager* is more scalable and faster with more threads than the *simd-pvc-ro* version which enables the readonly-page optimization.

On the other hand, the *eager* versions of the tree-form recursion benchmarks fft, matmult, nqueen and tsp have significantly lower speedups with more cores than the lazy buffering versions. This is because the eager buffering requires the use of in-order forking model, resulting in less parallel thread work coverage than the lazy buffering versions that utilize the mixed forking model. Overall, the integrated buffering version *simd-eager-ro* improves the geometric mean performance of the

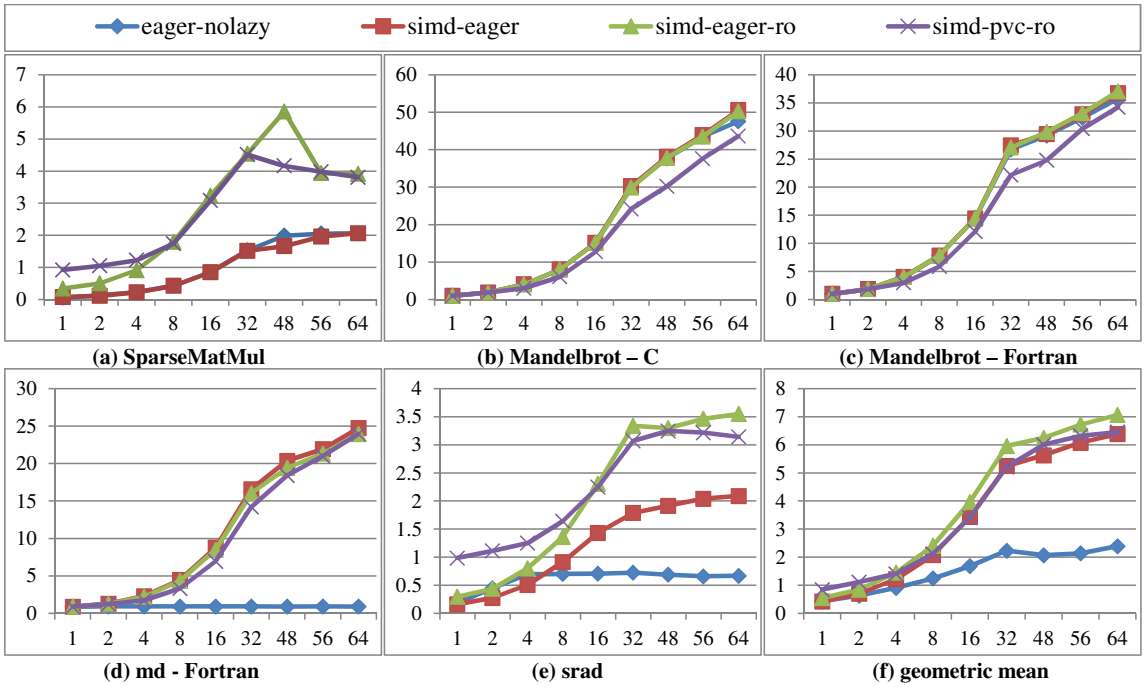


Figure 5–18: Speedups of Eager Buffering; higher is better (3/3). The geometric mean performance is 9% faster than the optimized lazy buffering.

optimized lazy buffering version *simd-pvc-ro* by 9% to 7.06, despite the slowdown of the tree-form recursion benchmarks due to required use of the in-order forking model.

The *eager-nolazy* version shows degraded performance for the bh, md, cfd, heartwall, nqueen, tsp, bwaves and srad benchmarks, due to unnecessary rollbacks. A common scenario is the access of privatized variables that incurs WAW dependencies, for example, temporary variables that are initialized at the beginning of each loop iteration. While for the bwaves benchmark, the rollback cause is memory access to the stack variables of the non-speculative thread. We cannot apply the eager buffering to the non-speculative stack, since speculative threads may access spilled

register variables and/or stack pointers, and thus misspeculation may corrupt the non-speculative stack. That the plunge of the *eager-nolazy* version of streamcluster at 48 and 56 cores is because the benchmark uses a boolean array while we track dependency of the eager buffering using 32-bit WORD, as was discussed in section 5.2, and thus causes rollbacks due to false dependencies. On the other hand, this benchmark demonstrates the advantage of accurate dependency tracking of the lazy page-table buffering.

5.4.3 Effectiveness of Buffering Optimizations

The performance results of the buffering integration mechanism are presented in Figures 5–19, 5–20 and 5–21. For better clarity and easier comparison, we scale the speedups to the best speedup version *simd-eager-ro*. Therefore, these versions have higher/lower speedups than *simd-eager-ro* if the speedup ratios are larger/less than 1, and higher/lower scalability if the curves go upward/downward with the number of cores. The *nostop* version does not enable the thread stopping optimization and uses the design of Figure 5–10 that rolls back all speculative thread tasks for restarting. The *nopreserve* version does not enable the buffer preserving optimization for the eager address-owner buffering. The *simd-eager-pvc-ro* version enables parallelized V/C for the lazy buffering of the *simd-eager-ro* version. The *heuristics* version enables the adaptive buffering selection heuristics for the *simd-eager-ro* version. We also show the *simd-ro* results for comparison with the corresponding optimized lazy buffering for the adaptive buffering selection heuristics.

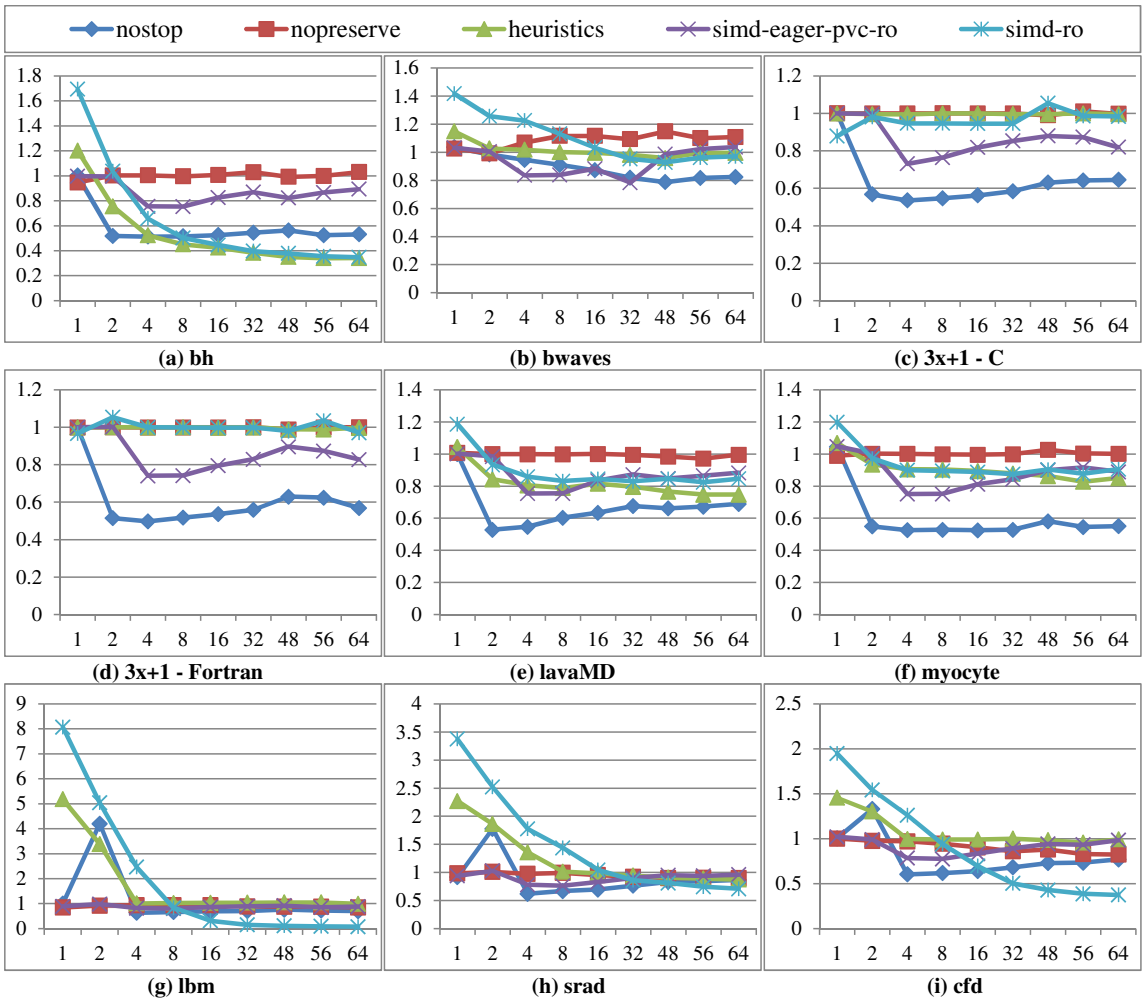


Figure 5–19: Effectiveness of Buffering Optimizations, scaled to the *simd-eager-ro* version; higher is better (1/3). The thread stopping optimization benefits loop-based benchmarks with shared readonly and independent variables.

The thread stopping optimization is effective for benchmarks with shared read-only and independent variables in loop speculative regions, and is especially beneficial for those with less iterations, for example, with the "blockize" transformation to statically distribute the loop iteration workloads. With more than two cores,

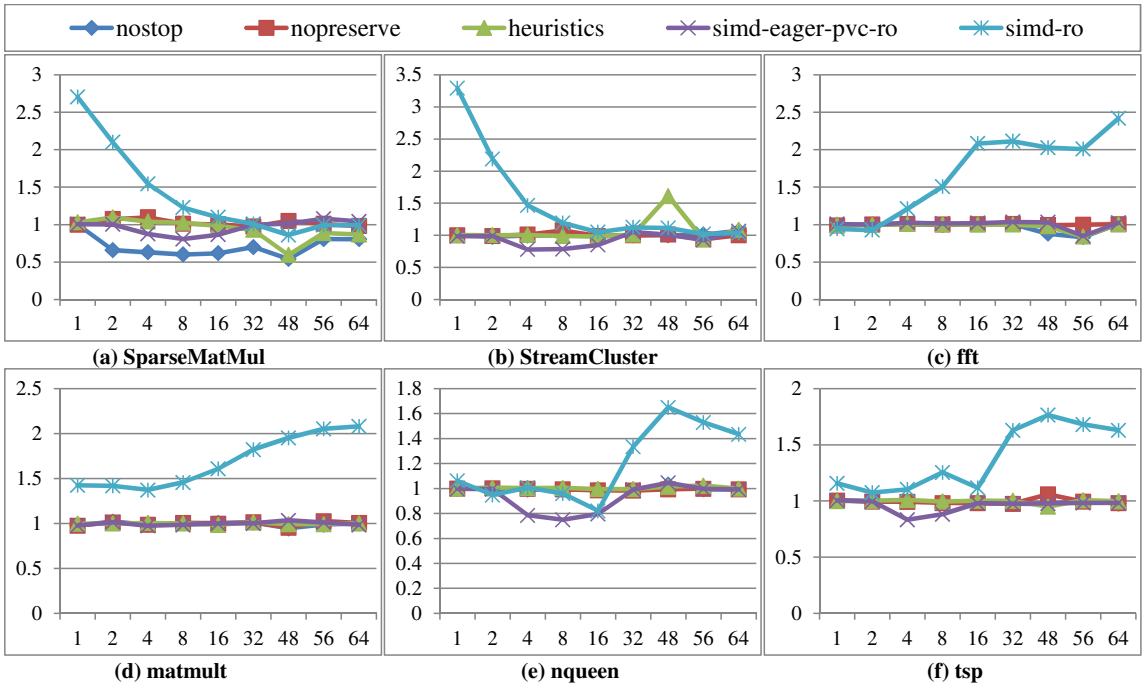


Figure 5–20: Effectiveness of Buffering Optimizations, scaled to the *simd-eager-ro* version; higher is better (2/3). The adaptive buffering selection heuristics cannot improve some benchmarks due to buffering integration overhead or mixed forking model.

the optimization significantly improves the speedups of bh, bwaves, 3x+1, lavaMD, myocyte, srad, cfd and sparsematmul, as a result of more parallel thread work coverage. That the md and heartwall benchmarks do not have noticeable speedups is because the page type preserving optimization discussed in section 5.3 significantly reduces the number of threads that are needed to rollback. On the other hand, using two cores the speedups of the lbm, srad and cfd benchmarks are significantly higher without the thread stopping optimization, as a result of the sequential region optimization of section 5.3, for which the non-speculative thread directly accesses the main memory without the eager buffering.

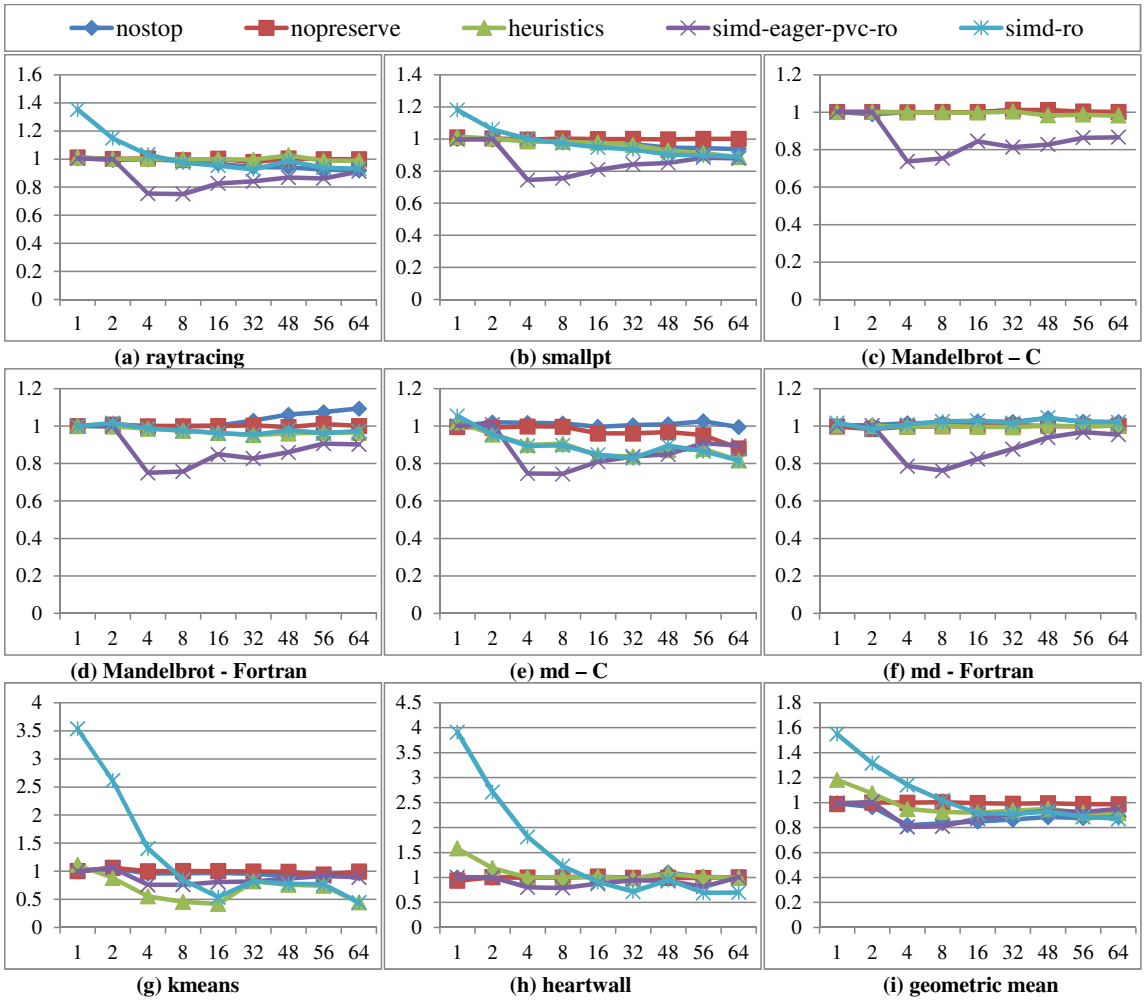


Figure 5–21: Effectiveness of Buffering Optimizations, scaled to the *simd-eager-ro* version; higher is better (3/3). The parallelized V/C optimization is not beneficial for most benchmarks with eager buffering.

The buffer preserving optimization improves the speedups of lbm, srad and cfd, but degrades bwaves due to the presence of data structures mixing shared and independent variables. From the geometric mean and most benchmarks, we can also see that the lazy buffering parallelized V/C optimization is generally not beneficial

for the integrated eager and lazy buffering implementations as a result of the higher scalability of the eager buffering.

The adaptive buffering selection heuristics helps to select the appropriate buffering. For memory intensive benchmarks such as lbm, srad, cfd and heartwall, the heuristics select the lazy buffering for fewer cores for its lower overhead on the non-speculative thread, and the eager buffering for more cores to benefit from its higher scalability, resulting in optimal solutions for different environments. However, for some benchmarks such as lavaMD, myocyte and kmeans, if eager buffering is selected, the heuristics may degrade the speedups due to extra rollbacks. We also see that there are scenarios where the buffering integration overhead could not be reduced by the heuristics, possibly due to various factors such as i-cache miss, branch prediction and more control flow/data access disabling compiler optimization. For example, while the heuristics select the lazy buffering for sparsematmul and streamcluster, the performance is still similar to that of the *simd-eager-ro* version. The heuristics is also unable to improve the tree-form recursion benchmarks since it assumes the potential presence of the eager buffering and uses the in-order forking model.

5.4.4 Speedup Comparison

The speedups of the benchmarks speculatively parallelized with the *simd-eager-ro* version that shows the highest geometric mean speedups are compared in Figures 5–22 and 5–23.

Here all benchmarks achieve speedups at 64 cores, and all linear speedup benchmarks including the bh and raytracing have high instead of mediocre speedups. Also

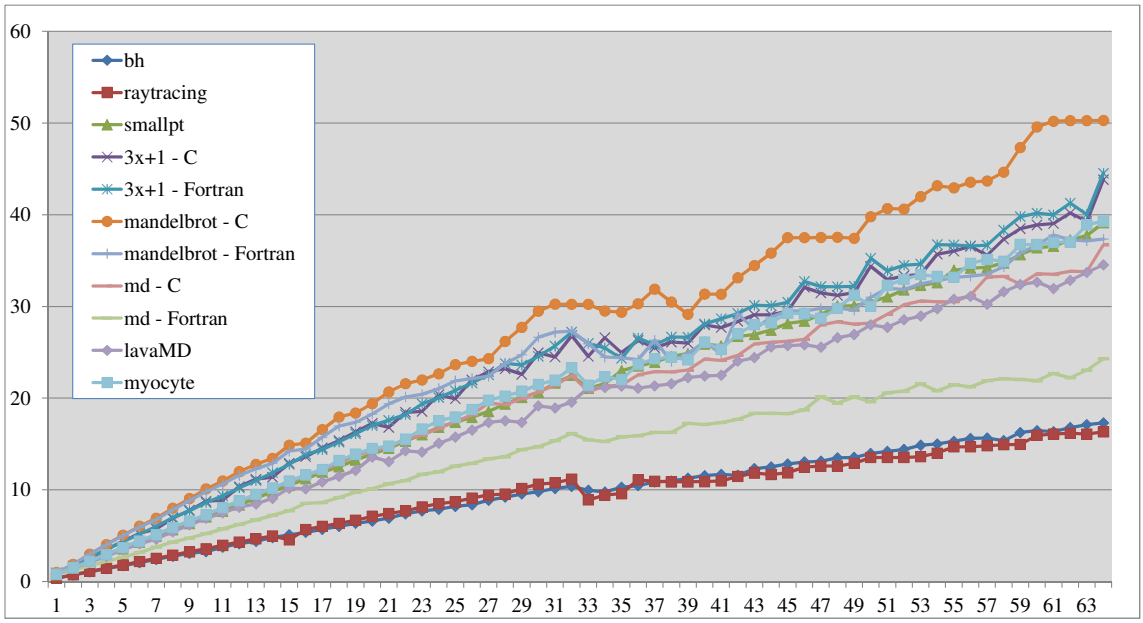


Figure 5–22: Eager Buffering Speedup Comparison - High Speedup

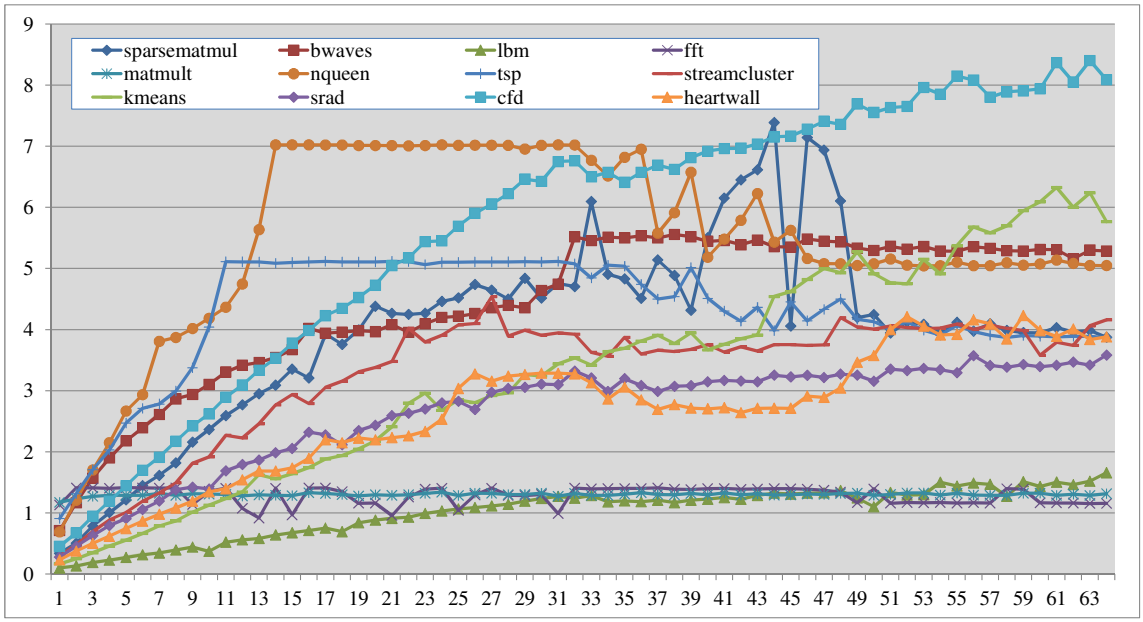


Figure 5–23: Eager Buffering Speedup Comparison - Mediocre Speedup

notable is that the speedups of nqueen and tsp are generally stable from 14 and 11 cores, respectively, until 32 cores, after which the performance start to degrade due to extra inter-socket messages on the non-uniform memory access (NUMA) [16] architecture machine.

The speedups of the benchmarks speculatively parallelized using the optimized lazy buffering version *simd-pvc-ro* are compared in Figures 5–24 and 5–25. For easier comparison of the buffering, we categorize the benchmarks the same way as in Figures 5–22 and 5–23.

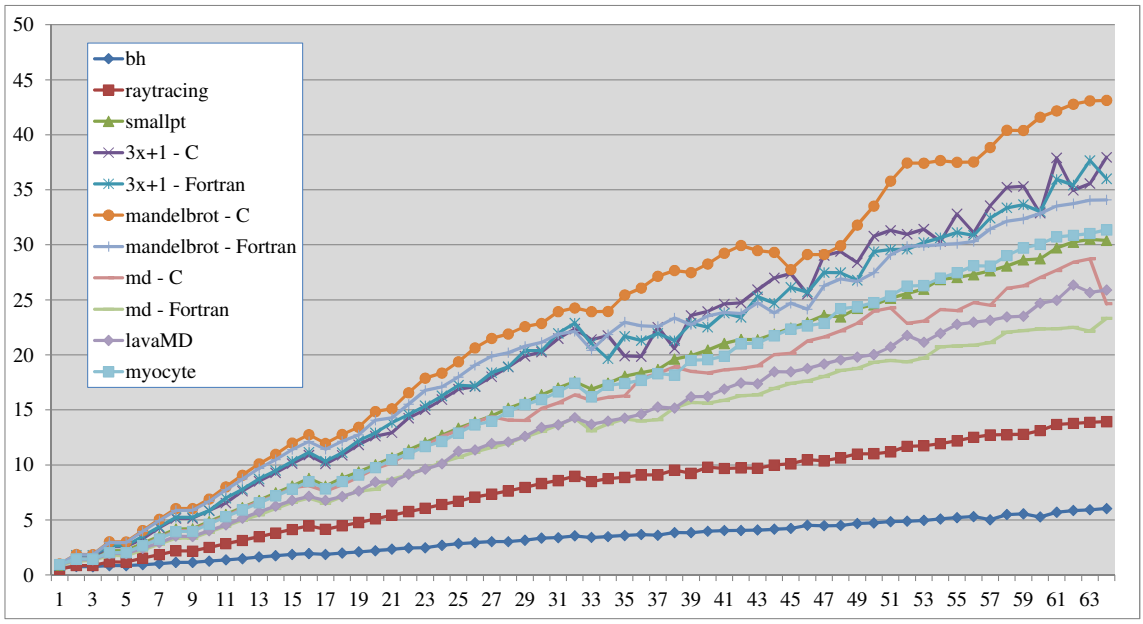


Figure 5–24: Lazy Buffering Speedup Comparison - High Speedup

It can be seen that the lbm benchmark still has a slowdown trend in the *simd-pvc-ro* lazy buffering implementation, though all other benchmarks achieve more than 2 times speedups at 64 cores. We can also see that the bh benchmark has significantly

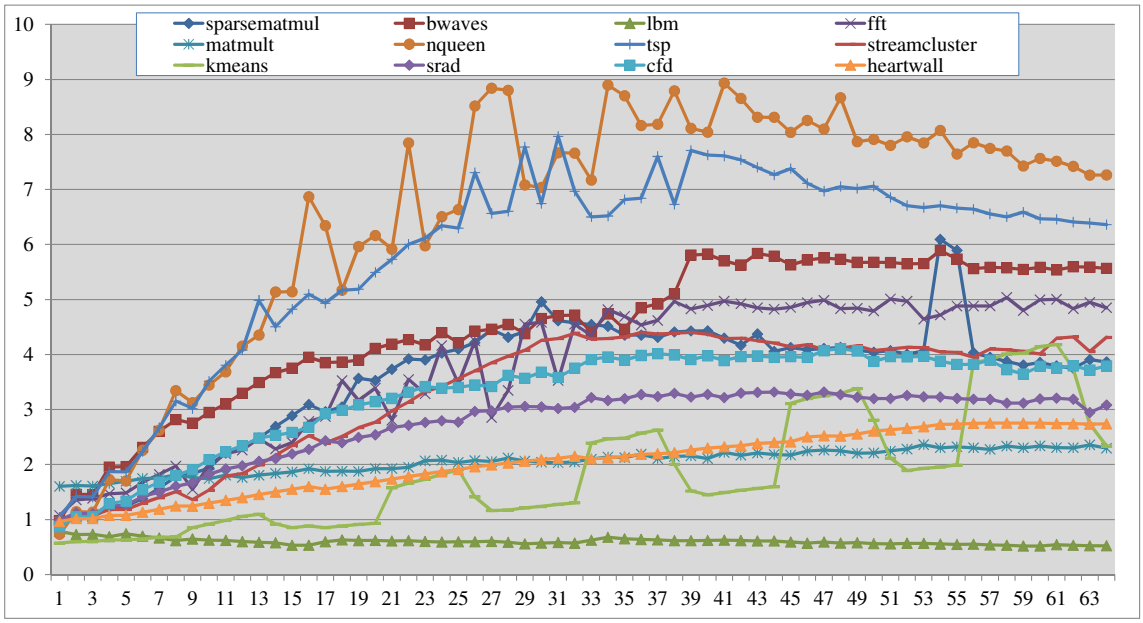


Figure 5–25: Lazy Buffering Speedup Comparison - Mediocre Speedup/Slowdown

less scalability than the eager buffering implementation, partly due to the presence of data structures mixing shared and independent variables. Another interesting benchmark is kmeans, which has a speedup trend but a periodically fluctuating performance that results from workload imbalance.

5.4.5 Theoretical Ideal Performance

The runtime ratios of the benchmarks speculatively parallelized using the MUTLS best speedup *simd-eager-ro* eager buffering to those manually parallelized using OpenMP are presented in Figures 5–26 to 5–29.

We can see that for all benchmarks except heartwall, the MUTLS *simd-eager-ro* version has similar or better scalability than the OpenMP version from 4 cores, which is encouraging. It can also be seen that for most benchmarks, the runtime of the

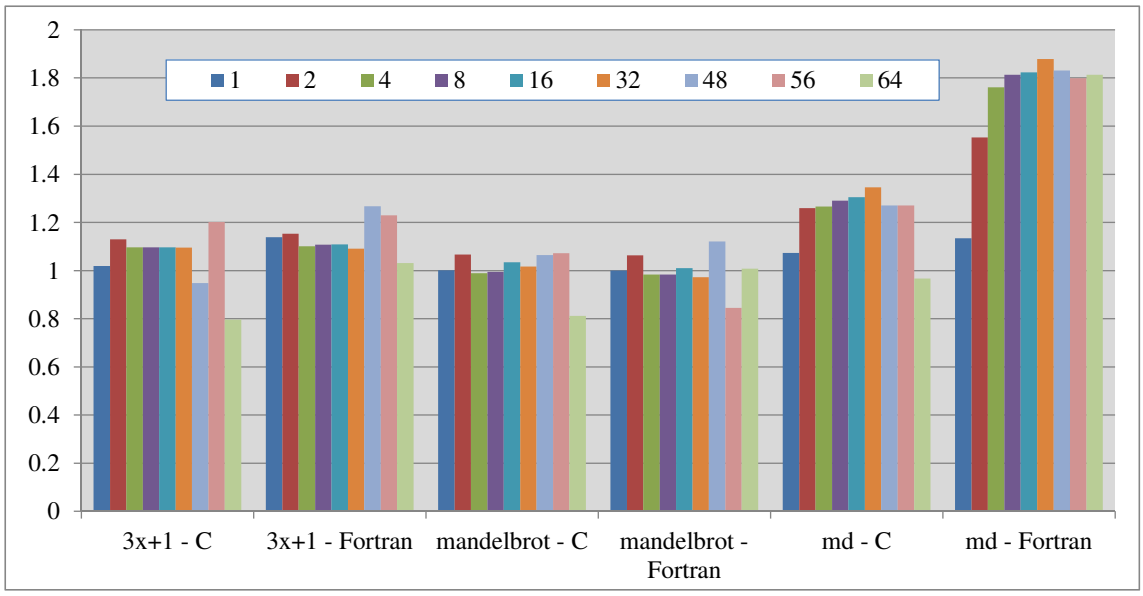


Figure 5–26: Eager Buffering MUTLS/OpenMP Runtime ratio; higher is worse (1/4)

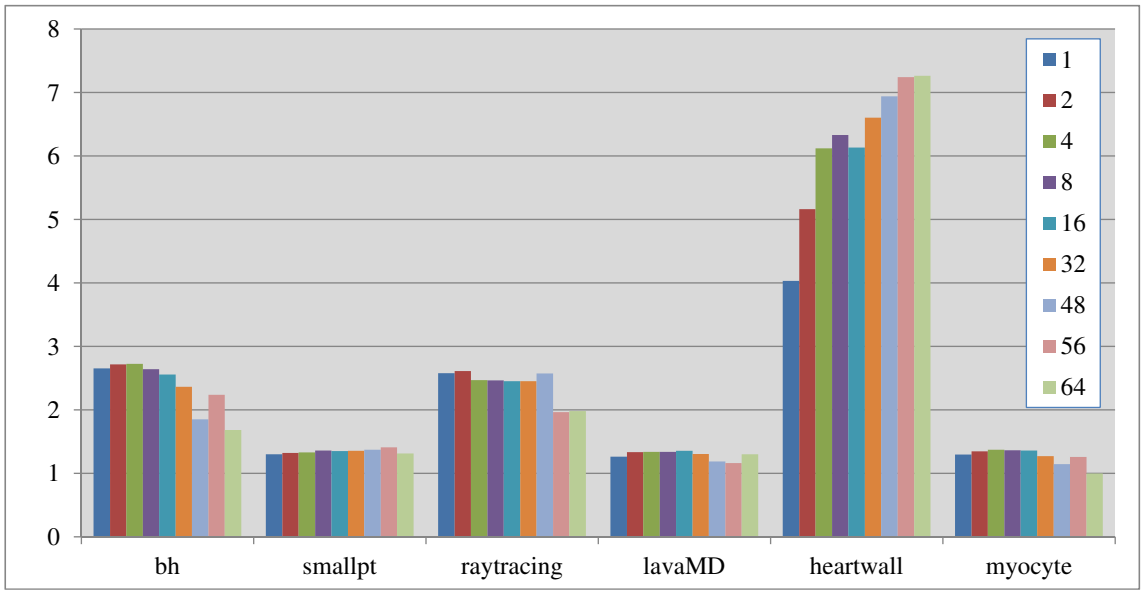


Figure 5–27: Eager Buffering MUTLS/OpenMP Runtime ratio; higher is worse (2/4)

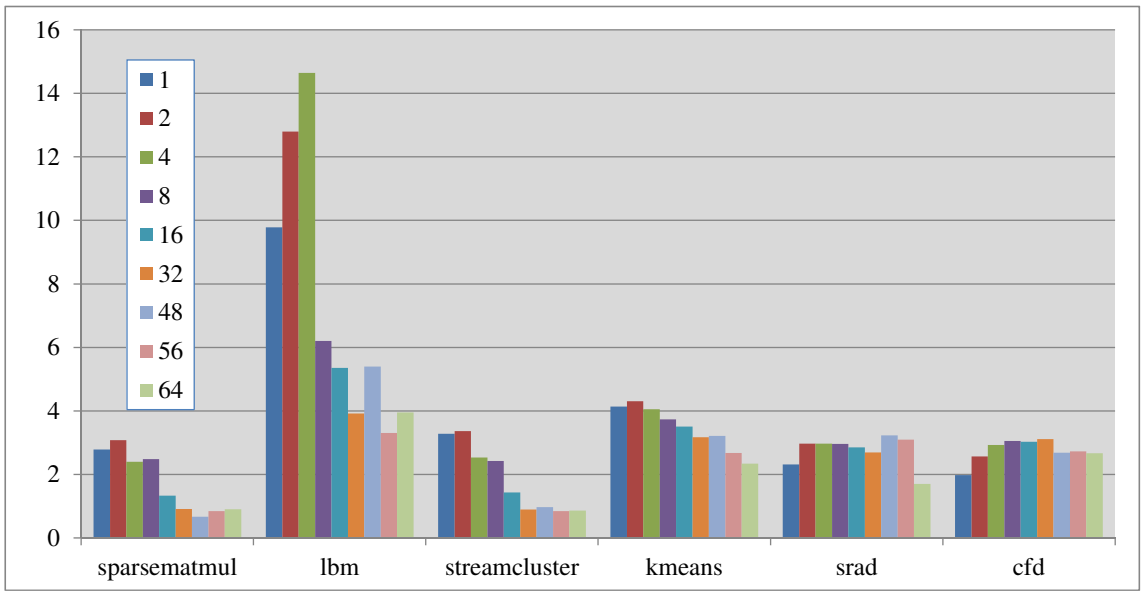


Figure 5–28: Eager Buffering MUTLS/OpenMP Runtime ratio; higher is worse (3/4)

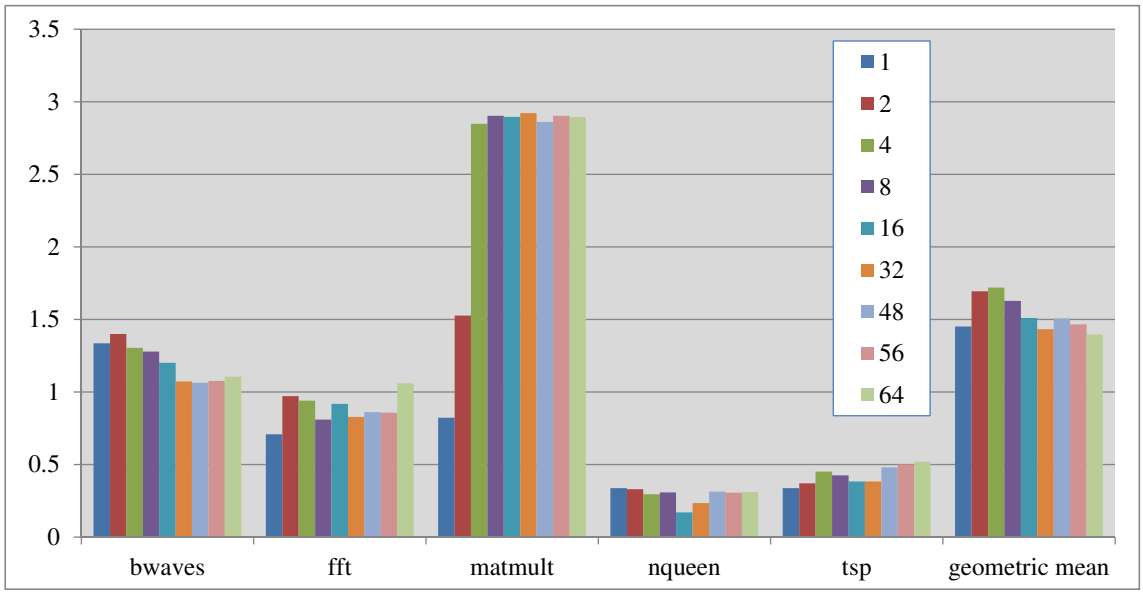


Figure 5–29: Eager Buffering MUTLS/OpenMP Runtime ratio; higher is worse (4/4)

MUTLS version is less than twice that of the OpenMP version at 64 cores; the only exceptions are heartwall (7.26), lbm (3.95), kmeans (2.34), cfd (2.67) and matmult (2.89). The geometric mean of all the benchmarks improves from the non-optimized buffering implementation of 2.43 slowdown (41% performance) in section 4.3 by 74% to 1.40 slowdown (71% performance).

We consider the performance overhead is reasonable, as parallelizing with MUTLS just needs to specify the fork point annotation and guarantees correctness, while OpenMP parallelization requires annotation of private/shared variables and does not guarantee safety.

5.4.6 Analysis of Parallel Execution

The critical path parallel execution efficiency η_{crit} and the power efficiency η_{power} of the *simd-eager-ro* eager buffering implementation are shown in Figures 5–32 and 5–33, respectively.

As expected, the eager buffering significantly improves the critical path efficiency: for all benchmarks at 64 cores, $\eta_{crit} > 0.75$, and the geometric mean is improved from 0.48 of the non-optimized buffering version in section 4.4 to 0.90. The scalability of the power efficiency is also improved significantly as a result of higher speedup scalability. The three benchmarks with the lowest power efficiency at 64 cores are lbm (0.030), sparsematmul (0.042) and streamcluster (0.083), as opposed to lbm (0.0035), streamcluster (0.0042) and sparsematmult (0.0077) of the non-optimized buffering version. However, we can also see that the power efficiency is lower at few cores than the non-optimized lazy buffering implementation due to the eager buffering overhead on the non-speculative thread. The geometric mean of

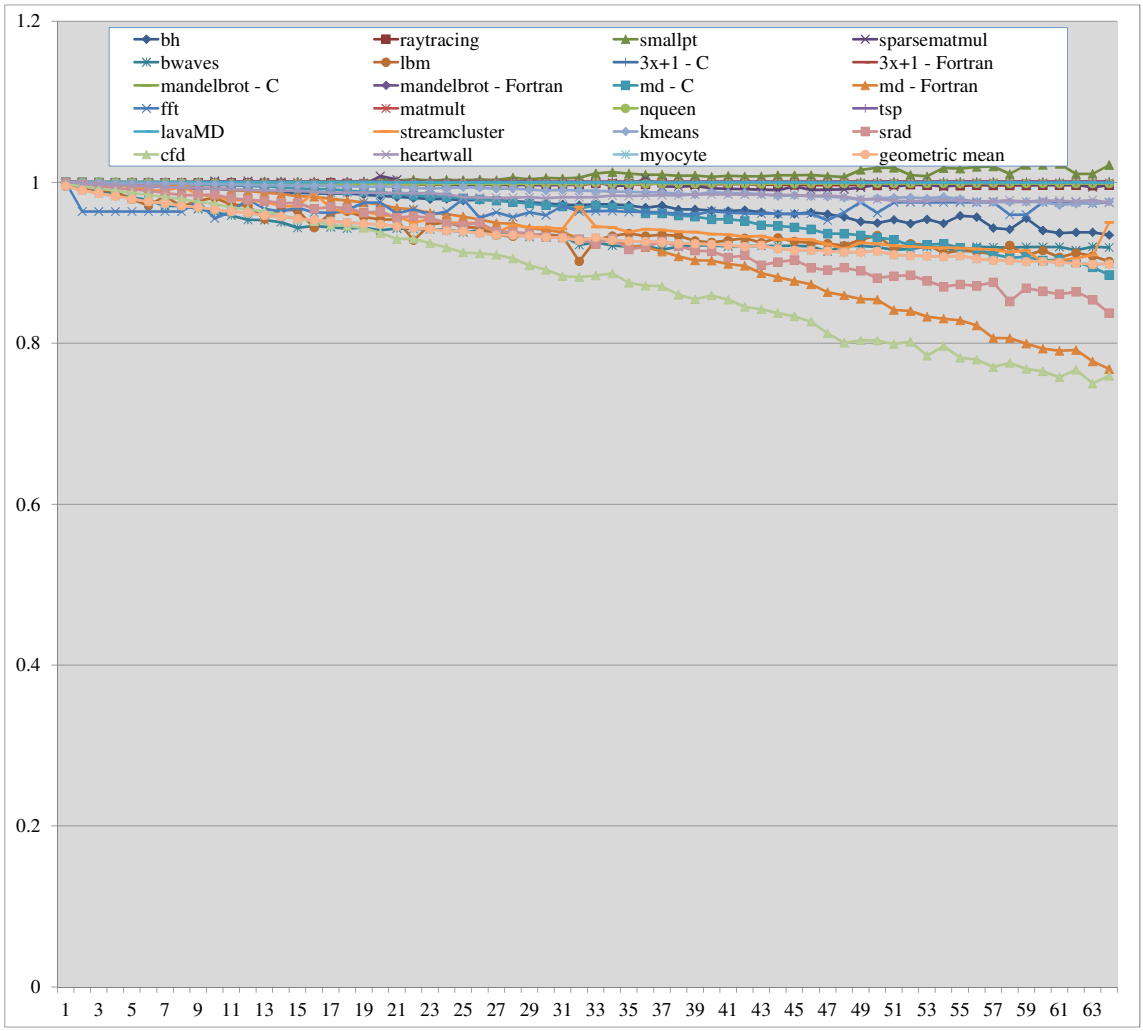


Figure 5–30: Eager Buffering Critical Path Execution Efficiency; higher is better

the power efficiency is improved from 0.16 to 0.34 at 64 cores with respect to the non-optimized buffering version, using less than half the energy for the speculative parallel programs, but degraded from 0.88 to 0.55 at 1 core.

The critical path efficiency and the power efficiency of the *simd-pvc-ro* lazy buffering implementation are shown in Figures 5–32 and 5–33, respectively.

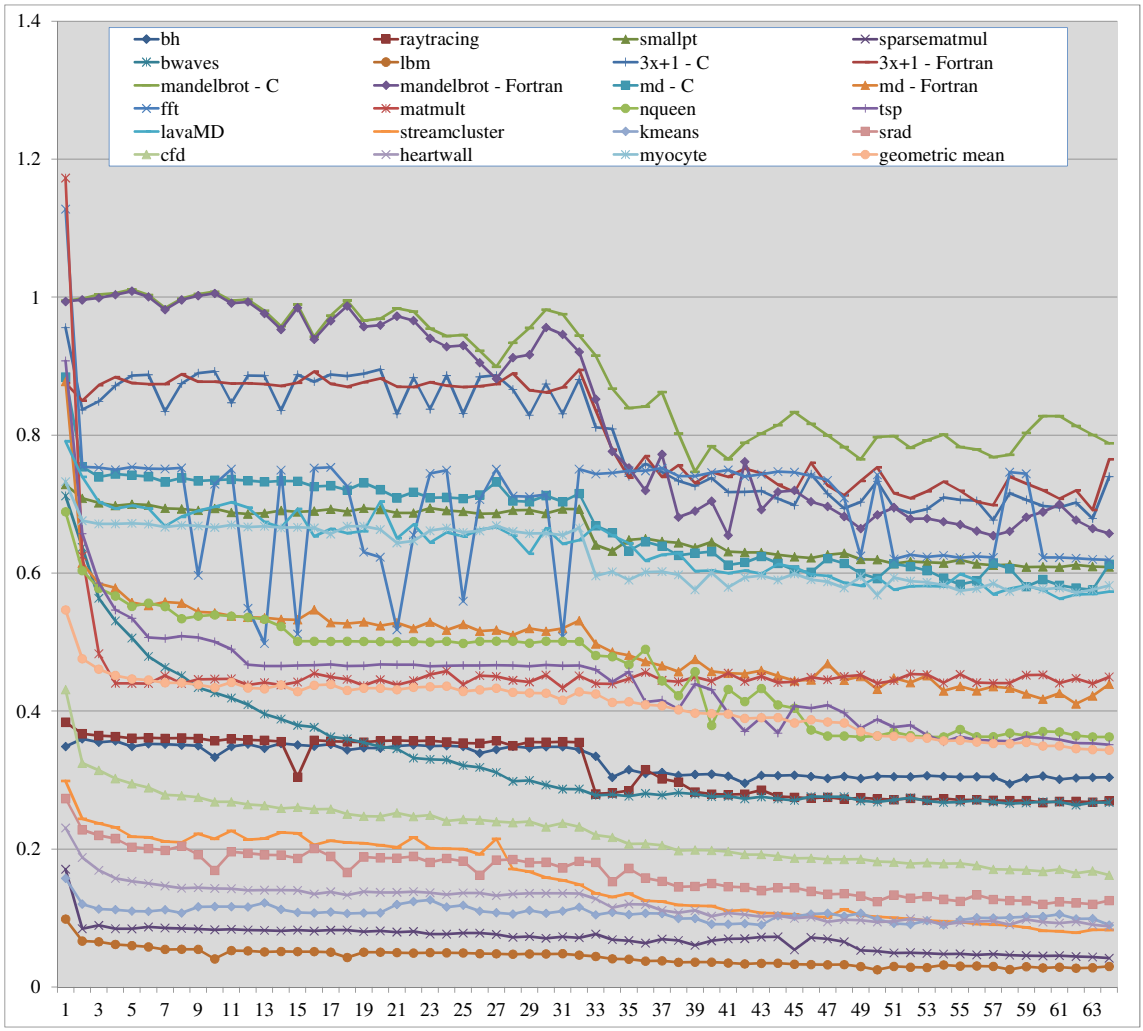


Figure 5–31: Eager Buffering Power Efficiency; higher is better

We can see that the optimized lazy buffering also improves the critical path efficiency and the power efficiency to 0.75 and 0.26, respectively, although the improvement is not as much as the eager buffering. On the other hand, the power efficiency of the optimized lazy buffering is higher than that of the eager buffering and the non-optimized lazy buffering at few cores, which is as expected since it does

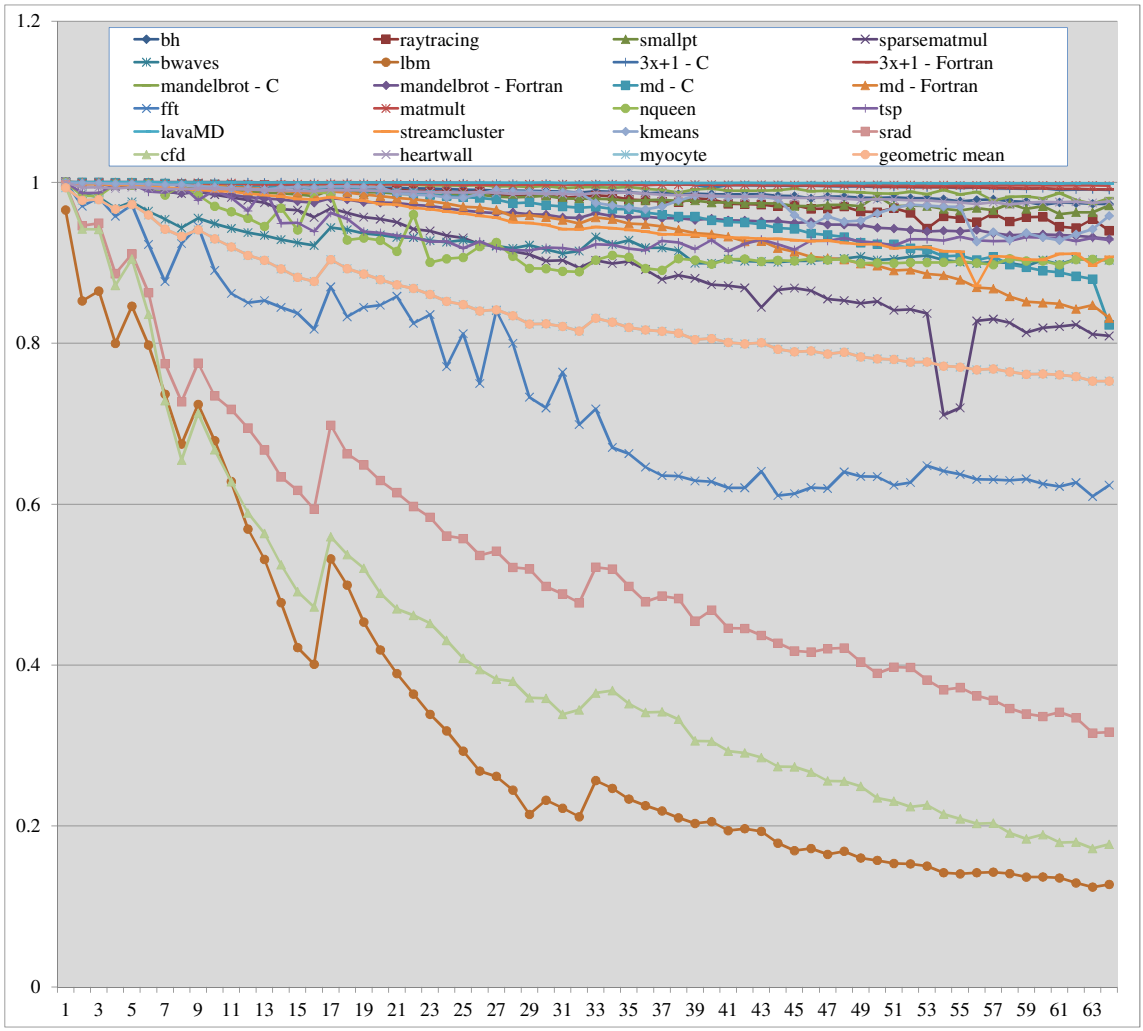


Figure 5–32: Lazy Buffering Critical Path Execution Efficiency; higher is better

not incur overhead on the non-speculative thread while optimizing the speculative threads for the non-optimized lazy buffering. The three benchmarks with the lowest critical path efficiency are lbm (0.12), cfd (0.18) and srad (0.32), improved from lbm

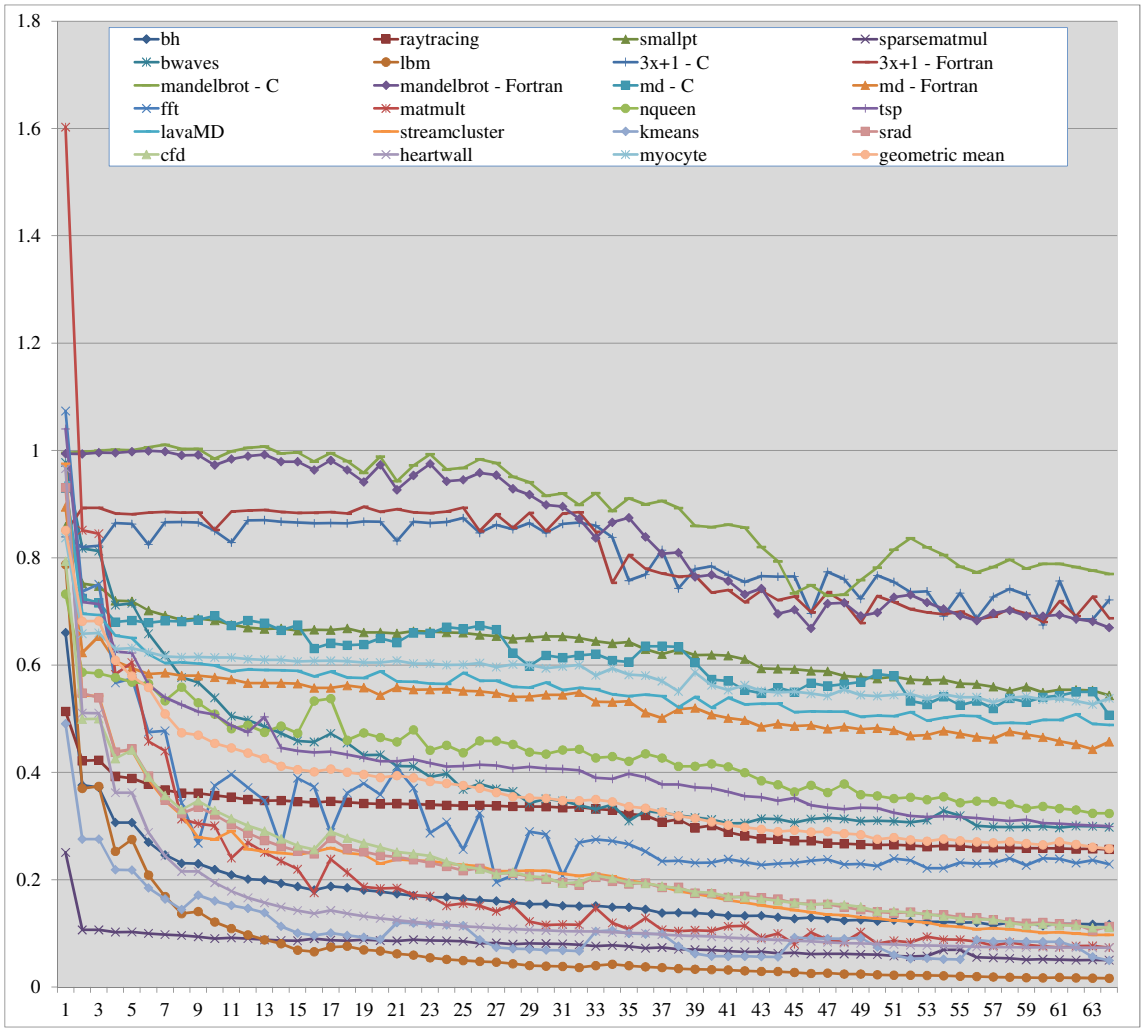


Figure 5–33: Lazy Buffering Power Efficiency; higher is better

(0.017), streamcluster (0.021) and cfd (0.030) of the non-optimized buffering implementation. The three benchmarks with lowest power efficiency at 64 cores are lbm (0.016), kmeans (0.049) and sparsematmul (0.050).

5.5 Chapter Summary

In this chapter, we proposed several optimizations to reduce memory buffering overhead for software-TLS. First we described the lazy per-thread page-table buffering that enables both coarse and fine grain parallelism for validation/commit, as well as the parallelized V/C and SIMD acceleration optimizations to exploit the parallelism. Then we presented the eager shared address-owner buffering whose space overhead is bounded by a constant factor of the program data size to enable speculation at any granularity without causing buffering overflow. The address-owner eager buffering design also enables the buffer preserving optimization to reduce the thread clearing/re-initialization overhead. Afterwards, we proposed the readonly-page optimization, buffering integration mechanism and adaptive buffering selection heuristics to automatically identify readonly, independent and dependent variables on-the-fly and apply the suitable optimization/buffering for each variable. We also proposed the thread stopping optimization to improve thread coverage for the readonly-page optimization and buffering integration mechanism. Experimental results show that the proposed memory buffering optimizations are helpful to achieving efficient software-TLS: (1) both the optimized lazy and eager buffering achieve significantly higher scalability, speedups and critical path and power efficiency than the non-optimized buffering; (2) the readonly-page optimization is one of the most effective memory buffering optimizations for benchmarks with large readonly variables, and the buffering integration mechanism can effectively combine the strengths of the eager and lazy buffering; (3) the SIMD acceleration, buffering preserving and thread stopping optimizations are generally beneficial and should be enabled for both eager and lazy

buffering, while the parallelized V/C optimization should be applied for the lazy buffering but is generally not beneficial if eager buffering is enabled. In all, the best speedup *simd-eager-ro* version achieves speedups of 24.3 to 50.3 and 1.2 to 17.3 for computation and memory intensive benchmarks, respectively, and on average 71% performance of the OpenMP manually parallelized version.

CHAPTER 6
Fork Heuristics

TLS can be applied to enable automatic parallelization, as it guarantees the safety and correctness of speculative parallel program execution dynamically at runtime. However, the selection of fork/join points plays a key role in the performance of TLS, especially for software implementations as a result of higher overhead than its hardware counterpart. Fork heuristics have been proposed to enable the effective selection of fork/join points for automatic parallelization using TLS.

So far there are three sorts of fork-heuristics: static heuristics [61], static profiling heuristics [51, 140, 162] and dynamic profiling heuristics [103]. The first build mathematical cost-benefit models of speculative execution using compile-time program information, and use the models to predict profitable fork/join points. This approach has the limitation that some model parameters, such as thread dependency probability and iteration count of nested loops, are unknown at compile-time, which in turn limits its effectiveness and application. The second heuristics compile and run the sequential program, collect profiling execution traces, and then use the traces to determine the best fork/join points. The drawback of this approach is lack of real parallel execution environment information, which limits accuracy of the fork point selection decision. The third approach is more promising for real estimation,

but is currently based on hardware implementation, which is inappropriate and cannot be directly applied to software TLS without accessing architecture-dependent performance counters.

6.1 Contribution

In this chapter we propose adaptive fork-heuristics to solve the above problems. Adaptive heuristics are dynamic profiling heuristics for software TLS, which insert all potential fork/join points into the speculative program and rely entirely on the runtime system to determine profitable fork/join points and disable inappropriate ones. Since fork/join points are evaluated during real speculative parallel execution, all necessary information such as the thread conflict ratio and thread execution time is available, enabling accurate estimation of cost-benefit of each thread and thus each pair of fork/join points. On-the-fly fork/join point selection also eliminates the requirement of profiling runs and enables adaptation to different fork/join points for different input data. Our investigation demonstrates feasibility of this approach, as well as providing concrete data on actual performance in a realistic thread-level speculative system.

6.2 Adaptive Fork-Heuristics

Adaptive fork-heuristics add potential pairs of fork/join points to the speculative program, evaluate the cost-benefit of each pair during parallel execution and disable unprofitable ones. The design involves three aspects: (1) what the potential fork/join points are, (2) how to estimate the cost-benefit of each pair of fork/join points, and (3) how to disable fork points.

6.2.1 Potential Fork/Join Points

Since loops usually take the majority of program's execution time and function calls usually represent independent computation tasks, the loop-level speculation and method-level speculation speculate on loop iterations and function (method) calls, respectively, as was discussed in section 2.2. We can also combine the two choices to take advantage of both. Arbitrary point hardware-TLS systems such as Mitosis [140] consider any pairs of basic blocks as candidate fork/join points irrespective of the control flow. For efficient implementation and simpler evaluation of the adaptive fork heuristics, we choose to speculate on loop-iterations. Other design choices of potential fork/join points are planned as future work.

We also apply two optimizations for each loop: "blockize" and "end-barrier." Suppose there are n processors. Blockize splits the loop iterations into n blocks and speculates on the loop blocks, which in turn avoids creating too many small threads. The exception is loops with a small constant number of iterations, which do not need this optimization. The end-barrier optimization adds a barrierpoint just after the end of the loop. This optimization is beneficial because loops usually have dependency with their continuation, particularly for loop nests, in which case an inner loop thread may cause cascading rollbacks of the outer loop threads.

6.2.2 Cost-Benefit Estimation

The design uses a cost-benefit model to evaluate the profitability of each thread and each pair of fork/join points. The model assumes a constant time $T_{overhead}$ of overhead (thread creation, cache miss, buffering, etc) for each thread. Although this

is an inaccurate approximation since threads with different memory access frequencies have different buffering overhead, we find it works well for our estimation, partly because we are only concerned with whether a thread is profitable, and not how profitable it is.

The runtime $T_{t,run}$ of a speculative thread t comprises two parts: work time $T_{t,work}$ and synchronization/validation/commit/rollback time, which are available through timing. If thread t commits, its cost-benefit is estimated as $\eta_t = T_{t,work}/(T_{t,run} + T_{overhead})$. If it rolls back, its cost-benefit is 0. Given a minimum cost-benefit threshold $\eta_{threshold}$, if $\eta_t < \eta_{threshold}$, then thread t is considered not profitable and should not have been speculated.

If the assumption holds that threads speculated at the same fork point always show similar behaviour (they always commit/rollback and have similar work time / runtime ratio), then we can directly use η_t to estimate the cost-benefit of the fork point. However, the assumption generally does not hold, as the characteristics of future program execution may differ from history. Also, for speculative regions with rare dependencies, speculation may cause nondeterministic rollbacks even though the fork/join points should be selected.

We propose 3 independent mechanisms to address this issue: global hint, local hint and interval hint. Global hint uses at least N_{warmup} threads instead of one thread to determine the cost-benefit of a pair of fork/join points. When a thread completes execution, its runtime plus overhead is accumulated to the runtime T_{run} of the pair $T_{run} = T_{run} + T_{t,run} + T_{overhead}$. The exception is when it is cascadingly rolled back or stopped by the thread stop optimization as was discussed in section 5.3.1,

as these cases do not represent the cost-benefit of a thread. If it commits, its work time is accumulated to the work time T_{work} of the pair $T_{work} = T_{work} + T_{t,work}$. After $N >= N_{warmup}$ threads completes, the cost-benefit of the pair is then estimated as $\eta = T_{work}/T_{run}$. The hint disables the fork point if the cost-benefit is below a threshold. We use a default threshold 0.5 to indicate overhead should not take more time than useful work. The threshold value may be set differently or dynamically for specific benchmark characteristics.

For local hint, if a thread decides not to speculate on a fork point then none of its child threads, grand-child threads, etc can speculate on the fork point. In other words, a local hint affects the sub-tree of a thread, hence the name. Interval hints directly use the cost-benefit of a thread to decide profitability of its fork point; if a fork-point is disabled, it will try to speculate again after certain amount of time has passed. We find the global hint is the most effective for our benchmarks. It seems to work well on independent loops while the other two might suit more irregular applications. We plan to compare these hints in future work.

6.2.3 Disabling Fork Points

Each fork point has a globally unique id. The TLS runtime system maintains the attributes of the fork point, which can be accessed given the id. When a thread reaches a fork point, it calls the `MUTLS_get_CPU` library function as was discussed in section 3.5.2, which queries the runtime system with the id whether it can speculate on the fork point. The runtime system then checks a flag variable of the fork point attributes and returns the result. When a thread commits/rollbacks, if the adaptive

fork-heuristics decide that one fork point is not profitable as was discussed in subsection 6.2.2, the runtime system then sets the flag variable to false to disable the fork point.

If a loop nest has independent outer loops, such as enumerating elements on a matrix, then we can select to speculate on any or all of these loops. Speculating on outer loops enables coarser granularity parallelism but tends to consume more memory than inner ones, while speculating on all loops maximizes parallelism. These decisions have important influence on performance. As disabling inner ones usually yields further speedups as a result of less thread overhead, we apply the "nest-loop-disabling" optimization to disable an inner loop if its parent nest loop is selected.

If a thread is waiting to be joined, it should be counted as work time only if the thread is waiting to join its child thread, since the current thread should have continued working if it had not speculated the child thread. If the thread is blocked at a terminate point or barrier point, the time should just be counted as thread overhead. We add a point type parameter to the update time function, and check if the speculative thread is waiting at a join point. We need not check the join point id, since a speculative thread can only wait for synchronization at a join point with the same fork point id as it was speculated.

We also propose an optimizing technique called *feedback-based selection* to achieve ideal speedups from the second compilation for our benchmarks. After the program completes execution, it records the cost-benefit of each fork point to a feedback-based selection log file. The next time the TLS compiler compiles the program, it reads the log file and does not insert inappropriate fork/join points as potential candidates.

For points that behave differently depending on the input, the programmer can annotate them so that the compiler still insert them even though they are in the log file. The optimization prevents unprofitable fork/join points from hurting performance repeatedly for each compilation.

6.3 Implementation Framework

We implement the adaptive fork-heuristics into the MUTLS software-TLS system. As was discussed in section 3.1, MUTLS supports compiler directives to annotate fork/join/barrier points. Each annotation also specifies an id. Threads speculated at a fork point will start execution from the join point with the same id, and will be joined when the non-speculative thread reaches that join point. A thread will also stop execution when it reaches a barrier point with the same id.

Since MUTLS fork/join/barrier point annotations can be inserted both manually by the programmer and automatically by the compiler or other tools, as discussed in Chapter 3, the fork heuristics can be implemented either semiautomatically with manually added fork heuristics annotations, or automatically with compiler fork heuristics annotations and related optimizations. Both have their application scenarios: the former can simplify manual parallelization and/or diagnosis of the parallelized program with the feedback-based selection log file, while the latter can automatically parallelize the program without manual parallelization effort.

A sample program as well as its semiautomatically parallelized program annotated with adaptive fork-heuristics and feedback-based selection log file generated by the MUTLS compiler is illustrated in Figure 6–1. Given the log file, there are various criteria to decide inappropriate fork/join points, such as whether a fork point

is disabled, whether the cost-benefit is below a threshold, and whether the ratio of committed/total threads is below a threshold. Combination of these criteria is also possible. In the current implementation, we simply decide not to add a pair of fork/join points as potential ones if the fork point is disabled. However, feedback-based selection can be iterated over each program compilation/run using different criteria and machine-learning approaches can be used to combine strengths of different runs to produce optimal versions, which we plan as future work.

```
void work(int n) {
    …
    for(i = 0; i < n; i++){
        for(j = 0; j < 8; j++){
            x = f(i, j);
            s[i][j] += x*x;
        }
    }
    …
}
```

(a) Original Program

```
fork point id 1 selected
cost-benefit 0.9 commit 80 rollback 0
fork point id 2 disabled
cost-benefit 0.3 commit 15 rollback 0
fork point id 3 disabled
cost-benefit 0 commit 0 rollback 10
```

(c) Feedback-based Selection Log File

```
void work(int n) {
    …
    for(p = 0; p < P; p++){
#pragma tls forkpoint loop id 1 maybe
        for(i = n * p / P; i < n * (p+1) / P; i++){
            for(j = 0; j < 8; j++){
#pragma tls forkpoint loop id 2 maybe
#pragma tls forkpoint id 3 maybe
                x = f(i, j);
#pragma tls joinpoint id 3
                s[i] += x*x;
            }
        }
    }
    …
}
```

(b) Program with Adaptive Fork-Heuristics

Figure 6–1: Semiautomatic-Parallelization of a Program

6.4 Automatic Parallelization

This section describes how we implement automatic parallelization in the MUTLS framework using adaptive fork heuristics. We present the LLVM transformations to

automatically insert the fork/join/barrier point annotation built-in functions and the implementation of related adaptive fork heuristics optimizations.

For each loop, we insert a fork point at the beginning of its header block, the corresponding join point at the end of its latch block, and a barrier point at the beginning of each exit block. An exit block is the destination block outside the loop that a block inside the loop branches to. We do not speculate on loops that have no latch blocks. For performance, for each loop nest, we do not speculate on the innermost loop unless it contains function calls, as such a loop usually does not have sufficient speculative work even with the "blockize" optimization as was discussed in section 6.2.1, while the speculative transformation for the loop fork/join/barrier points incurs significant overhead.

When speculating a child thread at a fork point, the parent thread needs to save register variables live at the join point to the runtime library as was discussed in section 3.6.4. Therefore, the program transformation should try to guarantee that register variables live at the join point also be live at the fork point; otherwise the child thread would have to apply value prediction for the register variables, which would cause thread rollback if the value was incorrectly predicted. Our approach is to hoist and sink non-live register variables. For each register variable live at the join point, we first check whether it is live at the fork point; if not, we try to hoist it above the fork point as a loop induction variable using LLVM ScalarEvolutionExpander; if not possible, we try to sink it below the join point; if still not possible, in the current implementation we consider the loop has data dependency on the register variable

and do not speculate on it. Speculation on loops with value prediction is planned for future work.

To implement the "blockize" optimization of section 6.2.1, for each loop, we add a block-header basic block and a block-latch basic block, each inserted with the fork and join point annotation built-in function, respectively. In the block-header block, we add the induction variable phi-node of the blockized loop iterating from 0 to $n-1$ and check the exit condition of the blockized loop in the block-latch block. For each phi-node of the original loop, we compute the values of the beginning and the end loop iterations of each blockized iteration using LLVM SCEVAddRecExpr in the block-header and assign the values to the phi-node. If the value of a phi-node of the original loop cannot be computed given the blockized iteration number (the phi-node is not SCEVable), we do not apply the "blockize" optimization to the loop.

The fork/join/barrier points for each loop should have the same unique point id within the function, and the same unique hint id globally, as was discussed in section 6.2.3. For this we maintain a point id counter for each function and a hint id counter for the module, and increment the counters if a fork/join/barrier point id group is added. To create the loop-nest structure for implementation of the "nest-loop-disabling" optimization of section 6.2.3, we assign the ids of each loop-nest in a pre-order traversal, and for each loop assign the loop id of its outer loop to be the parent loop id. At the beginning of the speculatively parallelized program execution, the loop id and parent loop id pairs are passed to the MUTLS runtime library to construct the loop-nest metadata structure.

6.5 Experimental Results

In this section we experiment with the adaptive fork-heuristics and the feedback-based selection. We use buffering version *simd-eager-ro* with the highest geometric mean speedups of section 5.4 for the experiments in this section. First we compare the speedups of the benchmarks automatically parallelized by the MUTLS system using the adaptive fork heuristics and then improved by the feedback-based selection. Then we show the number of enabled fork points during each iteration of compilation with the adaptive fork heuristics and feedback-based selection.

The speedup results of the automatically parallelized benchmarks are shown in Figures 6–2 to 6–5. The *heuristics* version is the automatically parallelized version using adaptive fork heuristics without the feedback-based selection. The *feedback-1* and *feedback-2* versions are automatically parallelized using adaptive fork heuristics with the feedback-based selection log file generated by the *heuristics* and the *feedback-1* versions, respectively. For each benchmark, we show two figures: the one with the "-b" suffix is applied the "blockize" optimization while the one with the "-n" suffix is not. For easier comparison, we also show the manually annotated version *simd-eager-ro* of section 5.4, which is the higher speedup version of the ones with and without the "blockize" optimization. Therefore, both figures of the manually annotated *simd-eager-ro* version are either blockized or not irrespective of the "-b" and "-n" suffixes: all benchmarks except raytracing, smallpt, bwaves and mandelbrot are applied the "blockize" optimization. However, we also note that the "blockize" optimization is irrelevant for the manually annotated version of the tree-form recursion benchmarks

fft, matmult, nqueen and tsp, since they speculate on function calls instead of loop iterations.

It can be seen that the adaptive fork-heuristics and feedback-based selection are effective approaches to automatic parallelization using software-TLS. For loop-based benchmarks with long inner loop iterations such as raytracing and smallpt, sparse-matmul, bwaves, lavaMD, myocyte, 3x+1, mandelbrot and md, the *heuristics* version achieves comparable or similar speedups to the manually annotated version *simd-eager-ro* for the corresponding blockized or non-blockized figures, demonstrating that the adaptive fork heuristics can effectively and efficiently parallelize such benchmarks on-the-fly without the need for re-compilation, which is encouraging. Currently, we use a compiler option to specify whether to apply the "blockize" optimization for the automatically parallelized versions, which may be inconvenient for the user and may not achieve ideal speedups if different loops suit different blockization strategies. We plan to implement adaptive blockization such as OpenMP guided loop iteration scheduling for future work.

On the other hand, for loop-based benchmarks witb small nested inner loops, such as lbm, kmeans, srad and cfd, the feedback-based selection is essential to achieving effective automatic parallelization. The feedback-based selection also significantly improves the performance of the *heuristics* versions of the bh, streamcluster, fft and tsp benchmarks. The speedups of the corresponding blockized or non-blockized feedback-based selection versions of these benchmarks are also comparable or similar to the manually annotated version, though the kmeans benchmark shows unstable speedups.

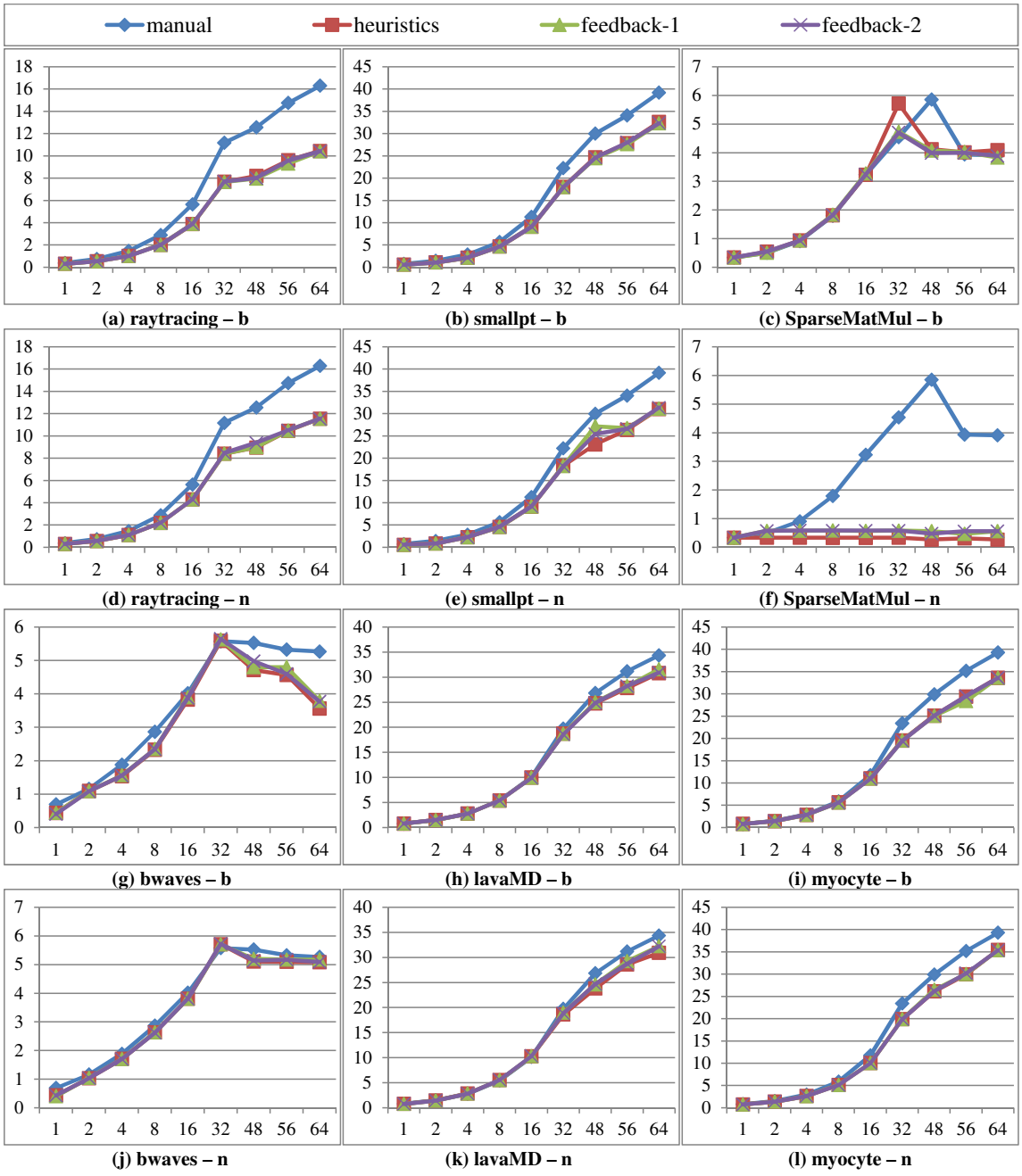


Figure 6–2: Speedup; higher is better (1/4). Adaptive fork-heuristics achieve close speedups to the manually annotated version for benchmarks with long loop iterations.

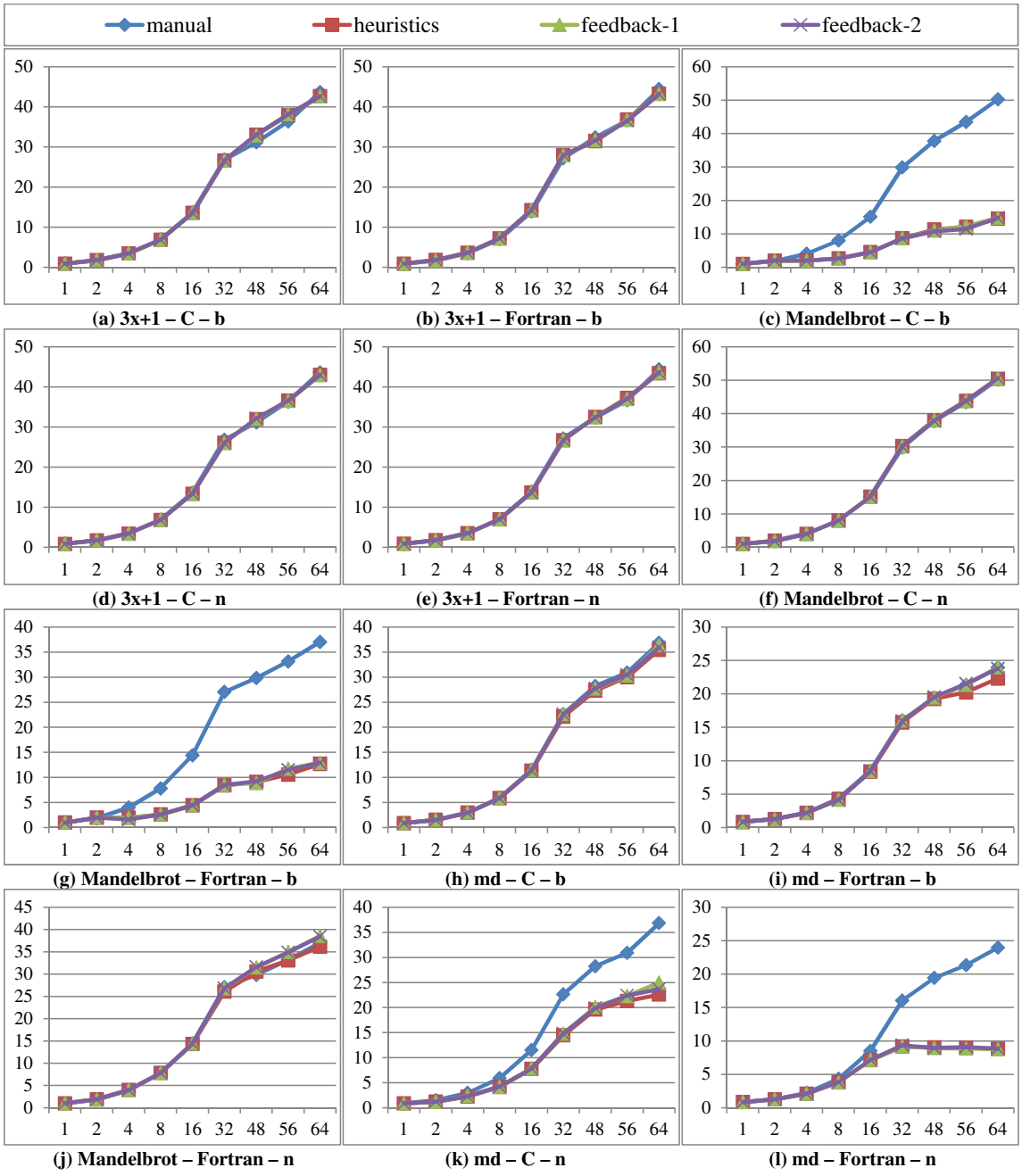


Figure 6–3: Speedup; higher is better (2/4). Adaptive fork-heuristics achieve close speedups to the manually annotated version for benchmarks with long loop iterations.

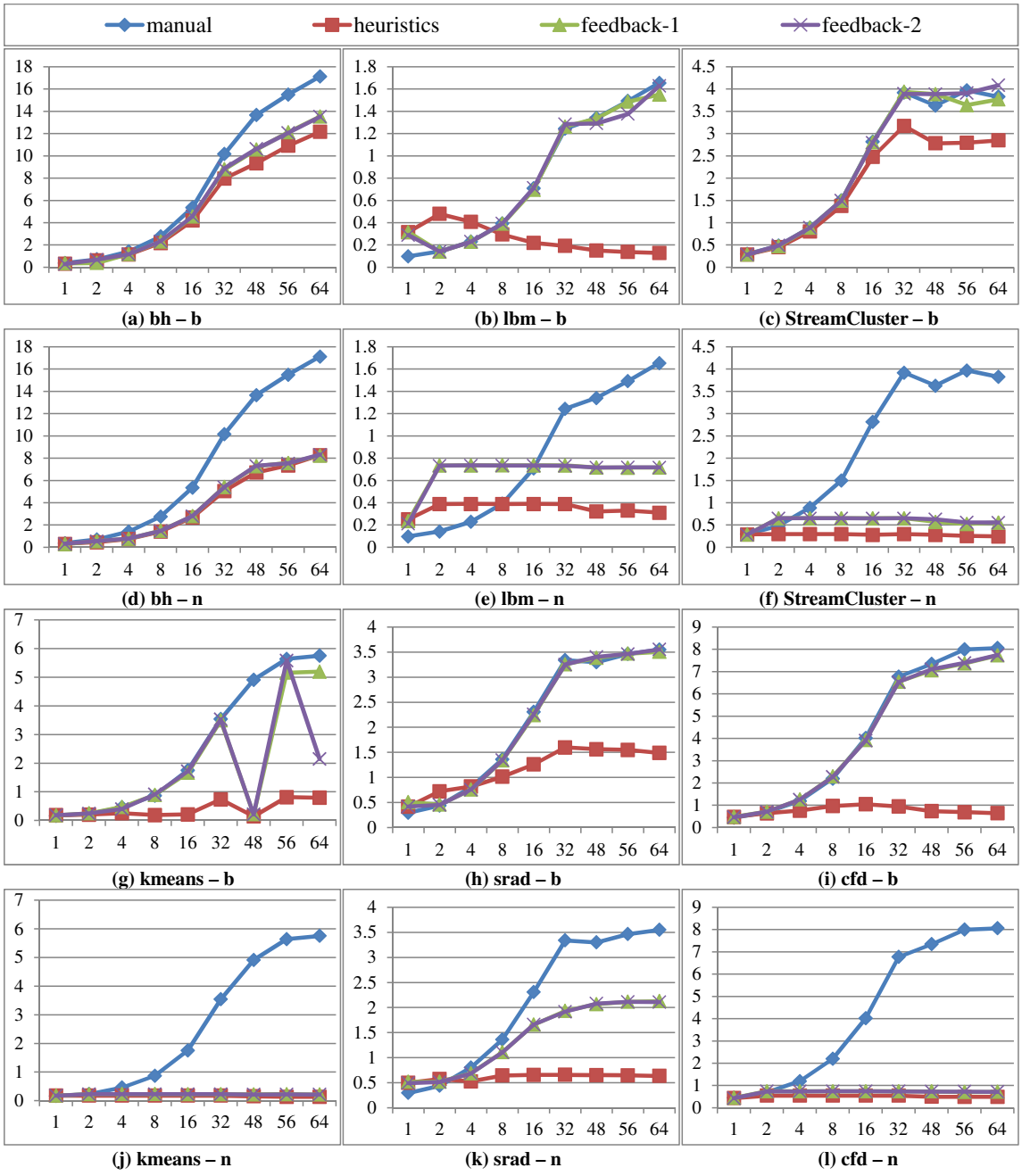


Figure 6–4: Speedup; higher is better (3/4). Feedback-based selection achieves close speedups to the manually annotated version for benchmarks with small inner loops.

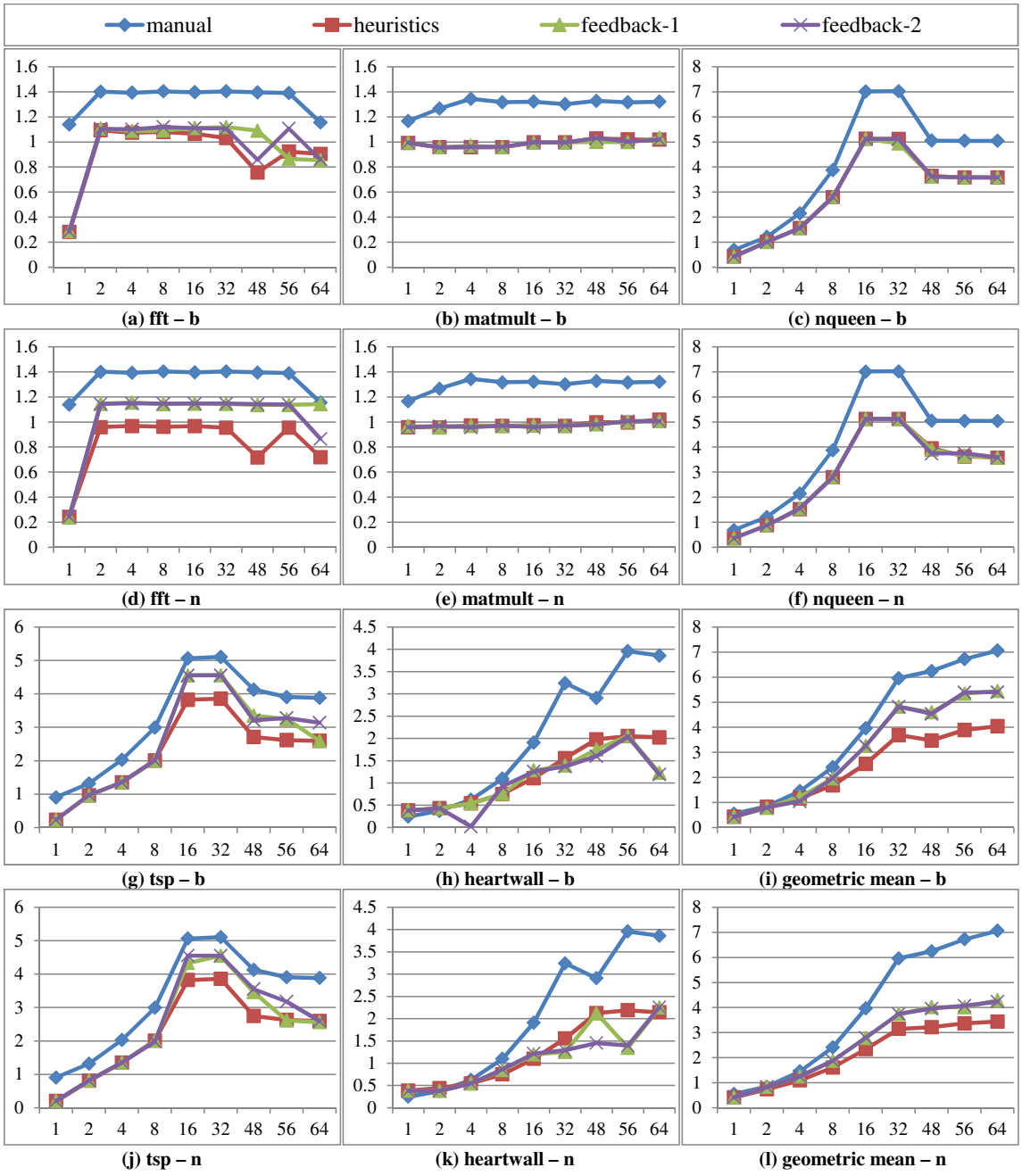


Figure 6–5: Speedup; higher is better (4/4). Feedback-based selection is slower due to inaccurate fork point selection or selected speculation enabling optimization.

For some benchmarks such as raytracing, smallpt, bh and heartwall, the feedback-based selection version has lower speedups than the manually annotated version, due to inaccurate selection of fork points. For the tree-form recursion benchmarks nqueen and tsp, that the feedback-based selection version is slower is because of the selected speculation enabling optimization that was discussed in section 3.5.2, which utilizes CPU resources more effectively and reduces overhead for the speculative parallel execution.

The geometric mean of the speedups of the *heuristics*, *feedback-1* and *feedback-2* versions over all the benchmarks are 4.0, 5.4 and 5.4 with blockization, and 3.4, 4.3 and 4.2 without blockization, respectively. The geometric mean of the *simd-eager-ro* version is 7.1. We can see that generally it is not necessary to apply the feedback-based selection more than once for the current implementation.

The number of fork points enabled by the MUTLS automatic parallelization compiler are presented in Figures 6–6 to 6–9. The *all* is the number of potential loop fork points for the adaptive fork heuristics as was discussed in section 6.2.1. The *no-feedback* is the number of loop fork points without register dependencies which are enabled by the compiler transformation that was discussed in section 6.4. The *feedback-1* and *feedback-2* are the number of fork points selected by adaptive fork heuristics for the *heuristics* and *feedback-1* version, respectively, of the speedup figures. For the manually annotated *simd-eager-ro* version, we speculate on one fork point for all benchmarks except matmult, which has three fork points parallelizing the computation of the four submatrixes. The *all* version covers the fork point of the

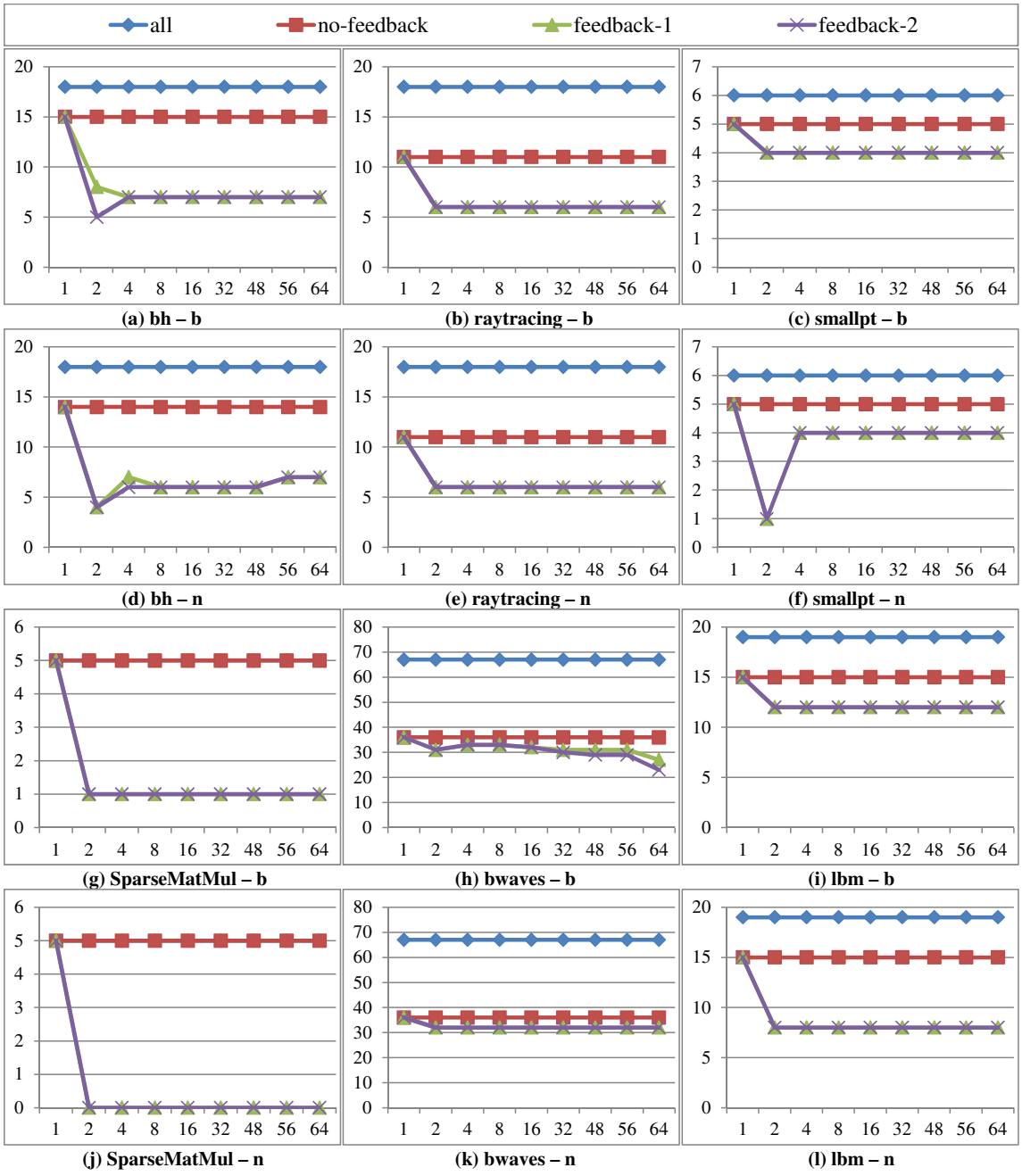


Figure 6–6: Enabled Fork Points by Adaptive Fork Heuristics and Feedback-based Selection (1/4)

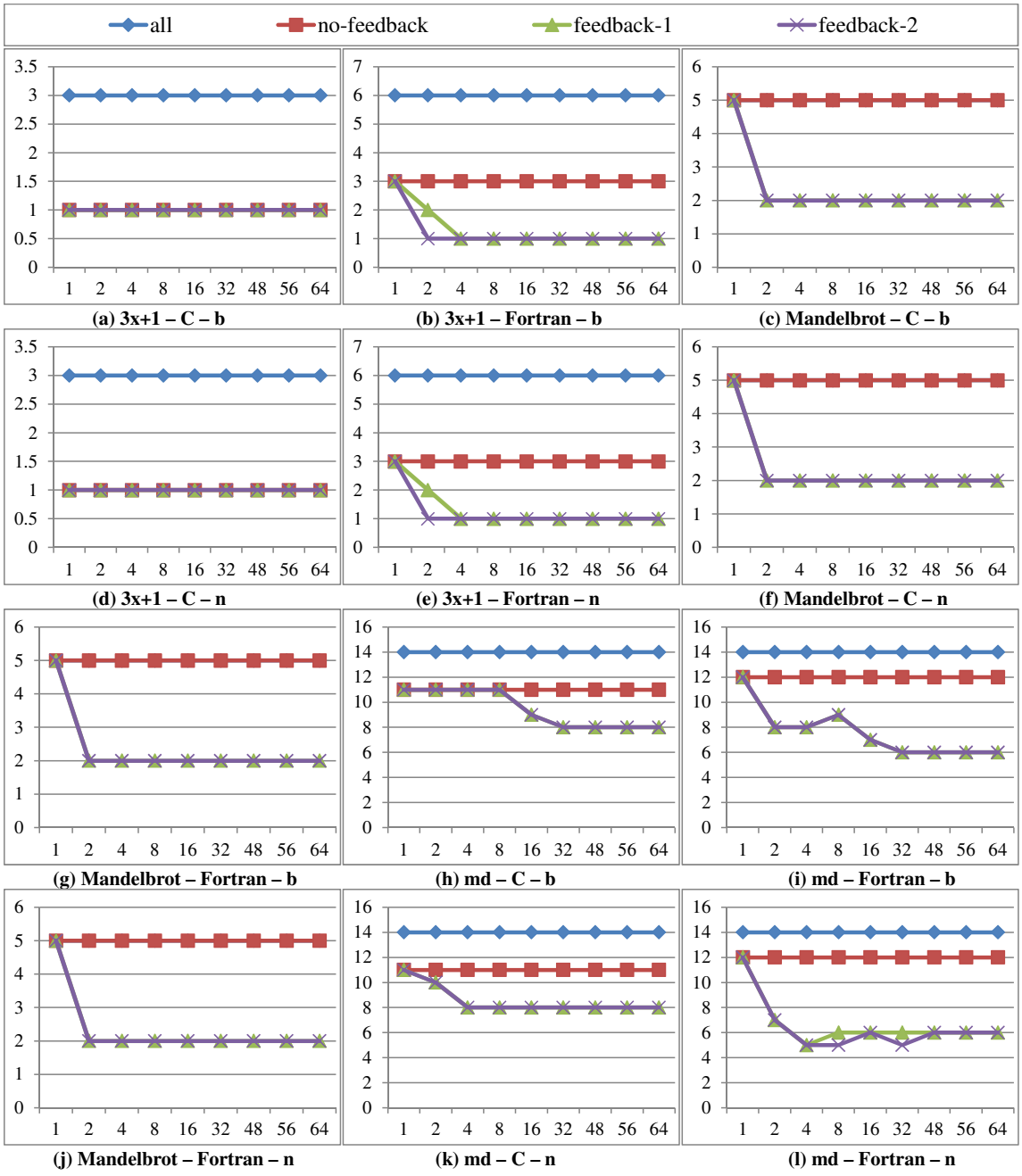


Figure 6–7: Enabled Fork Points by Adaptive Fork Heuristics and Feedback-based Selection (2/4)

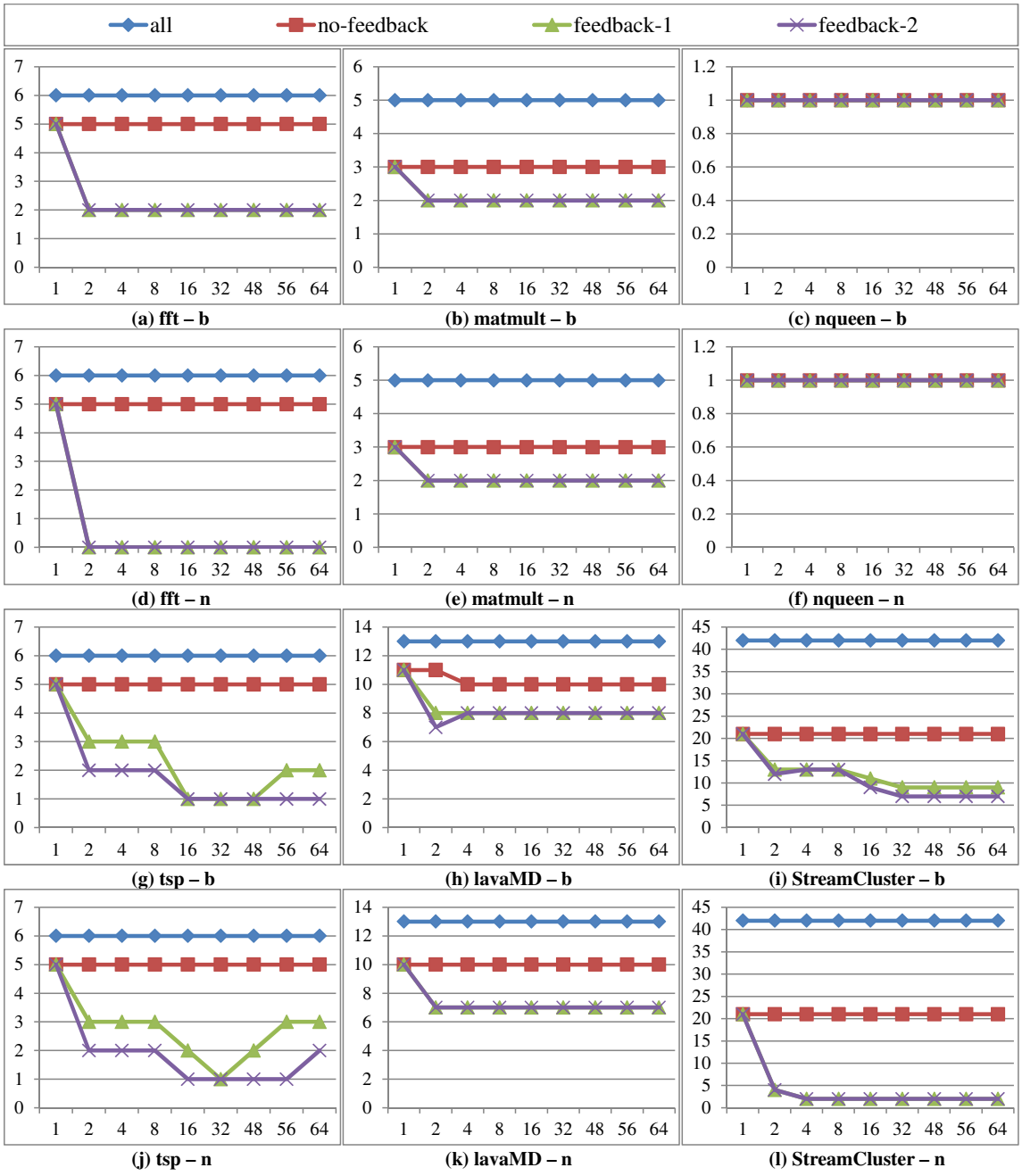


Figure 6–8: Enabled Fork Points by Adaptive Fork Heuristics and Feedback-based Selection (3/4)

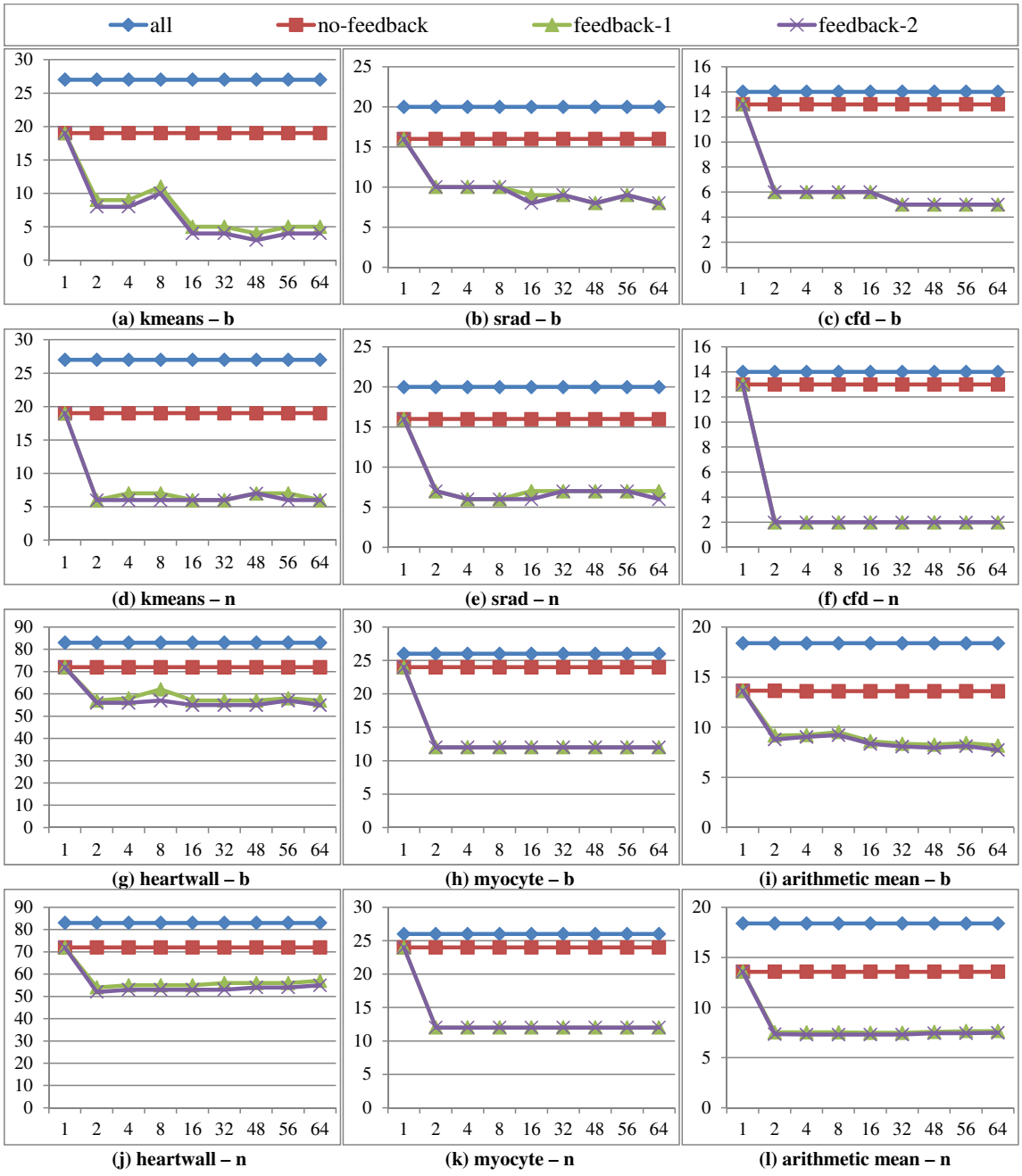


Figure 6–9: Enabled Fork Points by Adaptive Fork Heuristics and Feedback-based Selection (4/4)

manually annotated version for all the 19 loop-based benchmarks. For the depth-first search (DFS) tree-form recursion benchmarks nqueen and tsp, the *all* version covers the fork point of the loop enclosing the recursive function call speculated by the manually annotated version, which can be considered equivalent for in-order speculation. However, the *all* version does not cover the manual fork points for the divide-and-conquer benchmarks fft and matmult, which we plan to address as future work by including function calls as potential fork points.

It can be seen that both the static compiler transformation and the dynamic feedback-based selection disable a significant number of potential fork points. In addition, though not shown in the figures, they did not disable any fork point corresponding to those speculated by the manually annotated version. This demonstrates that both static compiler analysis and dynamic heuristics are valuable for automatic parallelization using thread-level speculation. We can also see that the two feedback versions select similar number of fork points for the benchmarks. The arithmetic mean over all the benchmarks for the *all*, *no-feedback*, *feedback-1* and *feedback-2* versions of the blockized and non-blockized figures are 18.4, 13.6, 8.2, 7.7, and 18.4, 13.6, 7.7, 7.5, respectively. Here we use arithmetic mean instead of geometric mean, for similar reasons as the Figures 4–17 and 4–18 of section 4.4: if no fork point is selected in only one benchmark, the geometric mean would be 0.

We can see that the feedback-based selection generally reaches stable decision for different running environments with varying number of CPU cores. Notable exceptions are the benchmarks that iteratively computes data-parallel kernels such md, streamcluster and kmeans, as well as the tree-form recursion benchmark tsp. For

these benchmarks, generally more fork points are disabled with more CPU cores until 32 cores, after which the number of enabled fork points tends to be stable for md, streamcluster and kmeans, while more fork points are enabled with more CPU cores for the tsp benchmark. For the non-blockized fft benchmark, though feedback-based selection disables all fork points from 2 cores, the MUTLS transformation makes more effective use of hardware resources such as i-cache and branch predictors that also results in speedups.

6.6 Chapter Summary

In this chapter, we proposed the adaptive fork-heuristics for automatic parallelization using TLS which automatically inserts potential fork/join/barrier points and purely relies on the TLS runtime system to disable unprofitable ones on-the-fly, which has the benefit to optimally adapt to different input data and execution environments. We also presented the feedback-based selection technique to reduce the heuristics overhead through recompilation based on the log file generated by adaptive fork heuristics. We integrated adaptive fork-heuristics and feedback-based selection into the MUTLS software-TLS framework and then implemented the related compiler transformations and optimizations to enable automatic parallelization. Experimental results demonstrate that though adaptive fork-heuristics and feedback-based selection are generally effective, appropriate workload distribution strategy such as adaptive blockization is also important for effective automatic parallelization using software-TLS. It is also encouraging that adaptive fork-heuristics can efficiently parallelize many benchmarks on-the-fly without re-compilation. We also see that both static compiler analysis and dynamic heuristics are helpful to disable unappropriate

fork points. On average, the blockized heuristics and feedback-based selection versions achieve respectively, 4.0 and 5.4 speedups, and 56% and 76% performance of the manually annotated *simd-eager-ro* version. This could be improved with better automatic workload distribution strategies.

CHAPTER 7
Dynamic Language Context

A direct and very practical application of our language and architecture independent TLS design is for ready incorporation into the execution context of more dynamic languages such as JavaScript, Matlab, and Python, where parallelism likely exists but is difficult to extract using traditional, conservative analyses and optimizations.

Some compiler frontends/runtime environments of dynamic languages such as Python and Matlab use LLVM for backend code generation. For example, The just-in-time (JIT) specializing Python compiler Numba [17] compiles decorator-annotated Python functions to LLVM functions and links them to a LLVM module. The McLab project [12] provides compilers and virtual machines for scientific programming languages such as Matlab and Aspect Matlab. Its backend is the McVM virtual machine and McJIT JIT compiler, which specializes type-specific function versions of each frontend function and generates executable code using LLVM. In this chapter, we develop a software-TLS system for Python based on the Numba and MUTLS frameworks and evaluate its performance.

7.1 Background on Numba

Numba is a JIT specializing Python compiler that dynamically translates decorated Python program at runtime to LLVM-IR, and then invokes LLVM execution engine to JIT compile and execute the generated LLVM code.

Numba uses Python decorators to annotate/transform functions that is intended to be JIT compiled into LLVM code. A decorator is a Python function that takes the decorated function as the input parameter and returns another Python function representing the decorated function, i.e. binding its name in the namespace. Numba has two important decorators: `@jit` and `@autojit`. The types of the parameters and optionally local variables are specified in the `@jit` decorator, which transforms the decorated function when Python parses the decorator. While `@autojit` transforms the decorated function based on the runtime type knowledge when the function is actually called. As a result, `@autojit` usually does not need type specifications, though the user can give hints to the type inference system by specifying type templates such as 2-dimensional arrays of arbitrary types.

Each `@jit`/`@autojit` decorator generates two LLVM functions for its decorated Python function: the working function that specializes the Python function, and the wrapper function serving as the decorator returned function that adapts the working function to the CPython interpreter. The prototype (parameter and return types) of the working function is Numba-defined and thus the working functions can directly call each other. The wrapper function gets the arguments of the working function through `PyArg_Parse*` Python/C API calls and wraps the return data by `Py_BuildValue` API calls. Not all Python features are currently supported in Numba. The working function coerces to and from unsupported Python objects such as lists and tuples using Python/C objects layer API calls.

Since Numba is mainly written in Python, while LLVM in C++, Numba uses llvmpy [8] to construct, JIT compile and execute the LLVM code. The llvmpy

framework is Python binding to LLVM that wraps the LLVM intermediate representation, analysis and transformation passes and execution engine. It comprises several Python modules providing similar interfaces to LLVM classes, and a CPython extension module linking with LLVM libraries.

When a Python programming is running, CPython processes each decorator annotated function one by one, specializing and transforming the function to an LLVM function. Numba maintains an LLVM context manager for the specialized functions in the environment. The LLVM context manager maintains a global LLVM module containing all transformed LLVM functions. Each time Numba specializes a function, it creates a new LLVM module for the generated LLVM function. In the linking stage, the LLVM context manager calls LLVM pass managers on the module to optimize the generate function, links the module to the global module, and then JIT compiles the function and updates the global variable and function pointers in the LLVM execution engine.

7.2 Python Frontend for MUTLS

As MUTLS is purely based on the language and architecture independent LLVM-IR, it is easy to add an LLVM front-end language for MUTLS. In this section, we first describe how we integrate Numba and MUTLS to get a primitive software-TLS system for Python, and then discuss improvements to make the system effective and efficient.

7.2.1 Primitive System Design

During initialization of the LLVM context manager, we first create a MUTLS module pass manager and add the MUTLS transformation pass to it, and then call

llvmpy's `load_library_permanently` function which in turn calls the corresponding LLVM method to load the MUTLS runtime dynamic library (shared object).

In the linking stage, after the generated function's module is linked to the global module, we first run the MUTLS pass manager on the global module to speculatively parallelize the module, which will speculate on functions annotated with fork/join/barrier points, as well as their nested called functions. The MUTLS pass generates a speculative version and transforms the original version for each speculated function. If a function has already been speculated (it already has a speculative version), it is not speculated again. Then we call the execution engine's `recompile_and_relink_function` method on the non-speculative version of the speculated function to update the JIT function pointer. The `recompile_and_relink_function` method was not in llvmpy; we added it to call the corresponding LLVM method.

The fork/join/barrier points are annotated in MUTLS using LLVM intrinsic functions, which Numba currently does not support. We work around this by defining an empty Numba decorated Python function `MUTLS_forkjoinpoint` for annotation. The MUTLS transformation pass then treats a function call with name prefix `__numba_specialized_` and suffix `MUTLS_forkjoinpoint` as the annotation point. It would not be removed by optimization as each specialized function is optimized in its own module. Another problem is that MUTLS automatically add `MUTLS_finalize` to a module's `main` function, while Numba modules does not have `main` function. We adopt a similar workaround by adding an empty Python `MUTLS_finalize` function which is transformed to the MUTLS library function call.

7.2.2 Optimization

The primitive Python software-TLS system of the above subsection 7.2.1 is inefficient with unnecessary rollbacks. We then address several issues specific to Python, in particular, reference counting, address space registration, and library inlining.

CPython uses reference counting for automatic heap memory management. Reference counts of Python objects can be incremented/decremented by `Py_{Inc|Dec}Ref` Python/C functions. To avoid type coercion in LLVM modules for performance, Numba re-implements reference counting subroutines `Py_{|X}{INC|DEC}REF` for Numba LLVM objects, which reads the reference count and writes back the updated one. If the reference count drops to zero, the subroutine calls `Py_DecRef` function. When the speculatively parallelized program is running, if the non-speculative thread updates the reference count of an object after the speculative thread reads the reference count, then memory conflict is detected at validation time which would cause the speculative thread to rollback.

We avoid such rollbacks by redirecting the Numba reference counting subroutines to MUTLS library call versions. The library does not keep track of the real reference count of an object, but the "reference count difference", i.e. the number (possibly negative) that should be incremented to the reference count by the speculative thread. A reference counting map with the object pointer and the reference count difference as key-value pairs is maintained for each speculative thread. When a reference counting function is called on an object, the reference count difference is fetched from the reference counting map given the object pointer. If the object is not in the map, then

the reference count difference is initialized 0. The reference count difference is then incremented/decremented according to the called function and updated in the map. During commit time, for each object in the reference counting map, the reference count is read from memory, added the reference count difference and written back. If the reference count drops to zero, `Py_DecRef` is called. This technique for handling Numba reference counting can also be generalized for other reduction operations.

During buffering load/store of each speculative thread, the MUTLS system needs to know whether the given address of the load/store is global (heap, static or the non-speculative thread stack address), local (the speculative thread stack address) or invalid (the speculative thread should be rolled back). In order to achieve this, address spaces of global objects and stack frames should be registered in the MUTLS runtime library. Objects allocated in LLVM modules are automatically registered by the MUTLS speculator transformation pass that inserts `MUTLS_register_{heap|static}` and `MUTLS_set_{self|child}_stackptr` library calls in the module. However, objects allocated by the Python interpreter cannot be registered by the MUTLS speculator pass, which causes unnecessary rollbacks if accessed. We then call `MUTLS_register_heap` in CPython's `PyMem_{Malloc|Realloc}` and link the MUTLS runtime library to rebuild CPython. Since MUTLS library is written in C++, g++ instead of gcc should be used when linking the library. We also call `MUTLS_register_heap` in Numpy's `PyArray_{NEW|RENEW}` functions to register the address spaces.

MUTLS runtime library can be built to LLVM bitcode and link with the speculated LLVM module to enable further optimizations such as inlining library calls.

However, we cannot directly replace the dynamic runtime library with the bitcode library, since we added the `MUTLS_register_heap` library call in CPython. To resolve this problem, we split the runtime library into two components of address space registration and thread management. The former only implements address space registration API functions which is built to a static or dynamic library and linked to CPython, while the latter references the former and implements all the other MUTLS library API functions, which is built to a bitcode module and a dynamic library. Then the system can be configured to run in either *inline* or *non-inline* mode. In inline mode, the global LLVM module links the MUTLS bitcode module and runs its constructors during initialization of the LLVM context manager. In non-inline mode, the MUTLS dynamic library is loaded and initialized after speculative transformation. Function inlining as well as some other optimization passes such as CFG simplification and dead code elimination are also added to the MUTLS pass manager in inline mode. Global function aliases cause recursive compilation errors in the LLVM JIT execution engine, which we resolve by optimizing to remove such aliases in the bitcode module.

7.3 Experimental Results

We experiment with the Python software-TLS system in this section, using 3 computation intensive benchmarks 3x+1, mandelbrot and md, and 3 memory intensive benchmarks fft, nqueen and filter2d. We re-implement the MUTLS benchmarks 3x+1, mandelbrot, md, fft, nqueen of Chapter 4 in Python, while filter2d is a Numba benchmark that computes 2D image filtering.

The speedup results of the speculatively parallelized benchmarks are shown in Figures 7–1 to 7–4. Since in a JIT environment there is a trade-off between parallelized program running time reduction and JIT compilation/optimization time overhead, we show the speedup results both without and with the JIT speculative compilation/optimization time in Figures 7–1, 7–3 and Figures 7–2 and 7–4, respectively.

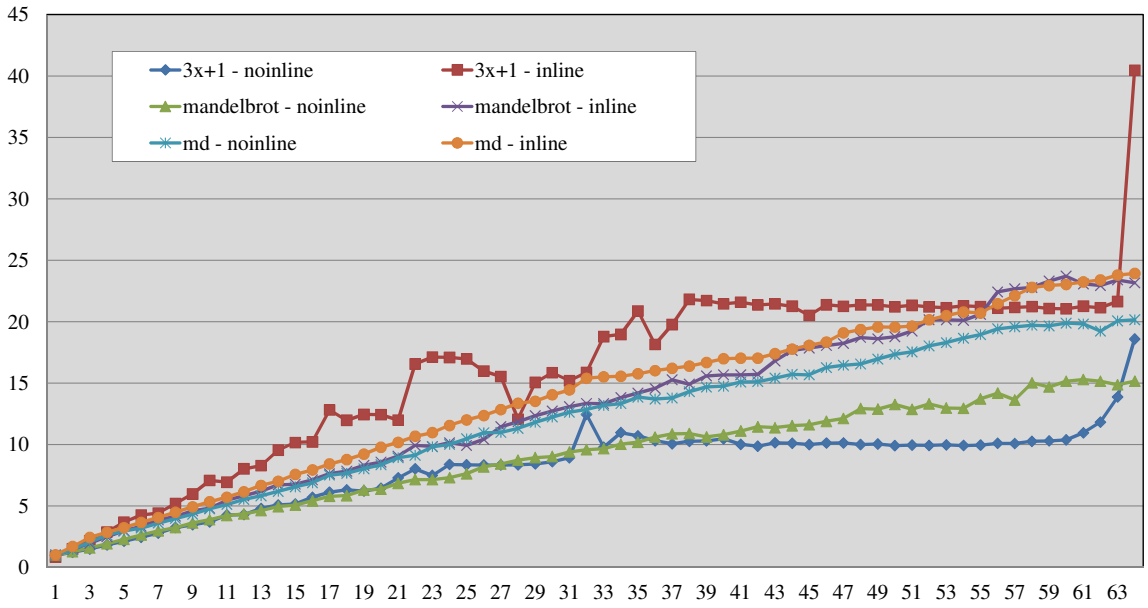


Figure 7–1: Compiled Program Speedup - Computation Intensive; higher is better

As expected, the computation intensive benchmarks achieve significantly higher speedups than memory-intensive ones, due to the memory buffering overhead of the MUTLS software-TLS system discussed in previous chapters. The reason that the speedups of 3x+1 between 32 and 63 cores are generally stable and jump up at 64 cores is our workload distribution strategy, which splits the computation into 64 loop iterations, and thus at least two iterations are computed sequentially. It can be seen

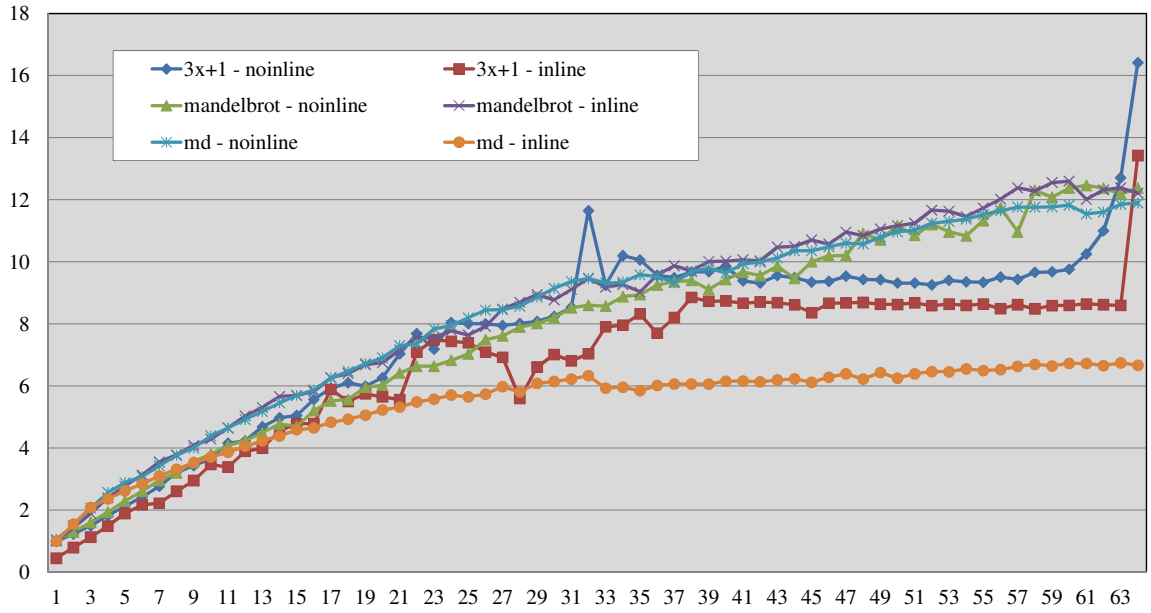


Figure 7–2: Whole Program Speedup - Computation Intensive; higher is better

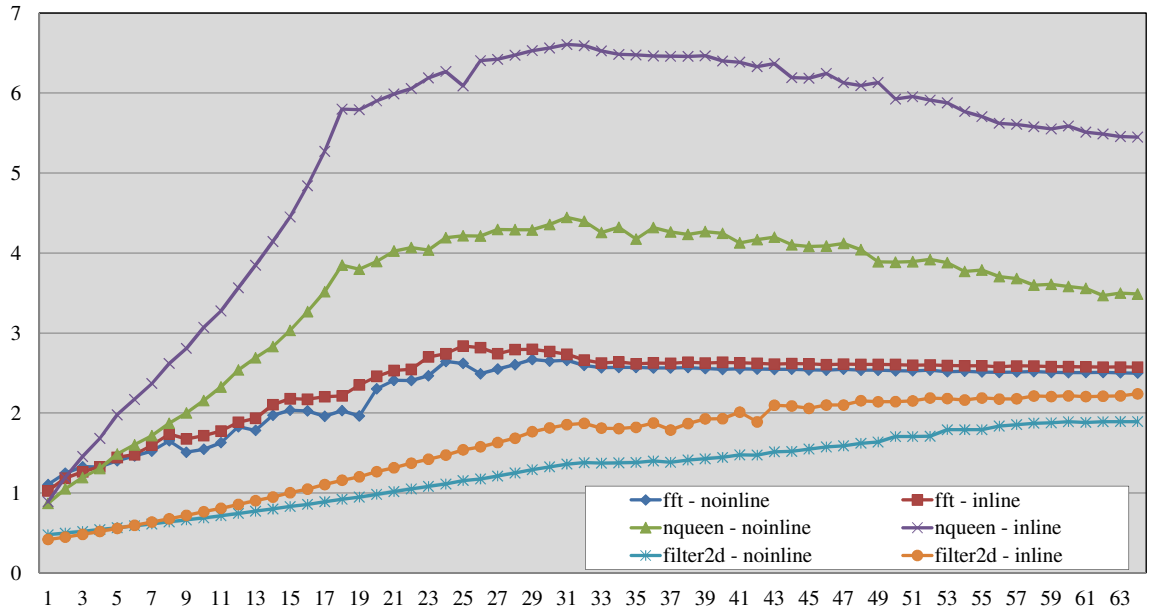


Figure 7–3: Compiled Program Speedup - Memory Intensive; higher is better

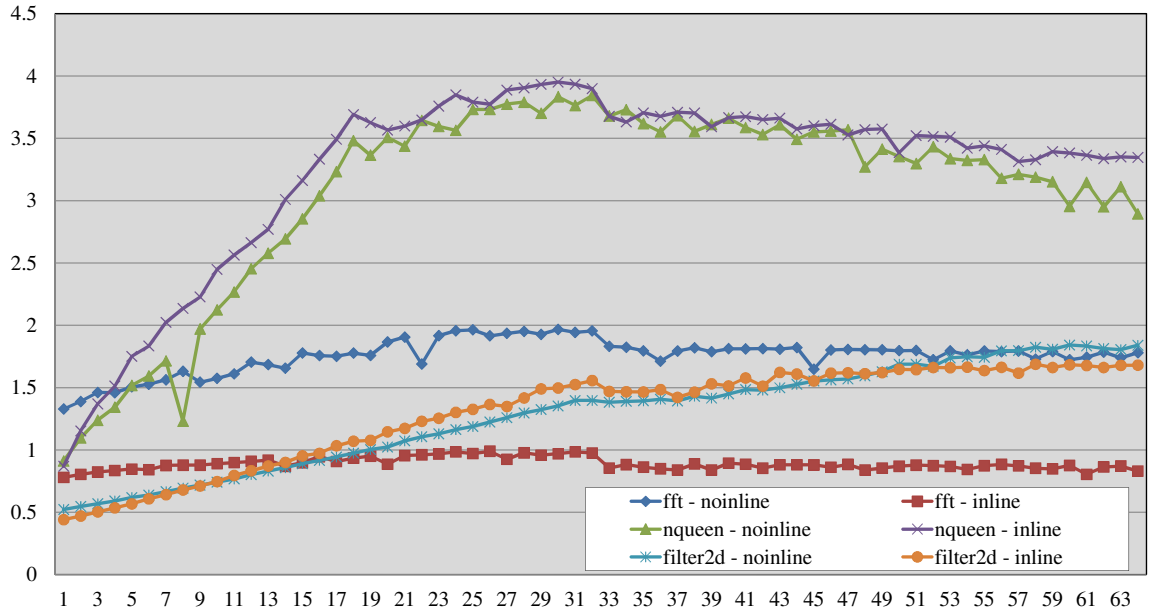


Figure 7–4: Whole Program Speedup - Memory Intensive; higher is better

that the tree-form recursion benchmarks fft and nqueen exhibit similar performance characteristics as the C versions of Chapter 4, whose speedups saturate at around 32 cores, for similar reasons of the limited parallelism in deeper recursive function calls and inter-socket extra message cost of the NUMA architecture machine.

We can also see that the JIT speculative compilation/optimization overhead is considerable for these programs/workloads, reducing the highest speedups from between 15.2 to 40.5 to between 6.7 and 16.4 for computation intensive benchmarks, and from between 1.9 and 6.6 to between 1.0 and 4.0 for memory intensive ones. However, with larger problem size and longer program execution time, we expect that the amortized overhead of JIT compilation/optimization will be reduced. On the other hand, the inlining optimization benefits the performance of the speculatively parallelized programs significantly, improving the speedups of 3x+1, mandelbrot, md, fft,

nqueen and filter2d from 18.6, 15.3, 20.2, 2.7, 4.4 and 1.9 to 40.5, 23.7, 23.9, 2.8, 6.6 and 2.2, respectively. As a result, there is a trade-off between more optimization resulting in more efficient executable programs and more JIT compilation/optimization overhead. This is demonstrated by comparing the whole program speedups of the inline and noinline versions: the inline versions have higher speedups than the noinline versions for the mandelbrot (12.6/12.5) and nqueen (4.0/3.8) benchmarks, but lower speedups for the 3x+1 (13.4/16.4), md (6.7/11.9), fft (1.0/2.0) and filter2d (1.7/1.8) benchmarks.

7.4 Chapter Summery

Dynamic languages such as Python are difficult for compilers to analyze, optimize and parallelize due to lack of type information, where dynamically parallelizing techniques such as TLS are more effective. In this chapter, we presented a software-TLS system for the dynamically typed language Python based on the Numba JIT specializing Python compiler and the MUTLS software-TLS framework. We also proposed three optimizations that reduce unnecessary rollbacks to enable effective speculative parallelization: using reduction to process Numba reference counting, address-space registration of Python objects, and MUTLS runtime library inlining into the Numba JIT compiled LLVM module. This work further demonstrates that the language and architecture independent software-TLS approach of the MUTLS system is valuable for both static and dynamic languages. Experimental results show that significant speedups can be achieved for the JIT compiled Python by software-TLS, with from 1.8 to 16.4 for whole program speedups and from 2.2 to 40.5 for compiled program speedups. While memory buffering still accounts for significant

performance overhead, JIT compilation/optimization overhead is also considerable, and there are optimization trade-offs such as inlining between efficient program execution and JIT compilation overhead.

CHAPTER 8
Related Work

In this chapter, we present related research works for the MUTLS software-TLS system and related optimizations, heuristics and programming language/execution environments discussed in previous chapters, in particular, hardware and software TLS, software transactional memory, software-TLS memory buffering, fork-heuristics, dynamic language context and hardware acceleration.

8.1 Hardware-centric TLS

Many compiler frameworks have been proposed for thread-level speculation. These typically require significant hardware support, with the most efficient and modern designs involving different, hybrid forms of software and hardware cooperation.

The bulk of these works focus on loop-level in-order speculation, for which a number of feasible performance models have been proposed. The Java runtime parallelizing machine (Jrpm) [51], for instance, identifies the best loops to parallelize by analyzing speculative loops with dynamic compilation and a hardware profiler, and speculatively parallelizes the loops on the Hydra chip multiprocessor [128]. The runtime speculative automatic parallelization (RASP) [83] technique speculatively parallelizes loops of compiled x86 binary on-the-fly using dynamic binary translation on the DBT86 [28] runtime environment and TLS and performance monitoring hardware. Du et al. propose a misspeculation-based cost-model driven compilation

framework to select effective loops for speculative parallelization [63]. STAMPede is a cooperative approach with unified hardware support for TLS [161, 163, 162]. The speculative parallel iteration chunk execution (Spice) [143] technique improves the probability of speculative thread commit by proposing a software value prediction mechanism with high prediction accuracy for some loop iterations and only speculating on those iterations. Cascadia [183] is a hardware-only TLS architecture that selects the best loop level for speculation based runtime heuristics of loop nest across function calls.

Research on hardware MLS has suggested MLS is more amenable to unstructured parallelism, as is often found in method-heavy programming contexts, such as object-oriented languages [50]. This direction includes work on specific aspects of hardware MLS, such as determining appropriate fork heuristics to reduce speculative overhead [174, 177] and misspeculation [175], and a limit study on the potential speculative method-level parallelism in imperative and object-oriented programs [173]. MLS can work well with an out-of-order design, but for simplicity of hardware implementation these works usually assume in-order speculation.

Ioannou et al. [87] compared different hardware-centric TLS paradigms with various configurations such in-order/out-of-order and loop/method/both, suggesting that TLS performance varies significantly on different hardware architectures, and that performance is hindered by both data dependence and load imbalance/parallel thread coverage.

The Anaphase compiler/architecture [106, 107] exploits fine-grain speculative thread-level parallelism by decomposing independent program instructions in basic

blocks to different speculative threads based on profiled Program Dependence Graph (PDG) [67]. Therefore, parallelism is achieved at instruction task level. Forking model is not important in the current study since there are at most two threads running simultaneously.

Bhowmik and Franklin [38] propose an arbitrary point speculation compiler framework on the SUIF-MachSUIF [77] platform targeting linear-form mixed forking model hardware TLS. It partitions the loops and basic block groups of a function for speculative parallelization based on profiling and data/control dependence analysis. The Mitosis [140, 105] compiler/architecture is a linear-form mixed model arbitrary point TLS system. It uses a profiling-based estimation model to find fork/join point pairs called SP/CQIP. The order of a new speculation is defined to be just after the last thread speculated on the same SP/CQIP pair. POSH [101, 145] is another linear-form mixed model TLS system targeting both loop- and method-level parallelism. It requires the compiler to insert the fork/join points such that threads speculated by the same thread are joined in reverse order, which is used to assign the order of the speculative threads. Therefore it relies on a correct compiler control-flow analysis to distinguish nested structures such as function calls or loop-nests. Given the order of threads, these systems treat speculative threads the same as in-order speculation.

In most models thread joining is a highly linear process, with the rollback of one thread potentially causing cascaded rollbacks of all subsequent speculative threads in execution order. García-Yágüez et al. propose a mechanism to avoid such cascades, rolling back only threads that have consumed values from an aborted thread [70].

Our design for tree-based rollback also reduces these dependencies, more coarsely, but without the need to build a thread dependency matrix.

Hardware-centric approaches generally use compiler-based fork heuristics to find appropriate fork/join points. Of course the same underlying techniques can be applied through manual specification based on programmer directives [137, 138, 129], a general design approach adopted by most software approaches, as well as by us to simplify our current implementation.

Praun et al. [170] propose the Implicit Parallelism with Ordered Transactions (IPOT) deterministic parallel programming model to support speculative parallelization based on hardware-TLS. It also presents an online tool to effectively identify fine-grain parallelization opportunities by recommending transaction boundaries based on sequential execution.

8.2 Software-only TLS

With the absence of readily available TLS-specific hardware and growing ubiquity of commodity multiprocessors, pure software-based approaches have seen increased attention. We survey software-TLS approaches for traditional statically typed languages/runtime environments in this section and dynamic languages/runtime environments in section 8.6.

Focusing on out-of-order MLS in Java, Pickett and Verbrugge describe the SableSpMT software-TLS system, based on an interpreted virtual machine context [132, 133, 134, 131]. SableSpMT was then extended to allow mixed forking

model [135], and could be extended even further though method outlining to support arbitrary point speculation. This effort aimed primarily at facilitating TLS analysis, but other works have focused directly on showing speedup.

Ding et al. propose the in-order, arbitrary-point speculation system behavior oriented parallelization (BOP), showing that even a coarse-grained strategy based on spawning system processes (rather than threads) can generate speedups of 2.08 to 3.31 for 3 SPEC CPU benchmarks on an 8-core Intel Xeon 7140M machine, while still providing safety [60]. Fast-track [92] is another in-order arbitrary point software speculation system, whose programming interface indicates the *fast track* to execute sequentially and the *normal track* to be speculatively parallelized running on different processes. It is applied to parallelize the GCC memory safety-checking library Mudflap [64], reducing the checking time by factors from 2 to 7 for 4 SPEC CPU2000 and SPEC CPU2006 benchmarks.

Other, more TLS-specific approaches have also been proposed. Oancea and Mycroft present an optimistic C++ library for in-order software thread-level speculation SpLSC [124] as well as an in-place implementation SpLIP aimed at independent loops that rolls back for all WAW, WAR and RAW dependencies [125]. SpLSC and SpLIP achieve speedups of 0.09 to 5.86 and 1.44 to 5.88 respectively, for 7 applications from SciMark2, BYTEmark and JOlden benchmark suites on an 8-core AMD Opteron 2347HE machine. Yiapanis et al. [179] described two loop-level in-order Java software-TLS systems MiniTLS and Lector based on eager and lazy version management that achieve 7x and 8.2x speedups, respectively, for seven benchmarks using 32 threads on a UltraSPARC T2 machine. JaSPEx-MLS [33, 32] is an in-order

software method-level speculation (MLS) framework based on the OpenJDK Hotspot virtual machine (VM).

Safe futures [176] is an explicit, transaction-based speculative parallelism approach for Java. Safe futures are like MLS except that the future thread can be explicitly claimed (joined) in the continuation thread by the programmer. Safe futures apply the linear-form mixed forking model using logical semantics to order the threads and join as in-order speculation.

García-Yágüez et al. [69] address and present efficient solutions for robustness issues that should be solved for software-TLS to be used in productive environment, including speculative exceptions such as floating-point exception (FPE) and segmentation fault, out-of-bound memory access corrupting speculative metadata, branching out of speculative region, and falling into endless loop. Ke et al. [90] present dynamic dependence hints to support safe parallelization of do-across loops in the BOP speculative parallelization system [60]. Aguilar and Campero [31] study explicit software speculative parallelism on different speculative regions such conditional branches, loop iterations and mutual exclusion critical sections, achieving around 1.8x speedups for 4 SPEC CPU2000 benchmarks on a 4-core Intel Pentium machine.

While design of all these software and library approaches have informed our overall design, we extend them by developing a true cross-language, cross-platform design with a tree-form mixed forking model, fully incorporated into a compiler context. Table 8–1 provides a summary of the main approaches, how they differ, and

Table 8–1: Comparison of TLS systems

		Language	Forking Model	Speculative Region
Hardware	Jrpm [51]	Java	in-order	loop iteration
	SPT [63]	C	in-order	loop iteration
	STAMPede [162]	C	in-order	loop iteration
	Mitosis [140]	C	mixed (*linear*)	arbitrary
	POSH [101]	C	mixed (*linear*)	nested structure
	Spice [143]	C	in-order	loop iteration
	Cascadia [183]	C	in-order	loop iteration
	RASP [83]	arbitrary	in-order	loop iteration
Software	SableSpMT [132]	Java	out-of-order	method call
	Safe futures [176]	Java	mixed (*linear*)	method call
	BOP [60]	C	in-order	arbitrary
	SpLSC/SpLIP [124, 125]	C++	in-order	loop iteration
	Fast track [92]	C	in-order	arbitrary
	JaSPEx-MLS [33, 32]	Java	in-order	method call
	MiniTLS/Lector [179]	Java	in-order	loop iteration
	MUTLS [45]	arbitrary	mixed (*tree*)	arbitrary

where our design is situated. However, we do not consider hardware-TLS systems to be architecture independent as they need dedicated TLS hardware support.

8.2.1 LLVM

Our approach is built on the Low Level Virtual Machine (LLVM) [7]. LLVM is a popular framework for compiler research of various forms. Tristan et al., for example, present an approach to translation validation, verifying intra-procedural optimizations within LLVM [169]. Other work on parallelism has also used LLVM. Work on enforcing deterministic scheduling has been based on LLVM traces [53], and Prabhu et al. build a commutativity based programming extension on LLVM enabling multiple forms of implicit parallelism [139]. As far as we are aware, however, our work

is the first to use LLVM for speculative parallelism, although there has been recent work proposing to use LLVM to target IBM's BlueGene/Q TLS architecture [37].

8.3 Software Transactional Memory

Software Transactional Memory (STM) is a concurrent/parallel programming technique using optimistic concurrency control [20] to address lock-based programming issues such as contention [10], deadlock, livelock [3] and priority inversion [24]. As with software-TLS, STM buffers/logs the words/objects accessed by each transaction and rolls back conflicting ones when dependencies are detected. Here we present an overview of some related STM works.

Shavit and Touitou [152] introduced STM with a lock-free implementation based on `Load_Linked/Store_Conditional` (LL/SC) [9]. Herlihy el al. [82] proposed the Dynamic Software Transactional Memory (DSTM) to support dynamic-size data structures, which is based on a weaker non-blocking form obstruction-freedom [15] and thus is simpler and more efficient than lock-free implementations. It can use modular contention managers to implement different strategies for conflict resolution with information such as time, operating system scheduling and hardware environments. A variety of contention management strategies have been presented, such as Polite, Karma [85], PublishedTimestamp, Polka [86], Greedy [75], Timid [57, 65], and comprehensive strategies with good performance for both short and long transactions [62], and both low and high contention workloads [155].

There are also researches on STM conflict detection and validation. Spear et al. [157] presents a survey of conflict detection and validation strategies, proposing

the global commit counter heuristics to reduce validation overhead and mixed invalidation to delay read-write conflict detection since both transactions can succeed if the reader commits first. The Lazy Snapshot Algorithm (LSA) [146] provides transactions with consistent view of objects though a global clock counter, which is then improved to reduce contention by using an external or multiple synchronized clocks for time-based transactional memories [147]. This idea has been used by many STMs such as TL2 [57], McRT-STM [149], TinySTM [65] and SwissSTM [62]. Ramadan et al. [141] propose dependence-aware transactional memory (DASM) model and prove DASM accepts all conflict-serializable concurrent interleavings. InvalSTM [73] uses commit-time invalidation that resolves conflicts with in-flight transactions to increase throughput.

Other techniques to improve STM performance such as eager version management [80, 149] and runtime system parallelism [118, 142] have also been proposed, which we will overview in section 8.4.

Many optimized STMs have been proposed, of which most are lock-based blocking designs. The transactional locking II (TL2) [57] STM based on the transactional locking (TL) [58] framework achieves competitive performance with manual fine-grained concurrent structures on small transactions by guaranteeing consistent memory state though version-clock validation. JudoSTM [127] uses dynamic binary rewriting to support C/C++ programs with irreversible system calls and library functions with locks. RingSTM [158] only requires at most one read-modify-write operation for a transaction by representing read and write sets as Bloom filters [39]. TinySTM [65] achieves significant performance gains by dynamically tuning runtime

parameter configuration such as cache locality with varying workloads. SwissSTM [62] increases inter-transaction parallelism by using commit-time and encounter-time conflict detection for read/write and write/write conflicts, respectively, and uses a two-phase contention manager to prioritize long transactions with no overhead for short transactions. Spear [154] proposes a lightweight STM that dynamically adapts different parameters of a given STM implementation, between different STM implementations and between STM and coarse-grained locks, which can substitute contention management.

Non-blocking STM optimizations have also been proposed. RSTM [109] reduces cache-misses on common-case path by not allocating transactional metadata dynamically, and makes reader transactions to writers without additional metadata copying though epoch-based storage management. Marathe and Moir [110] uses ownership stealing in a non-blocking STM to enable blocking STM optimizations such as timestamp-based validation and streamlined fast path. Based on a Java language-level STM JVSTM [40], Fernandes and Cachopo [66] implemented a lock-free STM which scales up to 192 cores.

There are also researches devoted to enhance the correctness guarantees and applicability of STM. Privatization addresses that transactional commit must be atomic and invalid transactions must abort safely, on which both correctness criterion [76] and strategies [156] have been proposed. Marathe et al. [159] propose partially visible reads to reduce transparent privatization overhead. Spear el al. [160] explore alternatives to inevitable execution of irreversible operations such as interactive I/O to improve scalability. Researches on open [123] and closed [34] nested transactions

have also been proposed, in which committed nested transactions are visible to the global main memory and only to the enclosing transactions, respectively.

Barreto et al. [36] propose a middleware TLSTM to combine software thread-level speculation (STLS) and STM based on SwissTM [62], achieving 48% throughput increase over SwissTM for long read-dominated transactions.

As both STM and TLS are optimistic parallelizing techniques using buffering, they are often considered similar. However, we note that there are differences, resulting from the fact that STM is to address issues of parallel programming while TLS is to parallelize sequential programs. Since STM is used for parallel programs, synchronization and commit ordering among transactions is generally not required for correctness. As long as a transaction completes without conflicts, it could commit irrespective of the execution of other transactions. While for TLS, to maintain sequential semantics, a speculative thread may only commit after the non-speculative thread reaches the join point from which the speculative thread started, as will be discussed in detail in section 9.1.4.

8.4 Memory Buffering

Memory buffering has been the concern of many researches on software-TLS and other speculative parallelization approaches.

The LRPD test [144] proposes speculative parallelization of loops using DOALL runtime checking based on array variable privatization and reduction parallelization, which targets distributed shared-memory architecture efficiently if the number of processors is comparable to the number of iterations. To avoid re-execute the whole loop in case of rare dependencies the R-LRPD test [55] is proposed using the sliding

windows strategy. Cintra and Llanos [52] propose a software-TLS scheme that also applies sliding window to reduce memory overhead and improve load balance. As with the lazy version management buffering, these approaches use the serial commit phase that reduces scalability.

Rundberg and Stenstrom [148] put forward a software-TLS system targeting shared memory multiprocessors, which enables *parallel commit* after speculative execution by detecting dependency and forwarding data on-the-fly during speculative loads/stores. The parallel commit design explores parallelism at a coarser granularity than parallelized V/C, but the approach may have impractically large memory space overhead in the presence of aliasing [125].

Oancea et al. [125] propose SpLIP, a lightweight, in-place update (eager version management) software-TLS approach to achieve higher scalability. We make use of this general technique as well, exploring it in conjunction with an optimized lazy version management scheme that also improves scalability, rather than as a direct replacement.

Yiapanis et al. [179] present a compact version management data structure and applied it to propose an eager and a lazy version management software-TLS systems, MiniTLS and Lector, achieving average speedups of 7x and 8.2x respectively using 32 threads. The rollback procedure of MiniTLS is parallelized, while the serial commit procedure of Lector is not. Our lazy version management buffering exploits coarse and fine grain parallelism of the runtime system to improve its scalability during serial validation/commit.

Eager version management has been proposed for software transactional memory (STM). Harris et al. [80] propose the eager version management direct-access Bartok-STM, supporting short scalable concurrent benchmarks with less than 50% overhead over the thread-unsafe baseline, and long atomic blocks comprising millions of shared memory accesses with 2.5x to 4.5x slowdown. Saha et al. [149] propose the eager version management McRT-STM, analysing STM design tradeoffs such as pessimistic/optimitic concurrency control, eager/lazy version management, and cache line/object based conflict detection. It is then improved by compiler and runtime optimizations [30] to support composable [79] and nested memory transactions.

Parallelism of the runtime system has also be explored by STM approaches. STMlite [118] removes lock overhead of transactional execution by centralizing transaction bookkeeping in a single core so that it runs in parallel with work threads. Raman et al. [142] proposed a Software Multi-threaded Transaction (SMTX) system that exposes data parallelism by dividing loop iterations to different pipeline stages while still committing them together. SMTX uses a separate centralized commit process to reduce inter-core communication latency on the critical path.

Several speculative parallelization systems propose efficient memory buffering load/store implementations. BOP [60] uses processes instead of threads for data protection. It enables *strong isolation* where memory overhead is proportional to the data size accessed instead of the number of data accesses, which benefits applications with high temporal locality. Abadi et al. [29] introduce a strong atomicity transactional memory by using page memory protection hardware to detect transactional and non-transactional memory access conflicts, achieving within 25% of a weak

atomicity implementation on C# versions of many STAMP [122] benchmarks. Tian et al. [167] proposed the Copy-or-Discard (CorD) speculative execution model that finds multi-versioned variables using a mapping table, and optimized it for dynamic pointer-intensive data structures [166] and to support incremental rollback recovery [168]. LSA-STM [146] improves lazy version management for object-based software transactional memory (STM) with eager ideas, which uses validity ranges to avoid revalidating previously read objects for each new reads.

BOP [60] and SpLSC [124] propose optimizations for readonly shared variables, but required profiling tools support or programmer specification to find likely-readonly variables. We propose the readonly-page optimization to automatically find and optimize readonly variables on-the-fly during speculative execution. Oancea et al. [125] discussed that integrating different memory buffering approaches should be more beneficial than single buffering implementation. We provide the first buffering integration solution, which seamlessly integrates different buffering or pseudo-buffering implementations into one buffering framework and automatically utilizes the most appropriate one for each variable.

DynTM [104] and SEL-TM [182] are hardware transactional memory (HTM) systems with hybrid conflict management policies that permit eager and lazy version management to cooperate during transactional execution. ASTM [108] and adapt-STM [130] use heuristics to select eager or lazy version management for software transaction memory (STM) threads. We propose the first software-TLS solutions to allow speculative threads to utilize both eager and lazy version management buffering.

Garzaran et al. [71] analyze complexity-benefit tradeoffs of different hardware-TLS memory buffering approaches. It discussed strengths and weaknesses as well as the required hardware supports for each approach, and compared the mechanisms using 7 benchmarks with different memory access patterns. Our work improves software-TLS memory buffering approaches and integrates them into a unified framework with the aim to maximize their combined strengths.

8.5 Fork Heuristics

The bulk of proposed fork-heuristics are static profiling heuristics. Java runtime parallelizing machine (Jrpm) [51], for instance, first profiles execution of a sequential program with a hardware profiler, and then dynamically speculates on the selected prospective loops after collecting enough profiling data to decide the best loops to parallelize. Du et al. [63] propose a cost-model-driven compilation framework SPT to select candidate loops for speculative parallelization, which builds control-flow graphs and data-dependence graphs with profiling information of a sequential execution and uses the graphs to evaluate candidate loops based on the cost model. The STAMPede [162] TLS approach selects speculatively parallel loops based on several filter criteria: the loop execution coverage and iteration count are above a threshold and the loop body is neither too large nor too small. The Mitosis [140] compiler/architecture uses arbitrary pairs of basic blocks as fork/join points and models parallel execution based on profiling traces to estimate candidate pairs. The POSH [101] compiler simulates sequential execution on a train input set and models TLS parallel execution to select beneficial fork/join points.

There is also research dedicated to static profiling heuristics. Warg and Stenstrom propose heuristics to reduce speculative threading overhead based on the module (method) run-length threshold [174], and to reduce misspeculation using history-based prediction [175]. Whaley and Kozyrakis [177] propose three classes of heuristics for method-level speculation, and found that single-pass heuristics lead to best speedups while simple/complex multi-pass heuristics tend to over/under speculation. Wang et al. [172] constructed a loop-graph and used it for global loop selection to maximize program performance. Liu et al. [102] proposed an online-profiling approach to speculatively parallelize candidate loops. Online static profiling approaches [51, 102] have the advantage over offline-profiling that they do not require additional profiling input and can dynamically profile on the real data. However, these still lack parallel execution environment information for accurate estimation of fork/join points. Pure static heuristics are less common, since they lack runtime parameters. Dou and Cintra [61] proposed a thread-tuple cost-model to estimate speedups of candidate loops. As well, some heuristics combine profiling and static approaches, such as SPT [63].

Dynamic profiling heuristics have recently been studied. Luo et al. [103] proposed a dynamic performance tuning technique for selection of candidate loops. It used hardware performance monitors to profile runtime statistics such as instruction fetch penalty and cache miss, and estimated the efficiency of each thread and loop with the statistics. Unfortunately, this approach cannot be directly applied to software TLS without low-level and machine-specific access to hardware performance monitors.

All the above are hardware-TLS heuristics, which tend to focus on finer-granularity parallelism due to hardware resource constraints. In software-TLS, heuristics should focus on coarser-granularity parallelism as software TLS has higher overhead than hardware implementation. So far as we know, the adaptive heuristics we propose are the first heuristics specifically proposed for and validated in software TLS.

Adaptive software speculation [89] improves the usability of BOP [60] by dynamically predicting the profitability of each possibly parallel region (PPR) based on misspeculation rate. HEUSPEC [178] is a software speculation parallel model that dynamically adapts to different value predictors and granularity tasks. While adaptive fork-heuristics target the problem of fork point selection of software-TLS.

8.6 Thread-Level Speculation for Dynamic Languages

Recently there are researches to apply software-TLS for dynamically-typed languages such as JavaScript in addition to static languages such as C/C++, Java and Fortran. Martinsen et al. [111, 112, 113, 114, 115, 116] implemented method-level linear-form mixed model speculation software-TLS system in Rhino and Squirrelfish JavaScript engines and achieved significant speedups on some popular web applications such as YouTube (8.4x), BlogSpot (4.5x), WordPress (3.8x), Imdb, Myspace, Ebay, MSN and Wikipedia (2-3x) on an 8-core computer. Mehrara et. al [119] developed an loop-level in-order dynamic parallelization system ParaScript for JavaScript applications, which combines data-flow analysis and runtime dependency detection based on reference counting and array index range checking, achieving an average of 2.18x for SunSpider benchmark suite and Pixastic image processing library filters over the SpiderMonkey Firefox engine on an 8-core machine. Fortuna et al. [68] conducted

a potential speedup limit study on JavaScript speculative parallelism for popular web pages and standard JavaScript benchmarks and showed encouraging speedups of averaging 8.9 and as high as 45.5. Crom [121] is a JavaScript-implemented event-based speculative engine that generates speculative event handlers for precomputation in the browser, achieving an order of magnitude speedup of subsequent page loads by speculative page fetching and layout precomputing. Tan et al. [165] explored speculative parallel optimization in JIT compilers and showed at most 23.7% speedups in V8 JavaScript engine.

8.7 Hardware Acceleration

With the trend of increasing CPU capability and system heterogeneity, more effective and efficient software-TLS designs with the aid of special commodity hardware acceleration such as hardware transactional memory (HTM) [41, 117, 59, 47] and Graphics Processing Unit (GPU) [84, 35, 120] have been proposed.

After decades of intensive research, hardware transactional memory (HTM) has finally made into commodity architectures, such as the HTM of IBM Blue Gene/Q [171, 95] and System z [88], and Intel Transactional Synchronization Extensions (Intel TSX) of the Haswell microarchitecture [180, 54, 97]. Odaira and Nakaike [126] study thread-level speculation implementation using the Intel TSX hardware transactional memory and get up to 11% speedup but slowdown in most cases, suggesting that to implement effective TLS future HTM should support not only ordered transactions but also data forwarding, synchronization, multi-version cache and word-level conflict detection.

Samadi et al. [150] propose a static/dynamic CPU-GPU collaborative compiler platform to automatically speculate possibly-parallel loops on GPUs and achieve up to 12x, 24x and 37x speedups for 6 rewritten Polybench [22] benchmarks over 4-, 2- and 1-thread CPU execution, respectively. Diamos and Yalamanchili [56] propose a Kernel Level Speculation (KLS) technique to extract parallelism from heterogeneous systems with multiple GPUs and achieve 41.2% to 98.6% of the theoretical ideal performance and 1.02x to 6.13 speedups over the non-KLS version on a system with a 4-core CPU and 6 GPUs. Liu et al. [99, 100] explore the performance of value prediction and speculative execution on GPUs and found that hardware extension reduces control divergent operations significantly with moderate hardware and power consumption overheads. Sun and Kaeli [164] exploit aggressive value prediction on a GPU and obtained up to 6.5x speedup on selected kernels from SPEC CPU 2006, PARSEC and Sequoia benchmark suites. Zhang et. al [181] propose the GPU-TLS runtime system to speculatively parallelize possibly-parallel loops and achieve 5x to 160x speedups on two machines with Nvidia GPUs. Based on GPU-TLS, Han et al. [78] propose the Japonica compiler framework and runtime system to automatically parallelize DOALL loops statically by the compiler and possibly-parallel loops speculatively at runtime, using a 16-core Intel Xeon CPU and an Nvidia Fermi GPU achieving on average 10x, 2.5x and 2.14x speedups than sequential, GPU-only and CPU-only versions, respectively.

CHAPTER 9
Conclusions and Future Work

The complexity of even a relatively plain implementation of TLS makes exploration of the many possible design choices difficult, a problem exacerbated by the potential for interaction with other aspects of an optimizing compiler and execution environment. The MUTLS system is intended to improve that situation, providing a full-featured TLS compiler framework that accommodates a wide variety of input and output contexts. With a mixed forking model, MUTLS has more potential to extract parallelism from tree-form recursion applications.

Efficient memory management is critical to the design of effective software approaches to thread-level speculation, with the competing buffering strategies used to either enforce isolation or to preserve undo information having different costs and potential benefits. We initially approached the problem as one of establishing which technique is better, developing highly optimized, but separate implementations of the buffering approaches. To reduce the expensive validation/commit phase in lazy management, for instance, we implemented a streamlined, page-table memory design that enables both coarse and fine grain parallelization of that overhead. Eager version management, on the other hand, has scalability limitations due to the need to keep multiple versions of data, a problem we address by using a single, shared address-owner buffering approach with better space complexity. Both techniques improve performance, and both benefit from further optimizations to identify readonly

data and so reduce buffering costs, but it depends very much on the benchmark and resource limits. By combining the techniques and performing an adaptive, runtime selection of the buffering mechanism we are thus able to demonstrate a design that gains the benefits of both, with a more general application that accommodates different benchmarks and numbers of available cores.

To enable effective and efficient automatic parallelization, we proposed adaptive fork-heuristics for software-TLS, which inserts all potential fork/join points and purely relies on the heuristics and runtime system to disable inappropriate ones. These adaptive heuristics have the ability to utilize the real parallel execution environment information to maximize performance. In addition, we proposed a feedback-based selection technique to reduce the heuristics overhead through recompilation using the log file generated by adaptive fork-heuristics. We integrated adaptive fork-heuristics and feedback-based selection into the MUTLS software-TLS framework and implemented related compiler transformation to achieve a fully automatic parallelizing compiler, demonstrating that software-TLS can be a practical approach to automatic parallelization of real-world applications on commodity multi-core processors.

Dynamically typed languages such as Matlab, Python and JavaScript are difficult for static analysis, optimization and parallelization, which are an ideal context for dynamic parallelization approaches such as software-TLS. We integrated the Python JIT specializing compiler Numba with the MUTLS framework to implement

a software-TLS system for Python, showing that software-TLS is effective to automatically parallelize dynamic languages and that there is a trade-off for optimization between efficient program execution and JIT compilation time overhead.

9.1 Future Work

Future work involves more fully fleshing out the design as discussed in previous chapters, adding in features and capabilities that enable deeper exploration of different aspects of TLS. This includes heap address space registration (section 3.6.1), value prediction (section 3.6.4, 6.4), potential fork points (section 6.2.1), fork-heuristics hints (section 6.2.2), feedback-based selection criteria (section 6.3) and adaptive blockization (section 6.5), as well as other improvements to the interface that simplify usage.

The following subsections put forward several other possible issues to be researched for more effective and efficient software-TLS design and implementation, which are organized into 5 categories: runtime system, compiler analysis and fork heuristics, dynamic languages, hardware transactional memory (HTM) acceleration and GPU acceleration.

9.1.1 Runtime System

For memory buffering optimization, future work involves further tuning the buffering integration mechanism—our adaptive buffering design are effective, but could perhaps be improved by maintaining different buffering integration data for different fork points, which should reduce the rollback time ratio for programs with iterations containing different speculative regions such as bwaves. More precise, and

ideally ahead-of-time identification of independent or readonly variables may also be possible through static analysis, profilers, user annotation and/or feedback logs.

We are also interested in exploiting common hardware accelerators such as hardware transactional memory (HTM) and graphics processing unit (GPU), as a means of alleviating memory buffering overhead without sacrificing the advantages of software-TLS in applying to existing, commodity hardware, which will be discussed in subsections 9.1.4 and 9.1.5.

Write-buffer value forwarding. This optimization may help reduce rollbacks for programs with RAW dependencies such as fft. If a speculative thread reads an address not in its write-set, it tries to read from the write-set of a thread representing sequentially earlier execution. If the address is not in the write-sets of those threads, it reads from its read-set or the main memory. We propose two approaches to implement this. One is to make each speculative thread read from the write-sets of its ancestor threads starting from its direct parent, since a thread always represents sequentially earlier execution than its child threads in our forking models. The other is to allow each speculative thread to read from the read-set of any sequentially earlier thread with the help of a shared *store vector* maintaining which threads write to each address. A sequential execution order number is assigned to each speculative thread at fork time and the threads in the store vector are ordered by the order number. Incorrect ordering may incur unnecessary rollbacks but does not produce incorrect results.

Preference forking models. Different forking models suit different program structures and/or execution environment; for instance, nested loops may prefer in-order speculation while head-recursive function calls may be most effective for out-of-order speculation. We may explore program characteristics and dynamically assign forking model preference values for each fork point. If several forking models are available at a fork point, probability or scheduling based approaches can be used to select a forking model based on the preference values. A child thread may not be speculated if the selected forking model does not permit speculation for the current thread. Speculative threads can also have preference values, inherited from the parent thread and the fork point where it was speculated, in which case the preference values of the thread also affect forking model selection at a fork point. Preference values can be adjusted during program execution to adapt to optimal forking models accounting for cost-benefit of the speculative threads.

Parallel thread joining model. Most software TLS systems adopt a *serial commit* model, which only allows the non-speculative thread to join threads and thus keeps all ready-for-validation speculative threads idle. While a parallel thread joining model allows speculative threads to join each other, making more efficient use of the processors. It can also reduce overhead on the critical path, including thread forking/joining, as well as memory buffering validation/commit if speculative threads share read/write memory addresses. Threads representing adjacent sequential execution can be joined if they are both waiting for validation and commit. The memory buffering and thread status of one thread is then merged to the other

if validation succeeds, and its thread resource is reclaimed and allocated for new speculative work.

9.1.2 Compiler Analysis and Fork Heuristics

For adaptive fork-heuristics, we can exploit more effective and stable heuristics and hints. With the help of hardware performance counters, we may implement more accurate cost-benefit estimation. We can also combine static compiler analysis cost model and/or static profiling data to guide potential fork/join/barrier point selection.

Analysis-aided TLS. Alias analysis may be utilized to enable optimizations otherwise impossible for memory buffering. We propose two scenarios for this optimization: (1) If alias analysis tells that a load/store must access global or local memory, we can eliminate the test of its address space and directly use the corresponding speculative memory buffering. (2) In a speculative region (such as a loop) if some memory reads are analyzed not to be aliased by memory writes of the region, then we can optimize the speculative region by not speculating these memory reads and directly reading the main memory, provided that the non-speculative thread is barriered when it reaches exits of the speculative region. LLVM has several alias analysis implementations such as basicaa, steens-aa, and scev-aa, which we can use to compare the effectiveness of the optimizations. Program performance and compilation time may both be concerns in some situations. The study would also experiment the sensitivity of the optimizations to the analysis and explore trade-offs.

Reduction delaying. If we speculate on the loop "*for i=1 to n do x+=a[i];*", then x will cause all speculative threads to rollback. However, this issue can be

resolved by delaying the reduction until commit time. Given a speculative thread with a sequence of instructions "*x=load p; y=x+d; store p, y*", then it remains sound if x is not validated and p is incremented by d during commit, provided that the speculative thread is barriered before x or y is used again. The optimization can be generalized: "*x=x0=load p; for i=1 to n do x+=f(i); store p, x*", then x is not validated and p is incremented by $x - x0$ during commit. When to apply this optimization is a decision of compiler analysis and transformation. We can study use cases of reductions and effectively implement the optimization.

9.1.3 Dynamic Languages

Numba coerces objects such as lists and arbitrary-digit integers and redirects them to the interpreter by CPython object-layer API calls, which stalls speculation. However, such stalls can be eliminated by specific processing of the API calls. We can make a shallow copy of each API-returned object using Python standard library calls and add the copy to the read-set. Before an object is written, we can make a shallow copy of the object, write the copy and add it to the write-set. The shallow copies can be validated and committed using Python standard library calls. However, this optimization might be inefficient; for instance, setting an item of a list would make a copy of the whole list. The inefficiency could be removed if Numba natively supported lists and/or other objects without using the object layer. We can study the effectiveness of this optimization and find whether further speedups can be achieved for Python objects.

We can integration the McVM/McJIT interpreter/JIT compiler of McLab and the MUTLS software-TLS system to implement software-TLS for Matlab, which

should be more straightforward than Numba since they are written in C++. During initialization of the JIT compiler, we can create MUTLS module pass manager and load the MUTLS runtime library. After a function is JIT compiled, we run the MUTLS pass manager to speculate the module. However, there is a problem that some functions may be partially compiled and speculation on those functions are invalid. We in turn add a global variable for each function in the LLVM module to indicate whether the function has completed compilation, and check the variable before speculating on a function. McVM/McJIT calls Matlab standard library functions by building objects and arrays as arguments, which unnecessarily stalls speculation for native scalar argument types since building objects/arrays is not thread-safe. We can transform Matlab library function calls with native scalar arguments to native LLVM function calls and exploit more parallelism opportunities.

9.1.4 Hardware Transactional Memory Acceleration

Intel Haswell microarchitecture processors provide hardware transactional memory (HTM) support Intel Transactional Synchronization Extensions (TSX). Intel TSX provides a new software HTM programming instruction set interface Restricted Transactional Memory (RTM), which has three instructions XBEGIN, XEND and XABORT. XBEGIN and XEND mark the beginning and the end of a transactional region while XABORT explicitly aborts a transaction. XBEGIN also specifies a fallback instruction address where execution resumes if the transaction aborts, due to either buffering conflicts or XABORT instruction. Intel TSX also adds an instruction XTEST to test whether the current CPU is in transactional execution. The fallback path may acquire a lock and re-execute the transactional region non-transactionally,

or retry the transaction again. Currently RTM detects conflicts at the granularity of a cache line and rolls back a transaction for RAW, WAR and WAW conflicts. For the current implementation, other events such as interrupts, page faults and cache replacement also cause transactions to abort.

We can accelerate memory buffering using hardware transactional memory. We can start RTM transactions at the beginning of a speculative thread function, and end the transactions in check points and terminate points when the non-speculative thread synchronizes the speculative thread. The fallback instruction address is after the RTM transaction begin marker in the speculative function, so that the speculative thread re-executes non-transactionally. Since speculative threads redirect memory loads/stores to runtime library function calls, we use XTEST to check whether the speculative thread is executing RTM transactions in the function calls, and return without memory buffering if it is. We may also generate a separate RTM speculative function version that does not redirect memory accesses to avoid the XTEST overhead. The speculative thread entry function then jumps to the corresponding version for the transactional and the fallback paths. We can also utilize compiler analysis to determine not to use RTM transactions for speculative threads that have WAR or WAW dependencies and/or access memory addresses within the same cache line.

Each speculative thread is only allowed to commit after the non-speculative thread reaches the join point from which it started execution; otherwise, after the speculative thread commits, the non-speculative thread may write the same variables read by the speculative thread before the non-speculative thread reaches the join

point, which incurs RAW true dependencies. However, in the current Intel TSX implementation, if a transactional CPU accesses the same cache line with any other CPU and at least one writes, the transaction is aborted, which prevents any shared variable synchronization between CPUs/threads.

We propose a synchronizing timestamp based approach to address the problem. The speculative thread redirects memory writes to its write buffer. For each memory read, the speculative thread reads from the write buffer if its address is in the write buffer and from memory otherwise. No validation is needed at join time, and the write buffer is committed to the main memory if speculation succeeds. When a speculative thread is forked, a timestamp field T_n is initialized to 0. When the non-speculative thread reaches the join point, it reads the CPU timestamp with RDTSC instruction and writes it to the T_n field of the speculative thread to be joined. Then it spins waiting for the `valid_status` of the speculative thread to be non-NULL, as was discussed in section 3.3. When a speculative thread reaches a terminate/barrier point, it reads the CPU timestamp T_s and commits the transaction. The speculative thread then compares its T_n and T_s; if $T_n < T_s$, meaning the non-speculative thread reached the join point before it committed the transaction, it thus commits its write buffer to the main memory and set `valid_status` to COMMIT; otherwise it sets `valid_status` to ROLLBACK.

In the following list, we present some possible extensions to the RTM implementation for support of more effective synchronization, from the easiest to the most difficult in the software implementation of the MUTLS system.

- Non-transactional address space. This address space can be used to maintain TLS system metadata. The address space is not necessarily cached as it is only accessed during forking/joining.
- Synchronization status register. A CPU can set its synchronization status register, which can be read by another CPU.
- Temporary non-transactional execution. A transactional thread can temporarily leave (out of) a transaction without commit/abort and then re-enters the transaction. When a thread is out of a transaction, the transaction can be aborted if other threads access memory addresses in its transaction buffer, but addresses only accessed when the thread is out of the transaction do not affect the transaction.
- Two-phase transaction commit. The first phase commit does not commit the write buffer to the main memory. After the first phase commit, the transaction does not have to abort if the thread only reads memory written by other threads. The second phase commit commits its write buffer to the main memory if no dependency is detected.
- Tree-form transaction nesting. A transaction can start a nested transaction, whose status can indicate whether a variable is accessed by another thread.
- Write-buffer dump. A transactional thread can dump its write buffer and abort the transaction. Then the buffer dump can be committed to the main memory after synchronizing timestamp/dependency checking.

9.1.5 GPU Acceleration

Stream processors such as GPUs exhibit massively parallel processing capabilities. If exploited appropriately, they can be of immense help to speculative parallelism. Previous software TLS researches have proposed various approaches to utilize GPU's computation power, such as CPU-GPU collaboration, kernel-level speculation and value prediction, as was discussed in section 8.7. With the advent of integrated architectures such as Accelerated Processing Unit (APU) [1], where the data-sharing cost between CPU and GPU can be minimized, there are increasingly more opportunities to use heterogeneous computing [5] for the design and implementation of efficient and effective software-TLS. The third/fourth generation APU provides unified address space and fully coherent memory features of the Heterogeneous System Architecture (HSA) [6], on which CPU and GPU share the same address space and cached data, which can be programmed by OpenCL 2.0 shared virtual memory (SVM) [74]. We can target the MUTLS system to this architecture and automatically generate speculative OpenCL/SPIR-V [93] kernels and study how well the GPU can accelerate the speculative parallelization system.

Extending MUTLS for new languages/architectures

Since MUTLS adopts a language and architecture independent approach to software-TLS, it is fairly straightforward to extend MUTLS to support additional source languages and/or target architectures.

Add Front-End Language for MUTLS

To add a new front-end programming language for MUTLS, the front-end compiler needs to use the same version of LLVM as MUTLS does. The following are necessary steps to enable effective parallelization using MUTLS.

(1) The front-end compiler compiles the source program and generates the LLVM-IR modules.

(2) The front-end compiler adds fork/join/barrier point annotations in the generated LLVM-IR using the LLVM intrinsic function `llvm.forkjoinpoints` as was discussed in section 3.1. This step may be ignored if the front-end compiler decides to use adaptive fork-heuristics of Chapter 6 for automatic parallelization.

(3) The front-end compiler registers the address spaces of global and heap variables with MUTLS runtime library API function calls `MUTLSLIB_register_static`, `MUTLSLIB_register_malloc` and `MUTLSLIB_register_free`, as was discussed in sections 3.6.1 and 7.2.2. This avoids unnecessary rollbacks if the speculative threads access global and heap variables. This step may be ignored if the front-end language/-compiler uses standard C library functions such as malloc/free for memory allocation/deallocation.

(4) The driver of the front-end compiler invokes the "`-speculation`" LLVM pass (MUTLS speculator pass described in Chapter 3) to transform the LLVM-IR. If using adaptive fork-heuristics and/or blockization, then before invoking the "`-speculation`" pass, the front-end compiler invokes the "`-mutls-auto=$MUTLS_AUTO_TYPE -forkpointmarker -blockizeloop`" passes, where `MUTLS_AUTO_TYPE` may be `blockize` or `noblockize`, meaning using adaptive fork-heuristics to automatically insert loop fork point annotation with or without blockization, respectively. `MUTLS_AUTO_TYPE` may also be `none` to denote using the LLVM-IR `llvm.forkjoinpoints` annotation instead of adaptive fork-heuristics.

(5) The driver of the front-end compiler links the transformed LLVM-IR with the MUTLS runtime library.

Add Back-End Architecture for MUTLS

Since MUTLS speculator pass does not use architecture dependent instructions such as inline assembly, it is not difficult for MUTLS to support other LLVM back-end architectures. It is just necessary that the compiler and runtime library of MUTLS are re-compiled for the target architecture.

One subtlety is the implementation of the MUTLS runtime library as well as the application binary interface (ABI) between the MUTLS transformed LLVM-IR and the library, as different architectures may have different implementations of C data types such as pointer bit-length. Nevertheless, we expect the effort to be small to add other back-end support for MUTLS.

MUTLS runtime library API functions

```
void MUTLSLIB_arch_check(const char* arch)
void MUTLSLIB_set_main_stack(int* stackptr)
void MUTLSLIB_register_static(int id, char* addr, size_t size, char* name)
void MUTLSLIB_register_malloc(int id, char* addr, size_t size, int rank)
void MUTLSLIB_register_realloc(int id, char* prev_addr, char* addr, size_t size, int
     rank)
void MUTLSLIB_register_free(char* addr, int rank)
void MUTLSLIB_set_readonly_globalvar_id(int id)
void MUTLSLIB_set_readonly_heapvar_id(int id)
int MUTLSLIB_get_task(int model, int hint, int hintid, int self_ptid, int selfrank)
void MUTLSLIB_speculate(int model, char* stackptr, int ptid, int hintid,
    MUTLSLIB_stub_func_type f, int taskrank, int selfrank)
void MUTLSLIB_initialize_speculative_thread(int rank)
int MUTLSLIB_get_speculated_rank(char* stackptr, int ptid, int rank)
char MUTLSLIB_synchronize(char* stackptr, int ptid, int *child_rank, int* bb)
void MUTLSLIB_sync_parent(char* stackptr, int ptid, int rank)
void MUTLSLIB_default_sync_point(char* stackptr, int rank)
char MUTLSLIB_sync_entry(char* stackptr, int* bb, int* child_rank, int rank)
void MUTLSLIB_prefork(int hintid, int rank)
void MUTLSLIB_end_speculative_region(int hintid, int rank)
void MUTLSLIB_intrin(int type, int arg1, int arg2, int arg3, int rank)
bool MUTLSLIB_barrier_point(int self_ptid, int ptid, int rank)
void MUTLSLIB_rollback_point(int no, int rank)
void MUTLSLIB_invalidate_point(int no, int rank)
void MUTLSLIB_unsupported_point(int no, int rank)
bool MUTLSLIB_check_point(int rank)
bool MUTLSLIB_check_joining(int rank)
void MUTLSLIB_reclaim_joined_frame(int rank)
void MUTLSLIB_set_self_stackptr(char* addr, int rank, int selfrank)
void MUTLSLIB_set_child_stackptr(char* addr, int rank)
void MUTLSLIB_return_frame(char* stackptr, int bb, int self_ptid, int rank)
void MUTLSLIB_start_validation_frame(int rank, int selfrank)
```

```
void MUTLSLIB_end_validation_frame(int rank, int selfrank)
void MUTLSLIB_commit_stub_frame(int retno, int rank)
void MUTLSLIB_llvm_memset(uint8_t* addr, uint8_t val, int size, int align, int rank)
void MUTLSLIB_llvm_memcpy(uint8_t* dest, uint8_t* src, int size, int align, int rank
    )
void MUTLSLIB_llvm_memmove(uint8_t* dest, uint8_t* src, int size, int align, int
    rank)
void MUTLSLIB_llvm_ptrtoint(char* ptr, int bb, int rank)
void MUTLSLIB_check_not_div_zero(uint64_t v, int bb, int rank)
void MUTLSLIB_c_assert_fail(char* assertion, char* file, int line, char *func, int
    rank)
void MUTLSLIB_Py_INCREF(NumbaRefCount* p, int rank)
void MUTLSLIB_Py_DECREF(NumbaRefCount* p, int rank)
void MUTLSLIB_Py_XINCREF(NumbaRefCount* p, int rank)
void MUTLSLIB_Py_XDECREF(NumbaRefCount* p, int rank)
void MUTLSLIB_set_local_int8(int index, uint8_t value, int rank, int selfrank)
void MUTLSLIB_set_local_int16(int index, uint16_t value, int rank, int selfrank)
void MUTLSLIB_set_local_int32(int index, uint32_t value, int rank, int selfrank)
void MUTLSLIB_set_local_int64(int index, uint64_t value, int rank, int selfrank)
void MUTLSLIB_set_local_float(int index, float value, int rank, int selfrank)
void MUTLSLIB_set_local_double(int index, double value, int rank, int selfrank)
void MUTLSLIB_set_local_longdouble(int index, long double value, int rank, int
    selfrank)
void MUTLSLIB_set_local_ptr(int index, int* value, int rank, int selfrank)
uint8_t MUTLSLIB_get_fork_local_int8(int index, int rank, int selfrank)
uint16_t MUTLSLIB_get_fork_local_int16(int index, int rank, int selfrank)
uint32_t MUTLSLIB_get_fork_local_int32(int index, int rank, int selfrank)
uint64_t MUTLSLIB_get_fork_local_int64(int index, int rank, int selfrank)
float MUTLSLIB_get_fork_local_float(int index, int rank, int selfrank)
double MUTLSLIB_get_fork_local_double(int index, int rank, int selfrank)
long double MUTLSLIB_get_fork_local_longdouble(int index, int rank, int selfrank)
int* MUTLSLIB_get_fork_local_ptr(int index, int rank, int selfrank)
uint8_t MUTLSLIB_get_join_local_int8(int index, int rank, int selfrank)
uint16_t MUTLSLIB_get_join_local_int16(int index, int rank, int selfrank)
uint32_t MUTLSLIB_get_join_local_int32(int index, int rank, int selfrank)
```

```
uint64_t MUTLSLIB_get_join_local_int64(int index, int rank, int selfrank)
float MUTLSLIB_get_join_local_float(int index, int rank, int selfrank)
double MUTLSLIB_get_join_local_double(int index, int rank, int selfrank)
long double MUTLSLIB_get_join_local_longdouble(int index, int rank, int selfrank)
int* MUTLSLIB_get_join_local_ptr(int index, int rank, int selfrank)
void MUTLSLIB_speculate_local_int8(int index, uint8_t value, int predictor_type, int
     rank, int selfrank)
void MUTLSLIB_speculate_local_int16(int index, uint16_t value, int predictor_type,
    int rank, int selfrank)
void MUTLSLIB_speculate_local_int32(int index, uint32_t value, int predictor_type,
    int rank, int selfrank)
void MUTLSLIB_speculate_local_int64(int index, uint64_t value, int predictor_type,
    int rank, int selfrank)
void MUTLSLIB_speculate_local_float(int index, float value, int predictor_type, int
    rank, int selfrank)
void MUTLSLIB_speculate_local_double(int index, double value, int predictor_type,
    int rank, int selfrank)
void MUTLSLIB_speculate_local_longdouble(int index, long double value, int
    predictor_type, int rank, int selfrank)
void MUTLSLIB_speculate_local_ptr(int index, int* value, int predictor_type, int
    rank, int selfrank)
void MUTLSLIB_validate_local_int8(int index, uint8_t value, int rank, int selfrank)
void MUTLSLIB_validate_local_int16(int index, uint16_t value, int rank, int selfrank
    )
void MUTLSLIB_validate_local_int32(int index, uint32_t value, int rank, int selfrank
    )
void MUTLSLIB_validate_local_int64(int index, uint64_t value, int rank, int selfrank
    )
void MUTLSLIB_validate_local_float(int index, float value, int rank, int selfrank)
void MUTLSLIB_validate_local_double(int index, double value, int rank, int selfrank)
void MUTLSLIB_validate_local_longdouble(int index, long double value, int rank, int
    selfrank)
void MUTLSLIB_validate_local_ptr(int index, int* value, int rank, int selfrank)
void MUTLSLIB_save_alloca(int self_ptid, int id, char* data, int size, int rank, int
     selfrank)
```

```
void MUTLSLIB_get_alloca_data(int id, char* data, int size, int rank, int selfrank)
void MUTLSLIB_map_alloca_ptr(char** ptr, int rank, int selfrank)
void MUTLSLIB_map_alloca_array_ptr(char** ptr, int n, int elementSize, int rank, int
     selfrank)
uint8_t MUTLSLIB_load_int8(uint8_t* addr, AddressScope scope, int rank)
uint16_t MUTLSLIB_load_int16(uint16_t* addr, AddressScope scope, int rank)
uint32_t MUTLSLIB_load_int32(uint32_t* addr, AddressScope scope, int rank)
uint64_t MUTLSLIB_load_int64(uint64_t* addr, AddressScope scope, int rank)
int* MUTLSLIB_load_ptr(int** addr, AddressScope scope, int rank)
float MUTLSLIB_load_float(float* addr, AddressScope scope, int rank)
double MUTLSLIB_load_double(double* addr, AddressScope scope, int rank)
uint64_t MUTLSLIB_load_mem(uint8_t* addr, int size, AddressScope scope, int rank)
void MUTLSLIB_load_nonsp_int8(uint8_t* addr, AddressScope scope)
void MUTLSLIB_load_nonsp_int16(uint16_t* addr, AddressScope scope)
void MUTLSLIB_load_nonsp_int32(uint32_t* addr, AddressScope scope)
void MUTLSLIB_load_nonsp_int64(uint64_t* addr, AddressScope scope)
void MUTLSLIB_load_nonsp_ptr(int** addr, AddressScope scope)
void MUTLSLIB_load_nonsp_float(float* addr, AddressScope scope)
void MUTLSLIB_load_nonsp_double(double* addr, AddressScope scope)
void MUTLSLIB_load_nonsp_mem(uint8_t* addr, int size, AddressScope scope)
void MUTLSLIB_store_int8(uint8_t* addr, uint8_t data, AddressScope scope, int rank)
void MUTLSLIB_store_int16(uint16_t* addr, uint16_t data, AddressScope scope, int
    rank)
void MUTLSLIB_store_int32(uint32_t* addr, uint32_t data, AddressScope scope, int
    rank)
void MUTLSLIB_store_int64(uint64_t* addr, uint64_t data, AddressScope scope, int
    rank)
void MUTLSLIB_store_ptr(int** addr, int* data, AddressScope scope, int rank)
void MUTLSLIB_store_float(float* addr, float data, AddressScope scope, int rank)
void MUTLSLIB_store_double(double* addr, double data, AddressScope scope, int rank)
void MUTLSLIB_store_mem(uint8_t* addr, uint64_t data, int size, AddressScope scope,
    int rank)
void MUTLSLIB_store_nonsp_int8(uint8_t* addr, uint8_t data, AddressScope scope)
void MUTLSLIB_store_nonsp_int16(uint16_t* addr, uint16_t data, AddressScope scope)
void MUTLSLIB_store_nonsp_int32(uint32_t* addr, uint32_t data, AddressScope scope)
```

```
void MUTLSLIB_store_nonsp_int64(uint64_t* addr, uint64_t data, AddressScope scope)
void MUTLSLIB_store_nonsp_ptr(int** addr, int* data, AddressScope scope)
void MUTLSLIB_store_nonsp_float(float* addr, float data, AddressScope scope)
void MUTLSLIB_store_nonsp_double(double* addr, double data, AddressScope scope)
void MUTLSLIB_store_nonsp_mem(uint8_t* addr, uint64_t data, int size, AddressScope
    scope)
void MUTLSLIB_check_nonsp_load()
void MUTLSLIB_check_nonsp_store()
void MUTLSLIB_check_nonsp_load_store()
void MUTLSLIB_set_func_name(const char* name, int rank)
void MUTLSLIB_set_pointer_name(int* addr, const char* name, const char* fname, int
    rank)
void MUTLSLIB_set_hintid_num(int n)
void MUTLSLIB_set_nest_hintid(int idc, int idp)
void MUTLSLIB_disable_hintid(int id)
void MUTLSLIB_compiler_timing_begin(int id, int rank)
void MUTLSLIB_compiler_timing_end(int id, int rank)
void MUTLSLIB_bb_trace(int bb, char* func_name, char* bb_name, int rank)
```

References

[1] Accelerated processing unit. http://en.wikipedia.org/wiki/Accelerated_processing_unit.

[2] C-ray raytracing benchmark results. http://www.futuretech.blinkenlights.nl/c-ray.html.

[3] Deadlock. https://en.wikipedia.org/wiki/Deadlock.

[4] DragonEgg. http://dragonegg.llvm.org.

[5] Heterogeneous computing. https://en.wikipedia.org/wiki/Heterogeneous_computing.

[6] Heterogeneous system architecture. https://en.wikipedia.org/wiki/Heterogeneous_System_Architecture.

[7] LLVM. http://llvm.org.

[8] LLVMPY. http://www.llvmpy.org.

[9] Load-link/store-conditional. https://en.wikipedia.org/wiki/Load-link/store-conditional.

[10] Lock (computer science). https://en.wikipedia.org/wiki/Lock_(computer_science).

[11] Mandelbrot_openmp - mandelbrot image using openmp. http://people.sc.fsu.edu/~jburkardt/f_src/mandelbrot_openmp/mandelbrot_openmp.html.

[12] Mclab. http://www.sable.mcgill.ca/mclab.

[13] Md_openmp - molecular dynamics using openmp. http://people.sc.fsu.edu/~jburkardt/f_src/md_openmp/md_openmp.html.

[14] Memory coherence. https://en.wikipedia.org/wiki/Memory_coherence.

[15] Non-blocking algorithm. https://en.wikipedia.org/wiki/Non-blocking_algorithm.

[16] Non-uniform memory access. https://en.wikipedia.org/wiki/Non-uniform_memory_access.

[17] Numba. http://numba.pydata.org.

[18] OpenCL. https://en.wikipedia.org/wiki/OpenCL.

[19] OpenMP. https://en.wikipedia.org/wiki/OpenMP.

[20] Optimistic concurrency control. https://en.wikipedia.org/wiki/Optimistic_concurrency_control.

[21] Parallel programming model. https://en.wikipedia.org/wiki/Parallel_programming_model.

[22] Polybench: the polyhedral benchmark suite. http://www.cs.ucla.edu/~pouchet/software/polybench.

[23] Posix threads. https://en.wikipedia.org/wiki/POSIX_Threads.

[24] Priority inversion. https://en.wikipedia.org/wiki/Priority_inversion.

[25] Race condition. https://en.wikipedia.org/wiki/Race_condition.

[26] Scimark 2.0. http://math.nist.gov/scimark2.

[27] smallpt: Global illumination in 99 lines of c++. http://www.kevinbeason.com/smallpt.

[28] Dbt86: A dynamic binary translation research framework for the cmp era. *PESPMA'09*, 2009.

[29] Martin Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP'09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 185–196, 2009.

[30] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, and Bratin Saha. Compiler and runtime support for efficient software transactional memory. In *PLDI'06: Proceedings of the 27th ACM SIGPLAN Conference on*

Programming Language Design and Implementation, volume 41, pages 26–37. ACM, 2006.

[31] José L. Aguilar and Kahlil Campero. An explicit parallelism study based on thread-level speculation. *CLEI Electronic Journal*, 17(2), 2014.

[32] Ivo Anjo and Joao Cachopo. Improving continuation-powered method-level speculation for jvm applications. In *Algorithms and Architectures for Parallel Processing*, pages 153–165. Springer, 2013.

[33] Ivo Anjo and Joao Cachopo. A software-based method-level speculation framework for the java platform. In *LCPC'12: Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing*, pages 205–219. Springer, 2013.

[34] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In *SPAA'10: Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 253–262. ACM, 2010.

[35] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP'10: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 45, pages 105–114. ACM, 2010.

[36] Joao Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. Unifying thread-level speculation and transactional memory. In *Middleware'12: Proceedings of the 13th International Middleware Conference*, pages 187–207. Springer-Verlag New York, Inc., 2012.

[37] Arnamoy Bhattacharyya. Using combined profiling to decide when thread level speculation is profitable. In *PACT'12: Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 483–484, New York, NY, USA, 2012. ACM.

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

[39] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[40] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[41] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *ISCA'13: Proceedings of the 40th Annual International Symposium on Computer Architecture*, volume 41, pages 225–236. ACM, jun 2013.

[42] Zhen Cao. MUTLS (mixed model universal software thread-level speculation). `http://www.sable.mcgill.ca/~zcao7/mutls`, 2013.

[43] Zhen Cao and Clark Verbrugge. Language and architecture independent software thread-level speculation. In *LCPC'12: Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing*, pages 270–272, 2012.

[44] Zhen Cao and Clark Verbrugge. Adaptive fork-heuristics for software thread-level speculation. In *PPAM'13: 10th International Conference on Parallel Processing and Applied Mathematics*, pages 523–533, 2013.

[45] Zhen Cao and Clark Verbrugge. Mixed model universal software thread-level speculation. In *ICPP'13: Proceedings of the 42nd International Conference on Parallel Processing*, pages 651–660, 2013.

[46] Zhen Cao and Clark Verbrugge. Reducing memory buffering overhead in software thread-level speculation. In *CGO 2015 ACM Student Research Competition*, 2015.

[47] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Hakan Zeffer, and Marc Tremblay. Simultaneous speculative threading: a novel pipeline architecture implemented in sun's rock processor. In *ISCA'09: Proceedings of the 36th annual international symposium on Computer architecture*, volume 37, pages 484–495. ACM, 2009.

[48] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC'09: Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, pages 44–54, 2009.

[49] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IISWC'10: Proceedings of the 2010 IEEE International Symposium on Workload Characterization*, pages 1–11, December 2010.

[50] Michael K. Chen and Kunle Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT'98: Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques*, pages 176–184, October 1998.

[51] Michael K. Chen and Kunle Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA'03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 434–446, June 2003.

[52] Marcelo Cintra and Diego R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*, pages 13–24, June 2003.

[53] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10)*, 2010.

[54] Sebastien Dabdoub and Stephen Tu. Supporting intel transactional synchronization extensions in qemu. `http://people.csail.mit.edu/stephentu/papers/tsx.pdf`.

[55] Francis H. Dang, Hao Yu, and Lawrence Rauchwerger. The r-lrpd test: Speculative parallelization of partially parallel loops. In *IPDPS'02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pages 318–327. IEEE Computer Society, 2002.

[56] Gregory Diamos and Sudhakar Yalamanchili. Speculative execution on multi-gpu systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[57] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006.

[58] Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *CGO'07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 21–33. IEEE, 2007.

[59] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *PACT'14: Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 3–14. ACM, 2014.

[60] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–234, June 2007.

[61] Jialin Dou and Marcelo Cintra. A compiler cost model for speculative parallelization. *ACM Transactions on Architecture and Code Optimization*, 4(2):12, 2007. "pages" is really "article number".

[62] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *PLDI'09: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 44(6):155–165, 2009.

[63] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *PLDI'04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 71–81, June 2004.

[64] Frank Ch Eigler. Mudflap: Pointer use checking for c/c++. In *GCC Developers Summit*. Citeseer, 2003.

[65] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP'08: Proceedings*

of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 237–246. ACM, 2008.

[66] Sérgio Miguel Fernandes and Joao Cachopo. Lock-free and scalable multi-version software transactional memory. In *PPoPP'11: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, volume 46, pages 179–188. ACM, 2011.

[67] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[68] Emily Fortuna, Owen Anderson, Luis Ceze, and Susan Eggers. A limit study of javascript parallelism. In *2010 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2010.

[69] Álvaro García-Yágüez, Diego R. Llanos, and Arturo González-Escribano. Robust thread-level speculation. In *HiPC'11: Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, pages 1–11. IEEE, 2011.

[70] Álvaro García-Yágüez, Diego R. Llanos, and Arturo González-Escribano. Squashing alternatives for software-based speculative parallelization. *IEEE Transactions on Computers*, pages 247–279, May 2013.

[71] María Jesús Garzarán, Milos Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(3):247–279, September 2005.

[72] Jos Gonzlez and Antonio Gonzlez. The potential of data value speculation to boost ilp. In *ICS'98: Proceedings of the 12th international conference on Supercomputing*, pages 21–28, 1998.

[73] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *CGO'10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 101–110. ACM, 2010.

[74] Khronos OpenCL Working Group. The opencl specification. `https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf`.

[75] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *PODC'05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 258–264. ACM, 2005.

[76] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP'08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, February 2008.

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

[78] Guodong Han, Chenggang Zhang, King Tin Lam, and Cho-Li Wang. Java with auto-parallelization on graphics coprocessing architecture. In *ICPP'13: Proceedings of the 42nd International Conference on Parallel Processing*, pages 504–509. IEEE, 2013.

[79] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP'05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

[80] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI'06: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementatio*, pages 14–25, 2006.

[81] John L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.

[82] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC'03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[83] Ben Hertzberg and Kunle Olukotun. Runtime automatic speculative parallelization. In *CGO'11: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 64–73. IEEE Computer Society, 2011.

[84] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ISCA'09: Proceedings of the 36th annual international symposium on Computer architecture*, volume 37, pages 152–163. ACM, 2009.

[85] William N Scherer III and Michael L Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, pages 70–79, 2004.

[86] William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *PODC'05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005.

[87] Nikolas Ioannou, Jeremy Singer, Salman Khan, Polychronis Xekalakis, Paraskevas Yiapanis, Adam Pocock, Gavin Brown, Mikel Lujan, Ian Watson, and Marcelo Cintra. Toward a more accurate understanding of the limits of the tls execution paradigm. In *IISWC'10: Proceedings of the 2010 IEEE International Symposium on Workload Characterization*, pages 1–12. IEEE, 2010.

[88] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for ibm system z. In *MICRO'12: Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 25–36. IEEE, 2012.

[89] Yunlian Jiang and Xipeng Shen. Adaptive software speculation for enhancing the cost-efficiency of behavior-oriented parallelization. In *ICPP'08: Proceedings of the 37th International Conference on Parallel Processing*, pages 270–278. IEEE, 2008.

[90] Chuanle Ke, Lei Liu, Chao Zhang, Tongxin Bai, Bryan Jacobs, and Chen Ding. Safe parallel programming using dynamic dependence hints. In *OOPSLA'11: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, volume 46, pages 243–258. ACM, 2011.

[91] Arun Kejariwal, Xinmin Tian, Milind Girkar, Wei Li, Sergey Kozhukhov, Utpal Banerjee, Alexander Nicolau, Alexander V. Veidenbaum, and Constantine D. Polychronopoulos. Tight analysis of the performance potential of thread speculation using SPEC CPU2006. In *PPoPP'07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 215–225, March 2007.

[92] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast track: A software system for speculative program optimization. In *CGO'09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 157–168. IEEE Computer Society, 2009.

[93] John Kessenich. An introduction to spir-v. `https://www.khronos.org/registry/spir-v/papers/WhitePaper.pdf`.

[94] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222, June 2007.

[95] Manaschai Kunaseth, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta, David F. Richards, and James N. Glosli. Performance characteristics of hardware transactional memory for molecular dynamics application on bluegene/q: Toward efficient multithreading strategies for large-scale scientific applications. In *IPDPS'13: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, pages 1326–1335, 2013.

[96] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pages 75–88, Washington, DC, USA, 2004. IEEE Computer Society.

[97] Viktor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE'14: Proceedings of the IEEE 30th International Conference on Data Engineering*, pages 580–591. IEEE, 2014.

[98] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *MICRO 29: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, December 1996.

[99] Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. Speculative execution on gpu: an exploratory study. In *ICPP'10: Proceedings of the 39th International Conference on Parallel Processing*, pages 453–461. IEEE, 2010.

[100] Shaoshan Liu, Christine Eisenbeis, and Jean-Luc Gaudiot. Value prediction and speculative execution on gpu. *International Journal of Parallel Programming*, 39(5):533–552, 2011.

[101] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS compiler that exploits program structure. In *PPoPP'06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, March 2006.

[102] Yuan Liu, Hong An, Bo Liang, and Li Wang. An online profile guided optimization approach for speculative parallel threading. In *Proceedings of the 12th Asia-Pacific conference on Advances in Computer Systems Architecture (ACSAC '07)*, pages 28–39, 2007.

[103] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, Antonia Zhai, Nikhil Mungre, and Ankit Tarkas. Dynamic performance tuning for speculative threads. *ISCA'09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 462–473, 2009.

[104] Marc Lupon, Grigorios Magklis, and Antonio Gonzalez. A dynamically adaptable hardware transactional memory. *MICRO'43: Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 27–38, 2010.

[105] Carlos Madriles, Carlos García-Quiñones, Javier Sanchez, Pedro Marcuello, Adriana Gonzalez, Dean M Tullsen, Hong Wang, and John P Shen. Mitosis: A speculative multithreaded processor based on precomputation slices. *IEEE Transactions on Parallel and Distributed Systems*, 19(7):914–925, 2008.

[106] Carlos Madriles, Pedro Lopez, Josep M. Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raul Martinez, and Antonio Gonzalez. Anaphase: A fine-grain thread decomposition scheme for speculative multithreading. In *PACT'09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 15–25, 2009.

[107] Carlos Madriles, Pedro Lopez, Josep M. Codina, Enric Gibert, Fernando Latorre, Alejandro Martinez, Raul Martinez, and Antonio Gonzalez. Boosting single-thread performance in multi-core systems through fine-grain multithreading. In *ISCA'09: Proceedings of the 36th annual international symposium on Computer architecture*, volume 37, pages 474–483, June 2009.

[108] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *DISC'05: Proceedings of the 19th International Conference on Distributed Computing*, pages 354–368. Springer, 2005.

[109] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[110] Virendra Jayant Marathe and Mark Moir. Toward high performance nonblocking software transactional memory. In *PPoPP'08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 227–236. ACM, 2008.

[111] Jan Kasper Martinsen and Håkan Grahn. Thread-level speculation for web applications. In *Second Swedish Workshop on Multi-Core Computing (MCC-09)*, pages 80–88. SICS, 2009.

[112] Jan Kasper Martinsen and Håkan Grahn. An alternative optimization technique for javascript engines. In *Third Swedish Workshop on Multi-Core Computing (MCC-10)*, pages 155–160. Chalmers University of Technology, 2010.

[113] Jan Kasper Martinsen and Håkan Grahn. Thread-level speculation as an optimization technique in web applications – initial results. In *2011 6th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 83–86. IEEE, 2011.

[114] Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg. Preliminary results of combining thread-level speculation and just-in-time compilation in googlefs v8. In *Sixth Swedish Workshop on Multi-Core Computing (MCC-13)*, page 37, 2013.

[115] Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg. *Using Speculation to Enhance JavaScript Performance in Web Applications*, volume 17. 2013.

[116] Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg. The effects of parameter tuning in software thread-level speculation in javascript engines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):46, 2015.

[117] Alexander Matveev and Nir Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *SPAA'13: Proceedings of the 25th annual ACM symposium on Parallelism in algorithms and architectures*, pages 11–22. ACM, 2013.

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

[119] Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. Dynamic parallelization of javascript applications using an ultra-lightweight speculation mechanism. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 87–98. IEEE, 2011.

[120] Jaikrishnan Menon, Marc de Kruijf, and Karthikeyan Sankaralingam. igpu: exception support and speculative execution on gpus. In *ISCA'12: Proceedings of the 39th Annual International Symposium on Computer Architecture*, volume 40, pages 72–83. IEEE Computer Society, 2012.

[121] James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch. Crom: Faster web browsing using speculative execution. In *NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation*, page 9. USENIX Association, 2010.

[122] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC'08:*

Proceedings of the 2008 IEEE International Symposium on Workload Characterization, pages 35–46. IEEE, 2008.

[123] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPoPP'07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78. ACM, 2007.

[124] Cosmin E. Oancea and Alan Mycroft. Software thread-level speculation: an optimistic library implementation. In *IWMSE'08: Proceedings of the 1st International Workshop on Multicore Software Engineering*, pages 23–32, May 2008.

[125] Cosmin E. Oancea, Alan Mycroft, and Tim Harris. A lightweight in-place implementation for software thread-level speculation. In *SPAA'09: Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, pages 223–232, August 2009.

[126] Rei Odaira and Takuya Nakaike. Thread-level speculation on off-the-shelf hardware transactional memory. In *IISWC'14: Proceedings of the 2014 IEEE International Symposium on Workload Characterization*, pages 212–221. IEEE, 2014.

[127] Marek Olszewski, Jeremy Cutler, and J Gregory Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375. IEEE Computer Society, 2007.

[128] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *ACM Sigplan Notices*, 31(9):2–11, 1996.

[129] Venkatesan Packirisamy, Antonia Zhai, Wei-Chung Hsu, Pen-Chung Yew, and Tin-Fook Ngai. Exploring speculative parallelism in spec2006. In *ISPASS'09: Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 77–88. IEEE, 2009.

[130] Mathias Payer and Thomas R. Gross. Performance evaluation of adaptivity in software transactional memory. In *ISPASS'11: Proceedings of the 2011 IEEE*

International Symposium on Performance Analysis of Systems and Software, pages 165–174. IEEE, 2011.

[131] Christopher J.F. Pickett. Software speculative multithreading for java. In *OOPSLA'07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 929–930. ACM, 2007.

[132] Christopher J.F. Pickett and Clark Verbrugge. SableSpMT: a software framework for analysing speculative multithreading in Java. In *PASTE'05: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*, September 2005.

[133] Christopher J.F. Pickett and Clark Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *LCPC'05: Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, volume 4339, pages 304–318, 2005.

[134] Christopher J.F. Pickett, Clark Verbrugge, and Allan Kielstra. libspmt: A library for speculative multithreading. Technical Report Sable Technical Report No. 2007-1, McGill University, 2007.

[135] Christopher John Francis Pickett. *Software Method Level Speculation for Java*. PhD thesis, School of Computer Science, McGill University, April 2012.

[136] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *PLDI'11: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, 2011.

[137] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *PPoPP'03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, 2003.

[138] Manohar K. Prabhu and Kunle Olukotun. Exposing speculative thread parallelism in spec2000. In *PPoPP'05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 142–152, June 2005.

[139] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: A language extension for implicit parallel programming. In *PLDI'11: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 1–11, June 2011.

[140] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 269–279, June 2005.

[141] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing conflicting transactions in an stm. In *PPoPP'09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, volume 44, pages 163–172. ACM, 2009.

[142] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS'10: Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, pages 65–76, March 2010.

[143] Easwaran Raman, Neil Vachharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *CGO'08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 175–184, April 2008.

[144] Lawrence Rauchwerger and David Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming Language Design and Implementation*, pages 218–232, 1995.

[145] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In *ICS'05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 179–188, June 2005.

[146] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC'06: Proceedings of the 20th international conference on Distributed Computing*, pages 284–298. Springer, 2006.

[147] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *SPAA'07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 221–228. ACM, 2007.

[148] Peter Rundberg and Per Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *JILP: Journal of Instruction-Level Parallelism*, 3:1–28, October 2001.

[149] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP'06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197. ACM, 2006.

[150] Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. Paragon: collaborative speculative loop execution on gpu and cpu. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 64–73. ACM, 2012.

[151] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *MICRO'30: Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 248–258, December 1997.

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

[153] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA'95: Proceedings of the 22nd annual international symposium on Computer architecture*, volume 23, pages 414–425. ACM, 1995.

[154] Michael F Spear. Lightweight, robust adaptivity for software transactional memory. In *SPAA'10: Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 273–283. ACM, 2010.

[155] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP'09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, volume 44, pages 141–150. ACM, 2009.

[156] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *PODC'07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 338–339. ACM, 2007.

[157] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC'06: Proceedings of the 20th international conference on Distributed Computing*, pages 179–193. Springer, 2006.

[158] Michael F Spear, Maged M Michael, and Christoph von Praun. Ringstm: scalable transactions with a single atomic instruction. In *SPAA'08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 275–284. ACM, 2008.

[159] Michael F. Spear, Michael Silverman, Luke Dalessandro, and Maged M. Michael. Scalable techniques for transparent privatization in software transactional memory. In *ICPP'08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 67–74. IEEE, 2008.

[160] Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and exploiting inevitability in software transactional memory. In *ICPP'08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 59–66. IEEE, 2008.

[161] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*, pages 1–12, June 2000.

[162] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems (TOCS)*, 23(3):253–300, August 2005.

[163] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving value communication for thread-level speculation. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 65–75, 2002.

[164] Enqiang Sun and David Kaeli. Aggressive value prediction on a gpu. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2011 23rd International Symposium on*, pages 9–16. IEEE, 2011.

[165] Andrew Tan, Yue Xing, and Pratch Piyawongwisal. Concurrent speculative optimization for just-in-time compiler. `http://www.andrew.cmu.edu/user/aktan/compilers/final_report.pdf`.

[166] Chen Tian, Min Feng, and Rajiv Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *PLDI'10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, volume 45, pages 62–73, 2010.

[167] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or discard execution model for speculative parallelization on multicores. In *MICRO'08: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–341. IEEE Computer Society, 2008.

[168] Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta. Enhanced speculative parallelization via incremental recovery. In *PPoPP'11: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, volume 46, pages 189–200, 2011.

[169] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 295–305, June 2011.

[170] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 79–89, March 2007.

[171] Amy Wang, Matthew Gaudet, Peng Wu, Jose Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. In *PACT'12:*

Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pages 127–136, 2012.

[172] Shengyue Wang, Xiaoru Dai, Kiran S. Yellajyosula, Antonia Zhai, and Pen-Chung Yew. Loop selection for thread-level speculation. In *Proceedings of the 18th international conference on Languages and Compilers for Parallel Computing (LCPC)*, pages 289–303, 2005.

[173] Fredrik Warg and Per Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *PACT'01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230, September 2001.

[174] Fredrik Warg and Per Stenström. Improving speculative thread-level parallelism through module run-length prediction. In *IPDPS'03: Proceedings of the 17th International Parallel and Distributed Processing Symposium*, pages 8–pp. IEEE, 2003.

[175] Fredrik Warg and Per Stenström. Reducing misspeculation overhead for module-level speculative execution. In *Proceedings of the 2nd conference on Computing frontiers (CF '05)*, pages 289–298, May 2005.

[176] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for Java. In *OOPSLA'05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 439–453, October 2005.

[177] John Whaley and Christos Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *ICPP'05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 147–156, June 2005.

[178] Fan Xu, Li Shen, Zhiying Wang, Hui Guo, Bo Su, and Wei Chen. Heuspec: A software speculation parallel model. In *ICPP'13: Proceedings of the 42nd International Conference on Parallel Processing*, pages 621–630, 2013.

[179] Paraskevas Yiapanis, Demian Rosas-Ham, Gavin Brown, and Mikel Lujan. Optimizing software runtime systems for speculative parallelization. In *ACM Transactions on Architecture and Code Optimization (TACO)*, volume 9, January 2013.

[180] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intelr transactional synchronization extensions for high-performance computing. In *SC'13: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[181] Chenggang Zhang, Guodong Han, and Cho-Li Wang. Gpu-tls: An efficient runtime for speculative loop parallelization on gpus. In *CCGrid'13: Proceedings of the 13th IEEE/ACM International Symposium onCluster, Cloud and Grid Computing*, pages 120–127. IEEE, 2013.

[182] Lihang Zhao, Woojin Choi, and Jeff Draper. Sel-tm: Selective eager-lazy management for improved concurrency in transactional memory. In *IPDPS'12: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 95–106, 2012.

[183] David Zier and Ben Lee. Performance evaluation of dynamic speculative multithreading with the cascadia architecture. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 21(1):47–59, 2010.