PARALLEL PROGRAMMING USING THREAD-LEVEL SPECULATION

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL

ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Manohar Karkal Prabhu

December 2005

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle A. Olukotun

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz

Approved for the University Committee on Graduate Studies.

Abstract

As the performance increases of single-threaded processors diminish, consumer desktop processors are moving toward multi-core designs. Thread-level speculation (TLS) increases the space of applications that can benefit from these designs. With TLS, a sequential application is divided into fairly independent tasks that are speculatively executed in parallel, while the hardware dynamically enforces data dependencies to provide the appearance of sequential execution. This thesis demonstrates that support for TLS greatly eases the task of manual parallel programming. Because TLS provides a sequential programming interface to parallel hardware, it enables the programmer to focus only on issues of performance, rather than correctness.

The dissertation starts by demonstrating the parallelization of a microbenchmark to introduce a number of techniques for manual TLS parallelization. Several of the advanced techniques leverage programmer expertise to surpass the capabilities of current advanced, automated parallelizers; the research presented here can provide guidance for the future development of such tools. Following this, the use of these techniques to parallelize seven of the SPEC CPU2000 applications is described. TLS parallelization yielded an average 120% speedup on four floating point applications and 70% speedup on three integer applications, while requiring only approximately 80 programmer hours and 150 lines of non-template code per application. These strong parallel performance results generated with relatively modest programmer effort support the inclusion of TLS in future chip multiprocessor designs.

For each application parallelized, a detailed description is provided of how and where parallelism was located, the impediments to extracting it using TLS, and the code transformations that were required to overcome these impediments. The results on these applications demonstrate that using advanced manual techniques is essential to effectively parallelize integer benchmarks. This leads to a discussion of common hindrances to TLS parallelization, and a subsequent description of methods of programming that help expose the parallelism in applications to TLS systems. These programming guidelines can help uniprocessor programmers create applications that can be easily ported to future TLS systems and yield good performance. In closing, the dissertation reviews the many advantages of manual TLS parallel programming and specifies potential future research areas.

Acknowledgments

I would like to thank the many people who have provided me the support and encouragement to complete this dissertation and my Ph.D. There are so many family members, friends and associates that it is hard to know where to stop, but I do know where to start the list. I would like to thank my daughter Vaishali, first and foremost. While many would argue that students with children take longer to complete, no distraction could be quite so grand as dear little Vaishali. Whether she was a baby sitting and cooing on my lap while I was debugging code, or was instead demanding I take time off to pay her some attention as she grew older, she has always made working from home the best way to get the job done. She is my other advisor, my live-in advisor (and is much more demanding, I might add!).

I would like to thank so many of my family members, as well. My mom, my brother and my two sisters have provided much of the inspiration that has led me down this path. Since moving to sunny California, there have been a host of other relatives who have provided fabulous fun and cheer, including Anita, Vivek, Farzaneh, Pandu, Mala and a bunch more.

And many a friend has brightened my way through grad school, as have so many workmates. I am always indebted to “Uncle Lance,” who has earned his title not by being here at Stanford for more years than me, but from the non-stop fun and action he provides Vaishali on her every visit to the lab. His presence in the great halls of Gates will be sorely missed by many. And likewise, it has been fun hanging out with Murali and Tara, the Hydra gang of old and new and the Future/Alumni Professors of Manufacturing. I am indebted to the various people who have worked behind the scenes to make my education possible, Darlene Hadding, Charlie Orgish and Marianne Marx, to name just a few. And, out in the "real" world, I owe a heap of gratitude to my many managers and work associates at HP, including most of all Ray, Bob, Emmanuelle and Steve.

But of course, the list would be incomplete without expressing my profound appreciation for the many advisors who have helped steer my path through to the light at the end of the tunnel. I thank Christos Kozyrakis and Mark Horowitz for the interest they have taken in my research and in providing me feedback on my conference presentations, my orals and this dissertation. I wish to thank Rick Reis and a variety of other professors at Stanford and beyond, who have motivated me to pursue a career in academia. But most of all, I wish to thank Kunle Olukotun, my doctoral advisor, for being a continuing and unwavering support through the many twists and turns of the Ph.D. Kunle has not only been an advisor, but also a friend, and I feel fortunate to have done my doctorate under an advisor whom I hold in such high regard.

Table of Contents

List of Tables

List of Figures

1 Introduction and Background

Workloads run on modern computer systems exhibit a large degree of inherent parallelism, which means that significant portions of the workloads can be executed concurrently. Computers can greatly improve their computational performance by exploiting inherent parallelism, which often exists at many different levels. At one extreme, instruction-level parallelism (ILP) occurs between the individual computer instructions which were intended to be executed sequentially. At the other extreme, process-level parallelism allows multi-tasking operating systems to execute separate, possibly unrelated instruction streams on the same computer hardware at different times, thereby tolerating latency and allowing more efficient use of a computer system's resources. Between these extremes lie various forms of thread-level parallelism (TLP). TLP allows a single program to be split into threads, which are sequential portions of the program that would have executed at different times. These threads can then be executed in parallel.

To extract parallelism, special-purpose hardware and software are generally used together. Extracting ILP requires tracking fairly modest amounts of speculative state and a limited number of interactions between instructions, and to be efficient this must be done very quickly. Hence, extracting ILP is done via the extensive use of special-purpose hardware, and the implementation details are purposely hidden from the programmer. On the other hand, extracting high-level parallelism from applications requires support from the operating system for thread creation and scheduling. If the threads share data, generally the programmer must be highly involved in enabling and extracting this parallelism, because of the extensive communication and synchronization that must be properly handled. This parallel programming places much of the burden of extracting high-level parallelism on the programmer.

Meanwhile, extracting parallelism at the intermediate level of threads is even more difficult. In fact, parallelism amongst fine-grained threads (on the order of 100 to 10,000 dynamic instructions in length) has rarely been exploited at all. Tracking the required state has been too difficult for the available hardware, while software-based solutions have required too much overhead, programmer effort or compiler intelligence. However, due to steady increases in hardware complexity, it has now become possible to extract fine-grain, thread-level parallelism with hardware in ways that are largely transparent to the programmer. The research discussed here presents the performance gains that can be expected by utilizing fine-grained thread-level speculation, a specific approach to extracting this level of parallelism. This research also demonstrates the low programmer effort required to conduct this form of parallelization for common applications and the different approach to programming taken by a programmer using TLS.

To open the discussion of the current research, this chapter reviews the evolution of and current issues surrounding both hardware and software for extracting parallelism. I discuss how hardware and software issues have impacted the prevalence and ease of parallel programming. First, I consider advances in hardware, and how challenges to continuing along Moore's Law have made single-chip multiprocessors an attractive design alternative to uniprocessors. Next, I shift the focus to software, and consider how parallel programming is typically conducted and what are its limitations. Hardware support for parallel programming can be used to combat these limitations, and I describe current and prior research that has been done in this area. This sets the stage for a discussion of what important research issues remain unaddressed, and how this thesis contributes knowledge in these areas. Following this, I describe the methodology utilized to conduct this research. Finally, I close with an overview of the rest of the thesis by providing a description of each of the remaining chapters.

1.1 Evolution of Hardware

It has been almost 40 years since Gordon Moore first described in 1965 what has become known as Moore's Law, the observation of the exponential rate of increase in transistor count on a single die. This exponential transistor count increase is expected to continue throughout the next decade [13]. The largest computer processors already contain approximately half a billion transistors on a single die and the next generation Itanium processors are expected to incorporate 1.7 billion transistors [28]. This has led to a situation in which transistors are almost free, designs are constrained less by transistor count than by design complexity and power constraints, and computer architects are under constant pressure to innovate new ways to utilize these transistors.

1.1.1. Increasing difficulties of hardware design

Until recently, the vast majority of this innovation had been directed toward single-threaded computational performance. Processor designs have evolved from simple, multiple-cycle-per-instruction, microcode-based cores to complex, pipelined, superscalar, out-of-order, speculative cores with extensive on-board caches. While processor performance has dramatically improved, implementing these extra layers of complexity has grown exponentially more difficult.

Not only are such complex designs difficult to design, but they are even more difficult to verify and validate. Verification is the process of checking that the state machine design actually implements the functionality desired. Validation checks that the implementation in silicon is a correct realization of the state machine previously verified. Process variations, manufacturing defects and unforeseen on-chip interactions can cause a test chip to fail validation. Verification and validation of processor designs is greatly confounded by the non-deterministic and irreproducible behavior of these complex processors. A design flaw may only be observed under certain voltage and temperature scenarios, or worse yet, only in a rare situation generated by a particular run state and a particular timing of communications received from off-chip sources. Generating and observing such flaws with real silicon are difficult due to the inability to easily control or observe the state of internal logic nodes with no direct connection to off-chip test equipment. For these reasons, validation and verification efforts grow at a faster rate than the rest of the design efforts. In fact, the verification and validation efforts for complex processors have become larger than the design efforts, making them primary concerns in the design process.

Additional complexities have frustrated and slowed the increase in uniprocessor computational performance. In recent years, power consumption has taken an especially high priority, shifting design effort away from peak compute performance. At high clock frequencies, signals can only propagate across a small portion of a processor's core, and only limited amounts of logic can be performed within a pipeline stage. This results in further complexity and an increased number of pipeline registers, which further exacerbates the power consumption problem. These issues combined have effectively limited the further scaling of clock frequencies. And, providing adequate off-chip bandwidth poses an increasing challenge, because higher performance usually requires either higher bandwidth off-chip or larger or more efficient caching on-chip. These increasing obstacles to improving uniprocessor compute performance are compounded by the fact that the improvements designed often provide disappointing performance on several important workloads, especially commercial server workloads such as OLTP (on-line transaction processing) and DSS (decision support systems) [2]. This is often due to the poor memory locality of the workloads.

1.1.2. Methods for reducing hardware design complexity

Many of these problems can be addressed by intellectual property (IP) reuse, which is the use multiple times of standard logic designs, from small cells of transistors to whole processors and interfaces. By sacrificing peak efficiency, IP reuse allows the designer to incorporate additional transistors into the design at the same rate that they become available, thereby enabling the designer to take full advantage of the benefits of Moore's Law. Standard cell design has been in use for a long time, but what is becoming more common now is the use of IP blocks as large as processors and chip interfaces. This reduces design time by providing another, higher level of abstraction to the design process, and the clearly defined interactions between IP blocks can reduce verification and validation efforts.

Of special interest to the research described here is the integration of multiple processor cores onto a single die. This can be seen as an extension of IP reuse to the chip level from the board or system level, which has been the traditional level at which to create a multiprocessor. Chip multiprocessors (CMPs) seem an inevitable consequence of increased integration and miniaturization, as they address many of the design concerns described above via IP reuse. CMPs provide excellent performance for multitasking operating systems for consumers, due to the many concurrent processes these operating systems typically support, such as virus checking, firewalls, data encryption and multimedia. Furthermore, CMPs can provide good performance on some applications that are difficult for uniprocessors [2]. The CMP design should also lower costs and increase reliability over multiple-chip designs. As a result, every major microprocessor manufacturer has announced plans to manufacture single chip multiprocessors in the near future [8][16][17][19][23]. A critical element in each of these designs is determining how the processors will work with each other, their coherence, consistency, communication and synchronization mechanisms. As part of that decision, it is important to understand the ways in which a programmer would parallelize programs for such platforms. Therefore, in the next section, I discuss parallel programming in general, and what makes it so challenging.

1.2 Design of Parallel Software

While increasingly sophisticated processor cores can provide improved performance, multiple cores can work together to provide further parallel speedup. Typically this is done by adapting the source code, in some situations automatically, to have multiple fairly independent components that execute in parallel and communicate with each other. The necessity of rewriting the software and the types of hardware that can best support this effort depend on the granularity of the parallelism present in the application, i.e. the average number of sequential, dynamic instructions of each portion of execution that is to be run in parallel with other portions of execution.

1.2.1. Granularities of parallelism

A survey of benchmark applications conducted by D. W. Wall [34] indicated that almost all applications, including the integer benchmarks, have enormous amounts of inherent parallelism, if problems arising over register reuse, control flow dependences and memory aliasing can be overcome. By addressing these problems with even a modest hardware-software approach (the “fair” model), factors of speedup in the range of 2 to 4 could commonly be achieved. A study by Lam and Wilson [18] also indicated the large amount of parallelism that exists within applications. In this study, the importance of mitigating control flow dependences was illustrated, and speculation was shown to be a key enabler of this.

While these large amounts of parallelism do exist, they occur at many different levels of granularity and are often impractical to extract. Instruction-level parallelism (ILP) can be very effectively addressed with advanced compilers and special-purpose hardware for superscalarity, instruction reordering, register renaming, branch prediction and predicated instructions. But, extracting higher-level parallelism requires more involvement from the programmer and the operating system. While ILP only requires tracking hazards within a small window of instructions, higher-level parallelism necessitates tracking data dependences between much larger sequences of instructions. If these instruction streams contain branches and pointer indirection, tracking dependences between them becomes exponentially more complex with increasing instruction stream length. This complicates the task of automating higher-level parallelization. Ideally, a sequential application could be divided into multiple fairly independent instruction streams and dependences could be resolved for each instruction in the stream prior to its execution. But, realistically, the complexity of the dependences and the many ways in which the sequential instruction stream could be divided into parallel streams makes this too difficult to automate for many, if not most, common applications.

1.2.2. Ability to automate parallelization

Whether an application is amenable to automated higher-level parallelization depends on a variety of characteristics. Applications with very regular accesses to data and few unpredictable branches in the instruction stream can easily be parallelized. This is especially true if the write accesses to the data progress in a pattern that either already is or can easily be made to be distributed over several independent memory locations. This is the case for many scientific and floating point applications, where a large, dense matrix is fully populated with values in an orderly sweep through the matrix, and where each value generated does not require input from any other value generated in the matrix in the current sweep. Another characteristic that can ease parallelization is for an application to have several fairly independent phases or tasks, such as in database applications, where searching for data can often be conducted in parallel with processing data retrieved in the previous search. Historically, the high-level parallelism in these types of applications has been heavily exploited, often automatically.

However, other applications are much more difficult to parallelize, even with programmer assistance. These have irregular control flow and tend to conduct most of their execution on individual variables, sparse matrices, heaps, stacks or other data structures that are accessed in a complex or pseudo-random pattern. The heavy use of a few key memory locations or unpredictable data access patterns make these applications difficult to parallelize, due to either unpredictable dependences or due to frequent, unavoidable dependences (even if predictable). Integer applications are a common example of this, and are characterized by instruction streams with frequent branches that are difficult to predict and by execution of relatively little computation in a regular way on large, dense matrices. Some integer applications are for all practical purposes unparallelizable, except for their initialization and completion routines and a few other portions of the application.

Between these two extremes lie a variety of applications with moderate amounts of higher-level parallelism that can be extracted. Interest in parallelizing them has steadily been growing over time, as computer processors and operating systems for the home consumer market have become capable of multitasking and multithreading, and as memory and disk delays versus increasing processor speeds have made multithreading for latency tolerance more attractive. Parallelizing these applications usually requires extensive programmer involvement and efforts and cannot be automated. But, with this investment, some of these applications can exhibit good parallel speedups, as is the case for multimedia applications.

1.2.3. Challenges to extracting parallelism

A common reason that applications have substantial inherent parallelism that cannot be automatically extracted is that when the applications were written, they were designed in a manner that obscured the inherent parallelism in the computation. This is typically through a choice of data structures and algorithms that are not amenable to parallelization. For example, the use of a stack can complicate parallelization, because each portion of the application that could run separately will attempt to access the same stack memory and stack pointer, even though the data each portion stores in the stack memory is often private and entirely independent of the data stored by other portions of the application. This is an example of an artificial dependence introduced by the programmer that is not inherent in the computation required for the application. It is due to the choice of a data structure with low TLP, the stack.

Parallelism can also be obscured by the programmer's choice of an algorithm with low TLP. A common example of this is the use of recursion, rather than iteration. Iterative loops often exhibit control flow and data parallelism between each iteration. This means that each iteration, except the last, will occur regardless of the computation in the previous iteration, and that the data used for computation will be fairly independent between iterations. But, recursion obscures parallelism, because the control flow to each portion of the recursion and the data it uses depends on the results from computation in the previous portion of the recursion, and even the latter portions of the recursion (except in the case of tail recursion, which is bad programming style). This renders the control flow of recursion difficult to predict and the data between portions dependent, thereby making recursion difficult to parallelize because of control flow and data dependences.

Applications with high inherent TLP that have had the parallelism obscured were often written targeting a uniprocessor platform. Frequent reuse of variables and the use of stack-based algorithms can yield good data locality and small working sets, thereby improving uniprocessor performance. The use of recursion can simplify programming. But pursuing the best strategy for uniprocessor programming can cause the programmer to obscure the inherent TLP in a program. For these applications, if parallelization is ever expected, a programmer may need to keep a multiprocessor target in mind while designing the program, even if this results in slightly less efficient or more complex code.

1.2.4. Approaches to parallelization of applications

Given the complexity of extracting higher-level parallelism, it must often be done manually. For applications with inherent TLP, there are two main models for parallel programming, shared-memory parallel programming and programming with a message-passing interface (MPI). In shared-memory programming, each process shares the same address space and reads and writes the same variables. In the message-passing model, processes each have a separate address space with private variables. Communication is conducted via messages that are explicitly sent and received between the processes.

Shared-memory programming is generally agreed to be the more natural programming model, because communication is done implicitly without much specification from the programmer, and all data is always available to all processes. On the other hand, MPI requires the programmer to plan in advance what data must be communicated between the processes, and generally the data structures must be split between the processes, so that each process has just the data it requires to compute its portion of the algorithm. This requirement of partitioning the data and explicitly sharing values makes MPI the more difficult model for programming, although it can facilitate subsequent performance optimization, since it renders the inter-process communication patterns more evident. While shared-memory programming both allows and necessitates frequent communication, often of small sets of data, MPI encourages the aggregation of reads and writes into larger messages. This allows for lower communication overheads for MPI implementations of applications with regular, large data accesses. However, the overheads of copying data into and out of the send and receive buffers limits performance on irregular or shorter-length messages, giving shared-memory programming an advantage for these applications. Given the greater ease of shared-memory programming and the substantially similar performance that can be achieved, the bulk of commercial parallel programming is done for shared memory.

While parallel programming can be done manually, for applications with exposed inherent parallelism, much of the work can be done automatically. This is achieved with parallelizing compilers. SUIF [10] and Polaris [3] can automatically parallelize for shared memory and are very well known. Automatic parallelizing compilers for MPI also exist, such as the MPI backend for SUIF described by Kwon, Han and Kim [15] and the commercial software development package PGI by Portland Group Compiler Technology for OpenMP, an MPI application program interface. Parallelizing compilers extract parallelism well for very regular floating-point, scientific applications, especially those written in Fortran. While some parallelism can be extracted by static analyses of the application, a more powerful method involves using a profiler in combination with the compiler. The profiler characterizes the dynamic behavior of the application under typical workloads. This allows the compiler to make more optimal decisions about the way in which to parallelize to achieve good parallel performance, taking account of dynamic issues such as load balancing and the overheads from forking and joining threads or processes.

While taking account of dynamic effects, parallelizing with profiling nevertheless relies upon a compiler constructing a parallel formulation of the application prior to the program executing on a possibly new dataset. This new dataset could cause different execution characteristics than the dataset under which the application was parallelized. Phrased differently, this is still a static approach to parallelization. While static parallelization performs very well for simple applications and moderately well on some less regular applications, the range of applications that can be addressed can be extended by allowing the use of dynamic parallelization.

In dynamic parallelization, the assignment of instructions to processors is formulated at the time of execution. This can be as simple as creating a queue of tasks. Alternatively, it can be as complex as speculative execution. Speculation on the final results of execution to be conducted by one processor can allow another processor to conduct sequentially later execution out of sequential order. This can be done purely using software support for dynamic parallelization without the addition of hardware. Some applications that are difficult to parallelize statically but are regular in their data access patterns are amenable to this software-controlled dynamic parallelization at the thread level [30]. But, for more complex applications, dynamically extracting TLP requires hardware support to reduce the overheads associated with preventing or correcting dependences. This method of extracting TLP is hardware-supported thread-level speculation (TLS), which is the focus of the research presented in this dissertation. A range of methods for implementing and using this approach to dynamic parallelization have been developed. Research on these is described in the next section, in the context of describing the contributions of my dissertation to related research. As this dissertation centers on TLS, a detailed description of its theory, implementation, strengths and limitations is provided in Chapter 2: Thread-Level Speculation (TLS).

1.3 Contributions of This Dissertation over Related Research

Having provided a background on the parallelism in applications and the conventional hardware and software approaches to extracting it, in this section I discuss my contributions to the field of the research described in this dissertation and place it into context with related research conducted by others.

Research on application analysis, profiling and parallelization [1][3][14][21][24] and on speculation [6][26][27][32][37] is underway at various universities. Several projects share my focus on general purpose applications. However, those projects primarily focus on developing methods to automatically parallelize applications. Instead, this dissertation investigates two other areas. First, I use manual parallelization, so that design modifications and programmer expertise can be utilized to yield higher parallel performance. In this manner, I approach the upper bounds of the parallelism that can be extracted from these applications when using a loosely-coupled chip multiprocessor with a fairly simple TLS mechanism. Second, I use my experience with manually parallelizing these applications to explain precisely where in these important benchmarks TLP exists, how to extract it and how to overcome the obstacles to parallelization of each application.

For my research, I used a loosely coupled (L2-cache-connected) CMP that supports only fairly simple TLS. Similar to my results, the Wisconsin Multiscalar team also achieves excellent speedups on general purpose applications, including integer applications [25][37]. However, Multiscalar allows register-to-register communication between the processors at the cost of more complex and high-speed hardware. So, their research explores a different hardware/software design space, generally utilizing finer-granularity threads. Research by the CMU STAMPede team [32][33][35] and at the University of Illinois at Urbana-Champaign [6][7][36] explores different design points with less closely coupled processors, more similar to the TLS CMP used as the test platform in this dissertation. This research supplements related research by providing a rough indication of an upper bound on performance for similar TLS systems exploiting TLP of the same granularity. My research into the capabilities of thread-level speculation is not constrained by the ability to automate the techniques I have utilized. This allows better performance on some applications that can be automatically parallelized, as well as speedups on applications that are not amenable to automatic parallelization. Together with an explanation of where and how I extracted this parallelism, this research can help direct future related research to see if some of these new parallelization techniques I have developed can be automated and also if improvements can be made to automated thread selection algorithms. It also helps define the performance limitations of TLS that are inherent to the TLS system and to the applications, without confounding this with limitations in the abilities of an automatic parallelizer.

Relevant research done by Rauchwerger, Padua and Amato considered software-based schemes of speculation and parallelization [30], while later work in conjunction with Zhang and Torrellas utilized hardware support, as well [36]. Other studies [6][33] have focused on achieving highly scalable parallelization. These studies differ from the current study in that they either focus on using software only, or on using hardware support specific to the code transformation applied, e.g. hardware for conducting parallel reductions or for achieving scalable speedups using non-blocking commits of speculative state. Also, much of the previous research has centered on scientific, floating-point-intensive Fortran applications [6][30][33][36], while the research described here considers both floating point and integer programs that are all written in C.

Finally, substantial work on exploiting value prediction and dynamic synchronization has been conducted in [5][7][9][20][25][32]. I incorporate the benefits of these studies where possible and extend upon them. For example, the earlier value prediction studies explore only predictions of values that do not change or are in a simple stride. In this study, I explore predictions of values that evolve in a more complex manner.

Following this different approach of utilizing manual parallelization without regard to the ability to automate the techniques employed allows for several other unique contributions. This dissertation provides insight into the effort required of a programmer to port commercial applications manually to a TLS platform and optimize their parallel performance. This research clearly demonstrates the simplicity of manual parallelization with TLS versus without it, and provides evidence of the performance advantages of the optimistic, dynamically determined synchronization of TLS versus the pessimistic, statically-determined conventional synchronization, which uses locks or barriers, for example. This information on programmer effort required and parallel performance achieved enables a rough cost-benefit analysis to be done versus other methods of parallelization, including conventional (non-TLS) manual parallelization and conventional or TLS automatic parallelization.

This dissertation also provides an indication of what phases of the programming task consume the most effort, i.e. analysis of the application, initial parallelization, performance tuning or debugging. These data provide further indications of the ease of TLS manual parallel programming compared to non-TLS manual parallelization. Information on the performance achieved and the source code transformation techniques utilized to expose more TLP to the TLS system provides an indication of the most useful techniques for extracting TLP, which can help direct developers toward the best places to invest future research efforts. The in-depth analysis of the parallelism in the SPEC2000 benchmarks and the limitations of a practical TLS system to extract the inherent TLP provides useful information about where TLS exists in several of the applications in the highly-utilized SPEC2000 suite, and what are the biggest barriers to extracting this parallelism, even with direct programmer involvement and expertise beyond the capabilities of current automated parallelization tools. Understanding these limitations to the extraction of TLP with a TLS system allows this dissertation to contribute knowledge about ways in which uniprocessor programmers often obscure the inherent TLP that exists within applications. This dissertation contributes a set of guidelines that uniprocessor programmers could easily follow that would allow for more rapid and high performance porting of uniprocessor applications to TLS platforms in the future.

1.4 Methodology

To conduct research on parallelizing applications with thread-level speculation (TLS), it was necessary to choose a set of applications, a hardware platform and parameters to vary. The rationale behind those selections will be explained in this section. I first discuss the objectives and the approach taken to this selection of applications. This leads next to a brief discussion of the applications selected and the reasons for their selection. Finally, I discuss the strategy utilized for measuring the research results, as this required a more sophisticated sampling strategy than has commonly been used in related research. This was due to the size and nature of the applications and their data sets.

1.4.1. Objectives and approach

The methodology decisions represent a tradeoff between using realistic applications and simulations on the one hand, and rendering the research tractable on the other. The approach adopted was to first conduct research on very small applications with a simplified hardware simulator, in order to get insight into the characteristics of the applications and the ability to extract thread-level parallelism from them using TLS. Then, more complex analyses and applications were studied to understand how non-idealities such as memory delay and speculation overheads impact the ability to extract thread-level parallelism (TLP) from real-world applications.

The selection of applications was of primary importance. TLP can be extracted in a variety of ways utilizing a number of different hardware platforms. I was interested to show how TLS assists in extracting TLP, without having the conclusions be strongly dependent upon the use of a particular hardware platform. As a result, the hardware platform chosen was a fairly simple system with multiple cores loosely coupled through a shared L2 cache. Specialized hardware was included in the caches and in the form of a coprocessor for each processor core. But the cores themselves were left unmodified from their uniprocessor versions, and much of the implementation, complexity and overhead of speculation was delegated instead to software handlers. The specific system chosen was the Hydra chip multiprocessor. This was selected as a matter of convenience, based upon past work conducted within our research group and the availability of a well-supported simulator infrastructure.

The applications, on the other hand, were not selected for convenience. They were selected for one of two reasons. Either the selected application was very simple and could be used to clearly illustrate an issue about TLS programming, or the application was representative of a large class of applications and was chosen to provide insight into the performance and parallelization issues that could be expected.

The objectives of the research presented in this dissertation are to understand where TLP exists in general purpose applications, and how this can be extracted using manual TLS programming. I aim to show the parallel performance that can be expected from using TLS, as well as to understand its limitations in extracting TLP. To gauge the difficulty of parallelizing uniprocessor applications for a TLS system, I present information both on the programmer effort required and on the difficulties that exist for this process to be conducted automatically. Finally, from my experiences conducting parallelization with TLS, I make suggestions for ways in which uniprocessor programs can be designed to ease the process of parallelizing with TLS later and to enhance the performance that can be easily obtained.

1.4.2. Selection of applications

To achieve these goals, I first designed and coded a very efficient uniprocessor microbenchmark that could specifically demonstrate the capabilities of TLS parallelization versus traditional, non-TLS parallelization. I then ported this program from a uniprocessor to a TLS system. This microbenchmark uses a heap sort algorithm to count the number of occurrences of each unique word in a passage of text. Parallelizing heap sort is fairly challenging due to both frequent, predictable dependences and also infrequent, unpredictable dependences. This microbenchmark is used to provide insight into how TLS programming can provide both better performance and also require less programmer effort. This insight provides the framework in which the larger, more complex complete applications can be discussed.

Following this, I sought applications that are widely considered representative of general-purpose applications. I chose SPEC CPU2000 (also called SPEC2000), which is a well-established, well-understood and generally accepted set of benchmarks for floating-point and integer applications. The SPEC2000 suite comprises 14 floating point and 12 integer applications. These applications are representative of a wide range of general-purpose applications, and exemplify different types of computation ranging from bus depot scheduling to molecular interaction modeling to compiling to file compressing.

Within SPEC2000, applications that are difficult to automatically parallelize but amenable to manual parallelization with TLS were selected. In particular, the most difficult floating point applications to parallelize and the most easily parallelizable integer applications were selected for this research. More details on all the SPEC2000 benchmarks and the ones selected is provided in Section 4.1.

1.4.3. Measurement and sampling strategy

Each of the SPEC2000 benchmarks comes with two or three standard input data sets (and execution parameters): a test data set, a training data set and a reference (full-size) data set. While the test and training data sets can enable smaller execution times, the behavior of the applications under these input sets is substantially different from the behavior on the full-size reference data sets. Likewise, the dynamic behavior of the applications changes substantially throughout the full execution with the reference data set. However, a significant problem with measuring the performance of hypothetical processor architectures using simulation is the length of execution that can be simulated practically. While executing the full application against the reference data set takes too long, using the test or the training data sets can give misleading results. Likewise, measuring the parallel performance of a system under test on just a small section of the application run with the reference data set must be done carefully. Ideally, segments of execution should be chosen at several spots throughout the full execution, and together, or better yet singly, these segments should have a pattern of execution that closely approximates the behavior of the entire execution. For example, the percentage of time spent in each subroutine should be approximately the same for the sampled and the full execution, as well as the ratio of nested iterations to their enclosing iterations being similar.

Initially, I parallelized the floating point applications. Because they are very parallel in nature, they were an easy starting point to gain experience with effective ways to manually parallelize using TLS and to understand the limitations of TLS that prevent linear speedups. However, since Fortran applications tend to both be extremely parallelizable and even automatically parallelizable, I decided not to study those. Many publications already exist on parallelizing those automatically, including publications on generating scalable speedups approaching linear speedups. These Fortran applications tend to be well-suited to automated program analysis. Hence, from SPEC CFP2000, I parallelized only the four applications written in C.

Following this, I focused on the integer applications. Here I selected applications on the opposite criterion, ease of parallelization. This is because many of the applications in SPEC CINT2000 are very difficult to parallelize, even manually. I specifically avoided the most difficult integer applications, which have a large source code size, execution time distributed between too many loops, loop dependences that are very difficult to understand or algorithms that appear to have a high probability of not conducting much parallel work. These applications appeared likely to be too difficult to parallelize effectively and could be truly sequential in character. However, other integer applications in SPEC2000 can be fairly easily parallelized. I chose those applications to understand how that could be done and to explore how the availability of a TLS system facilitates that.

The SPEC2000 applications typically have few or no source code comments. While a great effort could have been made to understand the algorithms within each application, I specifically avoided doing this. I wanted to gain experience with TLS programming in the way that many programmers must parallelize legacy applications. With limited knowledge and insight into the data structures and algorithms selected by the original designer of the application, I wanted to see if it was possible to achieve substantial parallel speedups with little effort and little chance of introducing parallel programming bugs like data races or deadlock. This would provide a better indication of the usefulness of hardware support for TLS for parallelizing commercial legacy applications.

1.5 Layout of dissertation

This chapter has provided the motivation for studying manual programming with TLS. It has covered the context from which hardware support for TLS arises and the current research into its development and usage. I have framed the contributions of this dissertation in that context and discussed the general methodology employed.

The next chapter provides the necessary detailed information about TLS to understand the research conducted with it. It discusses how TLS works from a theoretical perspective, which leads to a description of practical TLS systems with limitations due to difficulties of implementation. With this knowledge it is possible to understand the common causes of performance losses that arise in TLS systems.

In Chapter 3, the process of conducting manual parallel programming with TLS is described. This process is then applied to two, tiny example applications in order to clearly show the way in which this process of manual parallelization is conducted and also to explain the most common techniques utilized to transform programs to expose more of their inherent TLP to the TLS system for extraction. These examples also help demonstrate the relative ease of manual parallel programming with TLS versus conventional manual parallelization without TLS.

Chapter 4 describes how this process and these transformation techniques were used to parallelize whole applications within the SPEC2000 benchmarks suite. For each benchmark parallelized, the locations at which substantial TLP was found are listed, along with the techniques utilized to increase the amount of that parallelism that could be extracted. Limitations in the ability to extract further parallelism are discussed and performance data are provided indicating the relative usefulness of the each of the additional transformations conducted upon the original source code.

Finally Chapter 5 discusses higher level observations made across all applications of the usefulness of TLS for manual parallel programming. Specifically, the programming effort required and the distribution of that effort across the typical phases of programming are discussed for the parallelization of the selected SPEC2000 benchmarks. Statistics on the parallel performance across the different applications are analyzed to explain patterns in the performance results. There is also a discussion of common limitations to the extraction of TLP from applications, and guidelines for programming the original uniprocessor applications that would help to diminish these limitations to TLP extraction during subsequent parallelization. In closing, these findings are summarized and indications are provided of future research in the area that would be valuable to conduct.

2 Thread-Level Speculation (TLS)

In the previous chapter, the reasons for conducting parallel programming with hardware support for TLS were discussed. This chapter provides the necessary information on TLS to understand the research conducted and the results generated. It begins with a description of the theoretical basis of TLS, then progresses to describe how such a system can be practically implemented and the details of the particular implementation that was simulated for the current research. Finally, the performance limitations of TLS and its practical implementations are described, to set the stage for the following chapters which use this system to parallelize various benchmarks.

2.1 Ideal TLS systems

TLS facilitates the extraction of TLP by allowing multiple processors to work in parallel, while preserving the appearance of single-processor, sequential execution. In TLS, a sequential instruction stream is cut in multiple places, forming threads of contiguous instructions. The first thread that would be executed if these were to be executed sequentially is termed the head thread. In TLS, while the head thread is being executed, the other threads are executed speculatively. When the head thread completes, the least speculative thread is termed the new head thread, and typically execution of a new, more speculative thread is started to replace execution of the head thread just completed.

The selection of places at which to divide the instruction stream into threads can be conducted automatically or manually by the programmer. This selection is done in a way that produces threads that are likely to have few true (read-after-write) dependences between each other. The most common way in which to form threads is to split a sequential instruction stream at the beginning of each iteration of a loop, where the iterations are each of an appropriate instruction length (hundreds to thousands of instructions). The threads generated by doing this will often execute well in parallel, if the interactions between each iteration of the loop are fairly limited. An example of this is shown in Figure 2-1. To specify a selection of threads, the programmer or the automatic thread generator will generally mark the beginning and end of a section of sequential execution that can be parallelized on a TLS system. This is termed a speculative region. Additionally, the programmer will mark each separation point between the threads within this speculative region. For a loop parallelized with TLS, this corresponds to marking the start and the end of the loop as places to begin and end speculation (thereby defining a speculative region), and marking the beginning of each iteration within the loop as the point of separation between each thread.

The expectation is that the threads generated for TLS can execute substantially in parallel. When true dependences do occur between threads executing in parallel, the hope is that the read will occur in the more speculative thread after the write has been conducted by the less speculative thread upon which the read is dependent. This way, the correct value will be available so the true dependence can be correctly processed. This forwarding of data is shown in the data dependences labeled with the numeral *1* in Figure 2-1.

```
while (1)
{
    timeStep++;

    if (timeStep>1000 && carSpeed>50)
        {
            .
            .
            .
        }

    carSpeed += speedChange();
}
```

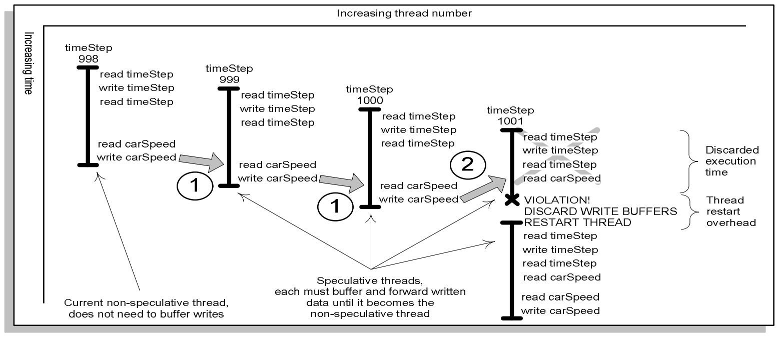


Figure 2-1: Thread-level speculation

However, unlike conventional (non-TLS) parallel processing, the availability of the correct value for the more speculative thread's read is not guaranteed. With conventional parallel processing, synchronization and data or algorithm partitioning prevent correctly parallelized (and debugged) programs from allowing reads to occur before the value has been generated by the prior write. With TLS, when an incorrect value is read due to its being processed before the write upon which it depends, a mechanism exists in the TLS system to automatically detect and correct this problem. This detection and correction of data is shown in the data dependence labeled with the numeral *2* in Figure 2-1.

Similarly, TLS systems must also detect and correct improperly processed anti-dependences, i.e. write-after-read and write-after-write dependences. All threads must ensure that the final values they write into every memory location are incorporated into the execution of all threads more speculative than themselves, while no values that are less speculative are incorporated. Correspondingly, speculative threads must make sure that the first exposed read of each memory location includes the final results of threads less speculative than themselves and none that are more speculative than themselves. (Exposed reads are those that occur within a speculative thread before any write to the same location within the same thread.)

One way to implement these two goals is to buffer all writes from speculative threads in a per-thread buffer. These writes are committed after each thread becomes the head thread, i.e. all threads commit their writes in sequential order. Also, the writes from all threads, including the head thread, are broadcast (even while speculative) to all threads that are more speculative, in order to inform them of the updates. At the same time, speculative threads note all exposed reads they have conducted while speculative and recompute all results dependent on values read that were subsequently updated (and broadcast to them) by less speculative threads. This update is termed a violation, which is the detection of a new value of a previously read value (for an exposed read) and the recomputation of results based upon that read. This is shown in the data dependence labeled with the numeral *2* in Figure 2-1.

By tracking dependence violations, TLS can guarantee correct execution of programs without requiring that the programmer understand all the data dependences in the application. The programmer (or an automatic speculative thread generator) can make assumptions about the best partitioning of the sequential execution into independent threads. If the choice of threads is conducted well, there will be a significant parallel speedup due to TLS. If the choice of threads is poor and there are frequent violations, there will be little or no speedup, and with practical TLS systems, there can even be a slowdown. But, these are purely performance issues. The application will never execute incorrectly due to incorrect assumptions about dependences and data races, unlike the situation with conventional parallel programming that renders it so difficult and error-prone. The TLS system enforces the correct ordering of computation required by the data dependences in the program under execution.

2.2 Practical implementations of TLS systems

For ease of design, TLS implementations do not adhere to the most efficient specification just described. For example, when a violation occurs, theoretically only results affected by the previously-read-and-now-updated value require recomputation. However, this minimum recomputation is difficult to implement, because it requires tracking which results are affected by each read value. The simpler but less efficient approach is to simply discard all results created by the violated thread (the speculative thread that conducted the read) and to restart execution of the entire thread. This ensures that all incorrect results based upon this read are discarded and all computation within the thread utilizes the updated value, which is read again. While this is simpler, it implies that the entire execution conducted by the speculative thread can be lost due to one violation which only affects a single value at the end of the thread. Research has been conducted into whether discarding all execution causes large performance losses and whether checkpointing mid-thread state can be useful to allow the loss of only a portion of a speculative thread's execution [26]. However, that research indicated that implementing mid-thread checkpointing for the same TLS implementation used for the current research introduced sizeable hardware overheads and little performance benefit for the applications considered in that study.

Another way in which implementation of TLS is simplified at the cost of performance is in the way updates are communicated between threads. For every exposed read by a speculative thread, a violation can occur. However, only the final value update (in sequential execution order) for each memory location out of all the value updates generated by all threads less speculative than this thread should be allowed to cause that violation. And, only if this final updated value is different from the one speculatively read by this thread should results be recomputed. If it has not, then no recomputation is necessary.

However, in the absence of perfect knowledge of when the final result before each exposed read has been generated, the much simpler strategy of broadcasting all writes by less speculative threads is utilized in most TLS implementations. Also, TLS implementations described in research publications generally assume that the updated value has changed, but it is possible to optimize for the special case in which the updated value is unchanged from its previous value [20]. Stores that do not change the value overwritten are termed silent stores. For simplicity of implementation, in the TLS implementation used for the current research no optimization is incorporated to take advantage of silent stores.

As I discuss below, violating on all stores, even if they are final or silent, can cause unnecessary violations. However, given the previous simplification of discarding all execution for a violation, violating on non-final updates can actually provide performance advantages. Since all execution prior to a violation must be completed again, assuming a violation due to a non-final update can provide an early restart for a thread that is likely to suffer a violation on the final update. Violating on a silent store, however, provides no benefit except by coincidental interaction with other violations.

An additional way in which TLS systems can be made simpler is by simplifying the way in which threads can be generated. In a simple implementation, when execution is started on a new speculative thread, it will always be the most speculative one being executed. In more complex TLS implementations, execution can be started on new speculative threads that are less speculative than some of the threads already under execution. The former implementation simplifies thread ordering; once started, threads always complete until speculation is completed; during the lifetime of each thread, the threads ordered before and after it always remain the same. For the more complex form of speculation, threads can be terminated before the end of speculation, and new speculative threads can be inserted between two threads already under speculative execution.

The more complex form allows threads which are not the most speculative thread to also fork off speculative threads. This can be useful when a thread executes a procedure call, by allowing the thread to fork off the less speculative procedure call while continuing execution on the more speculative instructions that should be executed after returning from the procedure call. The complex form of speculation similarly allows a thread to fork off multiple speculative threads when a loop is encountered within a single thread. This allows for re-entrant or multi-level speculation. This situation occurs when speculation is being conducted on an outer loop and each iteration forms a single thread. Then, if a single iteration-thread encounters a very long loop within itself, it forms speculative threads on this inner loop, i.e. to speculate at multiple loop levels. But, in order to do this, the newly generated threads may need to be inserted (ordered) before the other iteration-threads already in progress at the upper level. For our study, I chose non-reentrant, loop-only speculation, one of the simplest and lowest-overhead implementations of TLS. It allows sequential iterations of a loop to proceed in parallel and does not allow speculation on more than one loop simultaneously.

This is a brief overview of the relevant fundamentals of TLS and the implementation I chose for my research. More details on this form of TLS can be found in [12].

2.3 Performance limiters of TLS systems

On many applications, TLS can be used very effectively to extract TLP. However, on all but the simplest applications, some effort must be expended to eliminate or reduce a number of common situations that generate performance losses. I have broadly categorized these performance limiters as primary or secondary on the basis of their significance, i.e. how likely they are to be present, and how much they typically affect performance when present.

Most of these situations also severely limit performance for applications parallelized with conventional (non-TLS) methods. Others even represent fundamental limits to the ability to parallelize applications in any way. However, while they cause problems for almost any approach to parallelization, I highlight these here because the impact of many of them can be significantly alleviated in ways that are novel to the use of TLS. I will discuss the problems here, but leave discussing methods to alleviate these problems to future chapters. There, the solutions can be discussed along with representative applications illustrative of the success that can be had in addressing these limitations.

2.3.1. Primary performance limiters

(1) Relative positioning of stores and loads in successive iterations

The most important factor contributing to loss of parallel performance arises from the minimum critical path of dependences through an application. The minimum execution time required to compute a value in one thread that is read by a more speculative thread determines a lower bound on the execution time to complete the two threads, unless value prediction can be successfully employed. While this is usually complex to determine for all computed values with dependences, many programs have a few obvious dependence paths that are far longer than the minimum critical path length.

A simple example of this commonly arises with threads constructed from iterations of a loop with a loop induction variable. The induction variable is often tested at the start of the iteration to determine whether to execute the iteration. At the end of the iteration, this induction variable is usually incremented or updated. Because the next iteration initially tests this value, a true dependence is introduced between each iteration and its following iteration. The path to compute the value read at the top of each iteration requires completing the entire preceding iteration. If these iterations are made into separate threads, this early read and late update of the loop induction variable leads to almost certain dependence violations and virtually no parallel speedup. However, this is usually a simple problem to fix, by simply updating the loop induction variable early, right after it is first read. This reduces the critical path on that variable to its minimum and thereby increases the parallelism available. While this is a simple example, more complex examples will be discussed in future chapters. The common theme for all performance problems of this nature are that early reads of a value and late stores to the same memory location within a thread produce a long computation path that limit parallelism and exposes this thread and more speculative threads than it to a greater chance of experiencing a dependence violation.

This path of computation from exposed read to final update of the same memory location will hereafter be referred to as the violation window for this memory location within the thread. This violation window has the following special properties. It is only during and after this window that a violation can be suffered by this thread on this memory location (due to the exposed read). And, it is only during this window that a violation can be caused by this thread on this memory location (due to the writes). Each memory location associated with a dependence will in general have a different violation window. Violations can be guaranteed not to occur at all when, for every memory location with inter-thread dependences, no threads have overlapping violation windows and all the violation windows are ordered according to the speculative ordering of their threads. For example, assume a memory location exhibits an inter-thread memory dependence that could cause a violation. Then for every pair of threads for which there is a dependence on this variable the following must be true. The violation window of the sequentially first (less speculative) thread must complete execution before the violation window of the sequentially second (more speculative) thread begins execution. Non-overlapping sequentially-ordered violation windows for every variable with inter-thread dependences guarantees no violations can occur. Because these violation windows have the properties described above, minimizing their sizes and placing them at optimal locations within the execution path of the thread can have profound effects upon the violation rate of the parallelized application and the parallel performance that can be achieved. This can be done by delaying the first (exposed) read as long as possible and conducting the last write as early as possible for every shared variable.

(2) Variability of load and store timing between iterations

While long violation windows can create problems, even short windows can cause frequently violations if they occur with high inter-thread execution time variability. For example, if in one thread a ten-cycle violation window occurs hundreds of cycles of execution into the thread, while in the next thread the violation window for the same location occurs right at the start of execution of this next thread, a violation will very likely occur, because execution of the violation windows is unlikely to occur in sequential program order. More precisely, if there is high variability in the point at which the violation window occurs within each thread for a memory location with frequent dependences, it will usually lead to frequent violations and will severely impact performance.

Ideally, for each memory location with dependences, the following property for its violation windows will hold. For any thread (which I will label the reference thread for clarity), measure the number of cycles of execution from the start of execution to the end of the reference thread's violation window. Then, for each more speculative thread executing concurrently with the reference thread, measure the number of cycles of execution from the more speculative thread's start of execution to the beginning of that thread's violation window. Subtract each second measurement from the reference thread's measurement, and these differences will ideally be as small as possible or even negative, especially for the threads more speculative than the reference thread but ordered closest to the reference thread. The better this condition is met, the less the probability that violations will occur due to the execution of overlapping or sequentially unordered violation windows.

(3) Silent stores and temporally silent stores

A silent store occurs when an application writes a value into a memory location that is exactly the same as what was already stored there. Because the value is unchanged after the store, no recomputation need be done by more speculative threads that may have already read this value. This is commonly seen in many applications [20]. It can occur for boolean variables, for example, when a condition is tested that often yields the same result, such as whether the number of items left in a buffer being processed is zero yet or not. However, most TLS implementations do not detect whether a value being written is a silent store, and hence these silent stores cause violations when they do not need to do so.

While silent stores have been addressed in previous literature [20], I have seen no publications that discuss what I term temporally silent stores, and correspondingly no TLS architectures to handle this. Temporally silent stores occur when a value does change during stores, but eventually returns to the same value it had at a previous time. If a temporally silent store occurs within a single thread, none of the intervening writes need have been broadcast. In fact, if the final update of a value by a thread is a temporally silent store, then the first time that this value was written by the thread is the last time that that value need be updated to the other threads.

The performance problem caused by temporally silent stores is that the updates to the memory location from immediately after the first time this value was written up to and including the last update can cause violations in more speculative threads, even though none of these writes are necessary as the final value has already been written into the memory location. None of the writes following the first write that matches the final update need have been broadcast. Furthermore, if this final update was the same as the initial value read by the thread (the exposed read), then no writes to this memory location need have been broadcast at all. In this case, the violation window of this thread on this memory location is reduced down to only the exposed read by the thread. Hence, the thread can be violated but cannot cause more speculative threads to violate on this memory location, since it never updates the memory location in any meaningful way.

(4) Reduction of high-bandwidth dependences to low-bandwidth communication

Similar to the problem of temporally silent stores is the problem of the use of high-activity variables to communicate low-activity information. An example of this is the use of a queue length to check for the existence of an element remaining to be processed. While the queue length itself is a high-activity variable that frequently changes as elements are added and removed from the queue, the existence of at least one element remaining to be processed is typically a much lower-activity variable that remains true except in the rare case that the queue is empty, for queues that tend to have significantly more than one member. The frequent updates to the high-activity variable cause frequent violations for threads checking that variable, even though the low-activity information they actually require frequently remains unchanged.

(5) Stacks, queues, heaps, lists, linked lists and sequentiality

Access to data structures with data hot spots can result in high contention between threads and a high violation rate. Common structures with hot spots include stacks, queues, lists and linked lists (determining the number of elements and accessing the first or last element, which can often change), tree-based heaps (determining the number of elements and accessing the top nodes of the heap) and data structures where the elements are sequentially linked in any fashion (such as ordered lists). Likewise, algorithms with any of these characteristics yield high violation rates. Particularly difficult are recursive algorithms. These implicitly include a stack, and the leaves of recursion are often implicitly sequentially linked.

(6) Average and variance of the execution times of individual iterations

The remaining problems that commonly limit parallel performance for TLS pertain to the sizes and execution times of the threads. These are problems for conventional parallel programming, as well. The first is that typically it is optimal if the individual threads require approximately equal time to execute. However, in some applications speculative threads (that are ordered contiguously) require substantially different numbers of cycles to complete. This leads to load imbalance and stall time for threads that complete early, but must wait for less speculative threads to commit first, since threads must commit in order with our implementation of TLS. Some TLS architectures can allow new, more speculative threads to begin execution before old ones have committed, i.e. they allow a temporary increase in the number of threads in progress, but this requires extra hardware and complexity beyond basic TLS implementations [6]. Threads with high execution time variability typically occur when a loop's iterations are parallelized, and only some of the iterations make a function call or conduct some other form of conditional execution.

The second problem pertains to the average size of each individual thread. Threads that are too small fail to provide good parallel performance because practical TLS systems incur some execution overhead per thread that is executed. The overheads for our TLS implementation are discussed in the description of our simulated hardware below. On the other hand threads that are too large fail to provide good parallel performance due to the large exposure of each thread to suffering violations. As discussed previously, when a violation occurs in most practical TLS systems, the entire execution of the violated thread is discarded and execution is started again. The longer the thread, the more execution time will be lost on average. Also, the longer the thread, the more chance that a violation will occur for many applications. This is because most applications will have more exposed reads during that time and longer violation windows, which cannot be overlapped with more-speculative threads' violation windows, as discussed above.

While checkpointing could allow for longer thread lengths, it requires significant hardware overhead, and the benefits it provides are typically small. It only limits the loss of execution due to a violation and does not eliminate the occurrence of the violation. Anyway, the problem of thread sizes generally manifests itself as having threads that are too small, rather than threads that are too large and for which, furthermore, checkpointing would be useful.

Finally, a specific problem of uneven execution times and choosing an optimal thread length arises with nested loops. As discussed above, our implementation of TLS does not allow for multiple-level or re-entrant speculation. As a result, nested loops in which the inner loop is conditionally executed can lead to load balancing problems. This problem arises if the inner loop is infrequently executed per iteration of the outer loop, but consumes so much execution time when it is called that it consumes approximately half of the total execution time of the inner plus outer loops put together. Parallelizing just the inner iterations can yield threads that are too small and for which the TLS execution overheads are too high. Additionally, this leaves the outer loop body unparallelized. Those iterations that do not call the conditional inner loop run entirely sequentially, as do the rest of the loop bodies even for those iterations that do execute the inner loop. However, parallelizing just the outer loop iterations yields short threads when the conditional inner loop is not executed and much longer threads when the inner loop is executed. This is a specific load balancing and thread length problem that arises fairly often.

2.3.2. Secondary performance limiters

The performance limiters listed in this section have been termed secondary because they do not occur often in applications, they do not severely impact performance or both. All four limiters are directly related to implementation issues for practical TLS systems. As such, they are not inherent to TLS itself, but rather arise from the simplifications that are made to render the hardware less complex.

(1) Execution stall due to speculative state buffer overflow

In order to prevent the writes from a speculative thread from being affecting the values read by less speculative threads, these writes must be buffered until they can be committed when the thread becomes the head (non-speculative) thread. This requirement to buffer writes requires per-thread hardware that is limited in practical TLS implementations. When the limitations of the hardware have been reached, the speculative thread can no longer conduct any writes to new memory locations, and this requires stalling the thread at the first write which overflows the speculation write buffers.

Likewise, to track which memory locations have been read speculatively in order to know when a write from a less speculative processor has violated this exposed read, a list of memory locations that have been speculatively read by this thread must be maintained. When the storage for this list has been exceeded, then this thread cannot make any exposed reads to memory locations that are not already stored in this thread's list; otherwise, the thread must stall until it becomes the head thread.

Applications that have been split into speculative threads that cause buffer overflow suffer performance losses due to stalling. But this is rare because it usually only happens for long threads, and at these lengths the high violation rates and large amounts of lost execution time for long threads typically inhibit performance more than buffer overflow. When it does occur for applications with low violation rates, these long threads can often be split into smaller threads to prevent buffer overflow.

(2) Speculative region overheads

Besides per-thread overheads, practical TLS implementations have speculative region startup and shutdown overheads. These are incurred each time execution of the application switches from sequential to TLS-parallel mode. If the amount of work to be done within the speculative region is small, these overheads can reduce performance. However, this is only the case when speculative regions occur frequently and very little execution time is spent in each region. This is not typical for most applications parallelized.

(3) Small number of loop iterations per speculative region

The speculative threads run on a TLS system will usually be assigned to multiple processors. When the number of threads in a speculative region is less than three or four times the number of processors implementing the TLS system and the number of threads is just slightly more than a multiple of the number of processors in the system, performance can be significantly reduced due to the discrete effects of assigning a limited number of threads to processors. For example, if there are five similarly sized threads in a speculative region executing on four processors, one processor will need to conduct twice as much execution as the other three processors, yielding an optimal speedup of only two-and-a-half, rather than four. This is not often a problem, except for loops with long but few threads and with the advanced TLS technique of speculative threading discussed in Section 3.2.10.

A similar problem arises even when the number of threads is one or two times the number of processors in the TLS system. Here the problem is that upon beginning the first threads and upon ending the last threads, some of the processors will be idle. In the former case, it is those processors that have yet to start, and in the latter case it is those that have already finished. This leads to performance losses, but these are significant only when the number of threads is small and there is significant transient violation activity in the speculative region which would be better amortized if more threads were executed. This is the case when the violation windows of the speculative threads force them to begin execution offset at a spacing that is roughly the average thread length divided by the number of processors in the system on average. This combination of the number of threads and the specific violation behavior is not frequently encountered.

(4) False sharing and false violations

As is common for multiprocessor systems, performance on practical TLS implementations can suffer from false sharing. An additional problem with practical TLS implementations can occur due to the fact that violations on memory locations are not detected on a byte basis, but rather on a word basis or larger. In a condition similar to false sharing, false violations can occur because of exposed reads by one speculative thread and writes to a different portion of the same word or line by another, less speculative thread. In the TLS implementation I simulated, violation tracking is done on a word basis, and false sharing at the byte level is atypical amongst common applications. However, for TLS systems that track violations at a larger memory location granularity, this can introduce some performance losses in a few applications.

2.3.3. Measuring and understanding performance losses

When parallelizing applications with TLS, in order to understand the performance losses and diminish them, it is very helpful to be able to measure a number of key characteristics of the application as it executes with TLS. The first is the number of violations. Another is the lost execution time per violation. With the number of violations and the average lost execution time per violation, the total lost execution time due to all violations can be computed. While the ultimate measure of the parallel performance rendered with TLS is the total execution time of the application, the total lost execution time due to violations can help understand how much of the difference from perfect parallel speedup is due to violations and how much is due to overheads, either in the TLS implementation or in the redesign of the code to make it more amenable to parallelization.

Average lost execution time per violation for the entire application is useful, but much more useful is this same measure individually measured for each speculative region. Violation activity can vary widely across different speculative regions, and understanding where parallel performance is lost is simplified greatly with this level of detail. Likewise, the number of violations for each speculative region is helpful.

Another measure that is useful is the number of cycles or the percentage of execution lost to the overhead of the TLS system. Again, this is most useful when provided separately for each speculative region.

As just mentioned, the total execution time of the parallelized application is the correct measure to optimize for a parallel application. While this is similar to decreasing the total lost execution cycles due to violations, it is not the same. For example, applications with more TLS overhead or applications which have been redesigned may lose fewer cycles to violations, but nevertheless take more cycles to complete. Likewise, intuitively one might believe that reducing the number of violations is the best way to improve the performance of a TLS application, but that is not always the case. Redesigning an application to have many short violations early in its speculative threads can sometimes enable better performance than allowing it to execute a few long violations much later in its threads, instead.

In the same way that statistics for each speculative region are much more useful than statistics for the whole application aggregated together, likewise information for each violation are far more useful than information aggregated across an entire speculative region. For this, the simulated TLS implementation used for our research produces a violation trace, which includes the following information. For each violation between threads, the TLS system adds one record to the violation trace. The record includes information on the instruction address of the exposed load that was violated, the processor or thread that conducted that load and the cycle at which the load was initiated. The same information is recorded about the store that violated the exposed load. The information about the processors or threads involved also includes information about the speculative ordering of the threads. Additionally, the execution time that has passed since the violated thread began is recorded, so that the lost execution time due to the violation can be computed for each violation that occurs.

2.4 *TLS CMP hardware simulated*

Having described the applications that were selected, I will now describe the simulated TLS multiprocessor system that was used as the platform to execute the parallelized code. The system chosen for this study, the Stanford Hydra [11], comprises four pipelined MIPS-based R3000 processor cores, each with private L1 instruction and data caches, as shown in Figure 2-2.

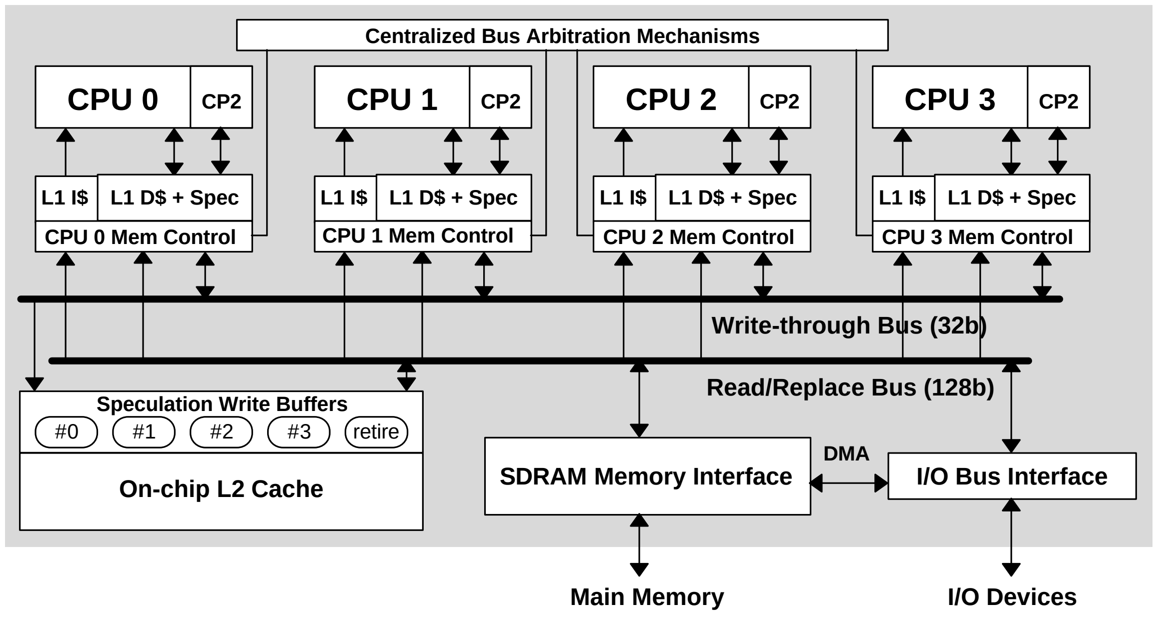


Figure 2-2: Hydra chip multiprocessor

The four processors share an on-chip, unified L2 write-back cache, and each processor executes a single thread. Each processor's L1 data cache is write-through, and the other processors snoop the bus connecting the processors and the L2 cache to permit data dependence violation detection. Dependences are tracked on a per-word basis, thereby eliminating almost all violations due to false sharing. Speculative result buffering is achieved by buffering speculative writes to the L2 cache in a group of 32-cache-line buffers, one for each processor. These buffers also monitor read requests made to the L2 cache. This allows them to forward data created by writes from less speculative processors to satisfy the requests of more speculative processors. Other hardware in the L1 data caches enforces the TLS protocols, such as the detection and processing of dependence violations.

Table 2-1: Memory system specifications

Characteristic	Memory system		
	L1 cache	L2 cache	Main memory
Configuration	Separate I & D SRAM cache pairs for each CPU	Shared, on-chip SRAM cache	Off-chip DRAM
Capacity	16 KB each	2 MB	256 MB
Bus width	32-bit connection to CPU	256-bit read bus and 32-bit write bus	64-bit SDRAM at half of CPU speed
Access time	1 CPU cycle	5 CPU cycles	At least 50 cycles
Associativity	4-way	4-way	N/A
Line size	32 bytes	64 bytes	4 KB pages
Write policy	Writethrough, no write allocate	Writeback, allocate on writes	“Writeback” (virtual memory)
Inclusion	N/A	Inclusion enforced by L2 on L1 caches	Includes all cached data

While the CMP hardware follows a partial store ordering memory consistency model, the TLS system causes the CMP to appear to the programmer like a single out-of-order processor, so the programmer need not consider memory consistency issues. Further details of this design can be found in Table 2-1 and [12]. While Hydra’s MIPS R3000 cores do not aggressively extract ILP, the TLP extracted for the speedups in this thesis are effectively independent of ILP due to the larger thread lengths involved, as validated by the thread lengths provided in our discussion of results in Section 4.4 and Table 4-5.

As a result, an implementation with high-ILP cores should still be able to find ILP, for an even greater speedup.

Table 2-2: Loop-only TLS overheads

Overheads for loop-only TLS	Software handler	Instruction count
Regular events	Start loop	~30
	End of each loop iteration	12
	Finish loop	~22
Irregular events	Violation: local	7
	Violation: receive from another CPU	7
	Hold: buffer full	12
	Hold: exception	17 + OS

The support for TLS is implemented in a combination of hardware and software for ease of implementation and adaptability of the protocols, although this does increase the thread control overheads. Software handlers are executed for regular events such as starting a speculative loop, ending a speculative loop and for completing each iteration within the loop. Other software overheads are incurred for irregular events such as processing violations committed within this processor or within another processor, preventing speculative buffer overflow and handling exceptions for SYSCALL instructions. These software overheads are provided in Table 2-2 and further information is available in [12] and [26]. The overhead due to the software handlers for regular events can be statically analyzed, while the software handlers for irregular events arise from dynamic situations.

The TLS system allows speculation only on loops and at a single level, i.e. not speculation on a loop nested within another speculative loop. The system could have conducted procedural speculation via the use of different software handlers [26], but loop-only speculation was chosen for its lower overheads. As a result, the performance losses resulting from the speculation software handler overheads are typically quite small.

A cycle-accurate, execution-driven simulator was used to simulate all application instructions, including the software handlers that support speculation. Only system calls were executed outside the simulation environment on the underlying native hardware, but these account for a very small percentage of the benchmarks' execution times on real hardware. The simulator has the capability to switch between two modes. In simulation mode, execution on a TLS CMP is realistically modeled and simulation measurements can be made. Contrastingly, in native mode the sequential version of an application is executed directly on the native machine to rapidly advance the simulation without statistics. Performance was measured under two scenarios, a CMP with a realistic memory system for which all memory delays were accurately simulated, and a CMP with a perfect memory system. The realistic memory model includes the effects of bus contention and memory access queuing. The prefect memory model was used to gauge the performance losses due to not scaling the memory system with the number of processors running in parallel. Performance was also estimated for a scenario in which the overhead from the software handlers was entirely eliminated by the use of dedicated hardware to replace the software handlers. Applications were compiled for the target architecture using GCC 2.7.2 with optimization level -O2 on SGI workstations running IRIX 5.3 or IRIX 6.4.

This chapter has provided an understanding of the tools that were used to conduct the research in this dissertation. The next chapter shows how these tools are used to parallelize programs and how using TLS simplifies the process of manual parallelization.

3 Manual Programming with TLS

I have described the mechanisms by which TLS allows multiple threads to execute in parallel, while providing the programmer with the appearance of sequential execution. The programmer or automatic parallelizer suggests portions of the application that are likely to exhibit TLP. The TLS system then dynamically verifies that this TLP actually exists. If data contention prevents parallel execution, the TLS system forces properly ordered execution.

The use of dynamic dependence checking to validate TLP and enforce correctly ordered execution might appear to be a fairly modest capability. However, it is actually a very powerful tool for parallelization. In this chapter, I investigate the substantial impact of this single capability on the entire process of parallel programming. Parallelization using TLS fundamentally changes the conventional approach to parallel programming. It simplifies parallelization, especially of legacy code, because it does not require the programmer or the automatic parallelizer to understand the data dependences that can occur within the application.

I start by describing the general process of manually parallelizing using TLS. I then use a microbenchmark to illustrate this process. I use this as a platform to branch into a description of various programming techniques that allow TLS to extract more TLP out of legacy applications. Finally, I conclude the chapter with a comparison of manual TLS parallelization with conventional parallelization, focusing on how much effort parallelizing this microbenchmark would have required either with or without TLS.

3.1 Parallel programming process using TLS

By providing the programmer with a sequential programming interface, manual parallelization with TLS is conducted very differently from conventional, non-TLS manual parallelization. Conventional parallelization typically requires extensive planning to avoid data races between different threads accessing shared memory. TLS does not, because data races cannot occur. All dependences are automatically resolved correctly. Because TLS systems appear to execute all instructions in order, data races are simply not possible.

This difference profoundly affects the way in which parallel applications are developed. With conventional parallelization, often a sequential application is written first or already exists. Then this application must be transformed into a parallel version by redesigning the algorithms and the data structures to reduce contention. For each possibility of data contention, a determination must be made whether these contending accesses must be ordered or not. Typically, they must be, and then access ordering must be enforced via synchronization methods such as locks, barriers and flags. If any possible instance of data contention is overlooked or incorrectly synchronized, data races may occur and generate incorrect execution. Detecting data races can be complicated, as they may occur infrequently and may only arise under very specific circumstances that are difficult to repeat.

In contrast, TLS can be applied to parallelize a sequential application without making any significant changes to the source code. Because TLS applications appear to run sequentially, manual TLS programming is roughly equivalent to single-threaded (sequential, uniprocessor) programming. The programmer simply indicates to the TLS system which sections of the application may contain TLP and which variables are shared. The application is then executed by the TLS system while the performance is measured. Then, the TLS system returns statistics about the performance of the execution of the parallelized application. This performance data is used by the programmer to modify the application to expose more of the inherent TLP to the TLS system and to reassess which portions of the application actually do possess TLP.

Initially, the programmer begins with a sequential application. If a sequential application does not yet exist, then the programmer develops it. The programmer should develop this application in a manner that makes it more amenable to TLS parallelization, as discussed in Section 5.2. However, this need not be done to the same extent as with conventional parallel programming. Only contentious accesses to shared memory locations that cause large performance losses need be specially addressed. Other shared memory accesses can be synchronized automatically by the TLS system with little loss in performance.

Once a sequential application is available, the programmer profiles this application on typical workloads to determine the execution time spent within each basic block of the application. The profile information ideally also includes information about the control flow interrelationships between the basic blocks. In other words, for any basic block under consideration, it is useful to know which basic blocks are usually executed immediately prior to it and immediately after it, and the percentage of times the control flow of the program proceeds through any of these leading or following blocks. If this is not available at the basic block level, the same information on a procedure/function call level can provide much of the necessary information.

The programmer uses the basic block execution times and the control flow interrelationships between them to predict spots where TLP may be effectively extracted. In the initial basic parallelization, the programmer will focus upon parallelism at the loop level. In the subsequent performance-tuning stages of parallelization, the programmer will examine the execution profile at the basic block level to conduct load balancing and the advanced methods described in Section 3.2 below, such as complex value prediction and speculative pipelining. When conducting basic parallelization, the programmer should focus on the loops where most of the execution time is spent. In applications that are amenable to TLP, the loops that dominate the execution time are almost always repeated a large number of times. The programmer must decide at which level to parallelize these loops using TLS. If a loop has many iterations per entry into the loop, then parallelizing each iteration of the loop as a single thread may be the best decision. However, if the size of a loop is large or dependences exist within or between iterations, it can be desirable to split an iteration into multiple threads to prevent violations from occurring. Likewise, if an iteration is too small, it may make sense to execute several iterations of the loop as a single thread. The programmer must make these decisions using the profile data, and then insert directives to the TLS system to denote how the system should attempt to extract TLP from the application. As discussed in Section 2.4, the TLS system utilized for our study provides only non-reentrant speculation. Hence, the programmer may need to decide the best level at which to parallelize an application when nested loops and other nested forms of iteration are encountered.

Having demarcated points in the application at which to begin a speculative region and end a speculative region and the points within each region at which to begin each new thread, the programmer executes the program on a TLS system, while allowing the TLS system to measure the execution performance data discussed in Section 2.3.3. The programmer can then inspect the number of violations, the lost execution time due to violations and the overhead of the speculative system beyond lost time due to violations. Most importantly, the programmer can inspect which pairs of lines in the source code cause the greatest losses in parallel performance due to violations. This can be done by inspecting the instruction addresses of the loads and the stores that interact most severely with each other by causing violations that limit performance. This allows the programmer to focus the optimization effort on only those violations that most critically reduce performance. This is in contrast to the necessity for conventional parallel programmers to synchronize every single data dependence that could even potentially be executed with incorrect ordering. The methods of analyzing the data gathered while monitoring TLS execution and the type of optimizations that can be implemented to prevent performance losses will be described later in this chapter using a microbenchmark as an example application.

The programmer iterates through this process of executing the parallelized program on a TLS system with performance monitoring capabilities, and then adjusting the program to reduce performance losses due to data dependence violations. The programmer does not need to be certain that TLP can be extracted from any section of the application. Because it is fairly simple to notate the program for sections that should be executed using TLS and to demark the start of each speculative thread and because correctness is guaranteed, the programmer is free to fairly quickly experiment with different thread sizes and locations to find the particular set of threads that yields the best performance. The only part that can be difficult is for the programmer to specify to the TLS system the memory locations (variables) that are shared. In the worst case, the programmer just declares all variables (that are likely to be shared) as shared variables to ensure that the TLS system monitors all these variables for violations that require correction. However, this can introduce a large amount of overhead into some programs, since in many TLS systems, including ours, shared variables must have all exposed reads and all writes directed to the shared cache, which prevents these variables from being register-allocated. Hence, the programmer will generally take some care to only specify as shared variables those that have some reasonable chance of being shared. If the programmer does fail to specify a shared variable, the application can execute incorrectly, as in this case data dependences will not be dynamically detected nor corrected on these memory locations. It is worth mentioning that this is the only place in which the original parallelization effort can require debugging. Debugging can also occur afterward, when the programmer attempts to improve performance by redesigning the application in a way that does not adhere to the original algorithms. Other than these two instances, debugging is never necessary with manual TLS parallelization. Instead, all efforts expended by the programmer are toward performance improvement, rather than correctness.

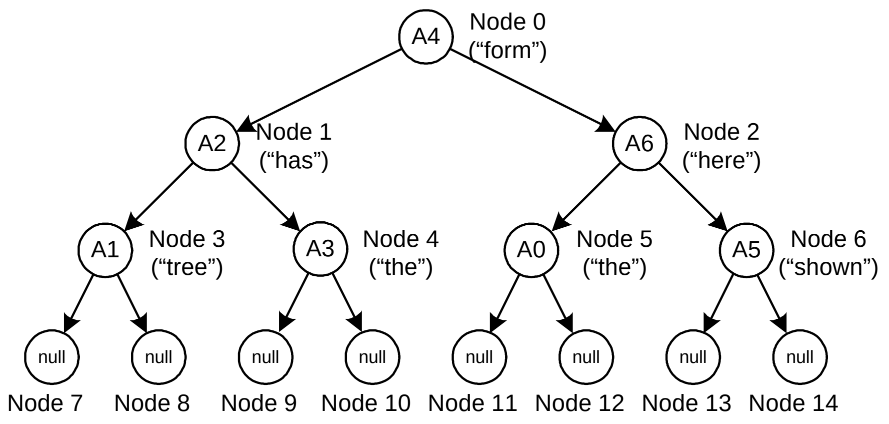
3.2 Microbenchmark example

To make the approach to manual TLS programming more concrete, in this section I will use two simple examples to illustrate many important points about how a programmer can use TLS to parallelize applications. First, I will show the ease of using TLS versus conventional (non-TLS) manual parallelization. Second, I will discuss the performance advantages of using even simple TLS parallelization versus a thorough redesign of applications using conventional parallelization. I will also show the performance advantages of manual over purely automatic TLS parallelization. Third, I will explain several types of source code transformations that can expose more of the TLP inherent in applications. Fourth, I will illustrate the very different code development cycle experienced by a manual TLS programmer.

A) Tree structure in memory

Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Address stored	A4	A2	A6	A1	A3	A0	A5	null	null	null	null	null	null	null	null

B) Implicit structure



C) Data elements in memory

A0	"the"
A1	"tree"
A2	"has"
A3	"the"
A4	"form"
A5	"shown"
A6	"here"

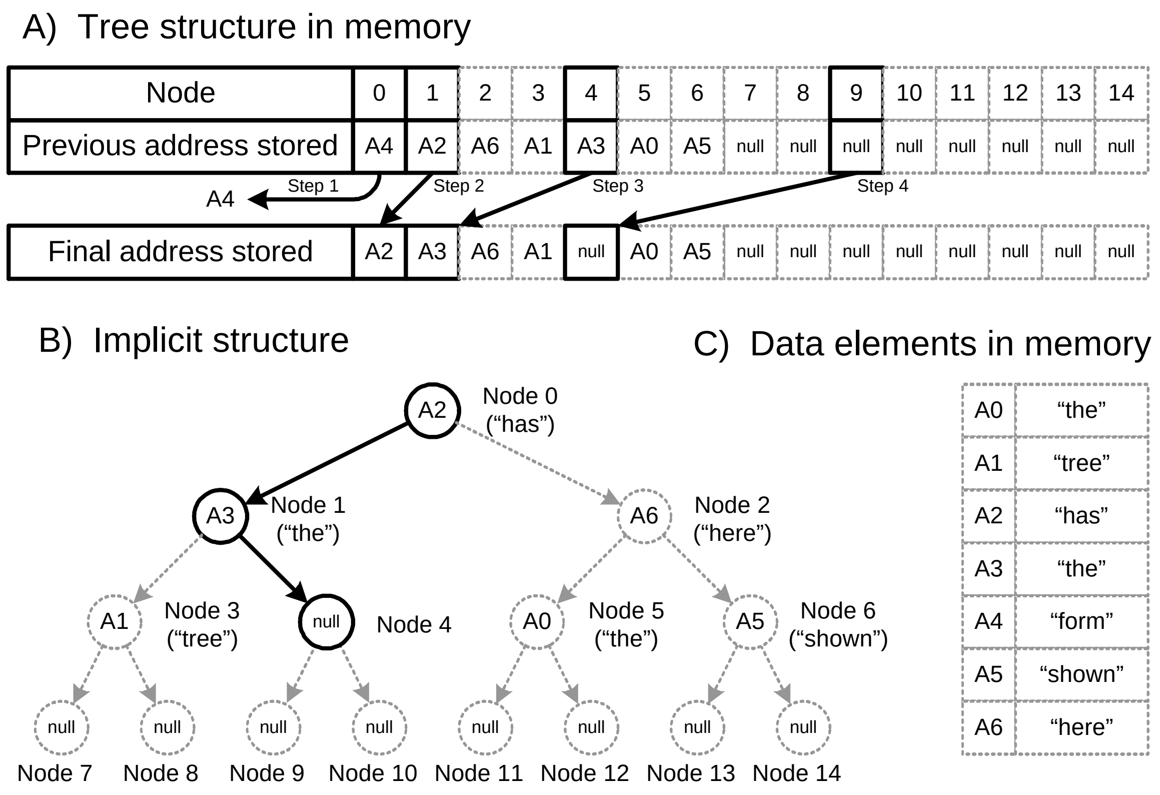
Figure 3-1: Organization of the heap array

3.2.1. Heap Sort Example

The first example is C code that implements the main algorithm for a heap sort. In this algorithm, an array of pointers to data elements is used to sort the elements. Encoded in memory as a simple linear array (Figure 3-1A), the node array is actually interpreted as a balanced binary tree by the algorithm (Figure 3-1B). Tree sibling nodes are recorded consecutively in the array, while child nodes are stored at indices approximately twice that of their parents.

For example, Node 2 is located directly after its sibling (Node 1) in the array, while Node 2's children (Nodes 5 and 6) are located adjacent to each other with indices approximately twice that of Node 2. This structure allows a complete binary tree to be recorded without requiring explicit pointers to connect parent and child nodes together, because the tree structure can always be determined arithmetically. In this example, each node of the tree consists of a single pointer to a variable-length data element located elsewhere in memory (Figure 3-1C).

The heap is partially sorted. The element pointed to by any parent is always less than the element pointed to by each of the children, so the first pointer always points to the smallest element. Nodes are added to the bottom of the tree (highest indices) and bubble upwards, switching places with parents that point to greater valued elements. Final sorting is conducted by removing the top node (first pointer) and iteratively filling the vacancy by selecting and moving up the child pointer that points to the lesser element (Figure 3-2). I will focus only on this final sorting, which typically dominates the execution time of heap sort.



Node	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Previous address stored	A4	A2	A6	A1	A3	A0	A5	null	null	null	null	null	null	null	null
Final address stored	A2	A3	A6	A1	null	A0	A5	null	null	null	null	null	null	null	null

A0	"the"
A1	"tree"
A2	"has"
A3	"the"
A4	"form"
A5	"shown"
A6	"here"

Figure 3-2: Top node removal and update of the heap

The code is provided in Figure 3-3. It can be used to count the number of appearances of each (linguistic) word in a passage of text. It has been optimized for uniprocessor performance, so that parallelization with TLS can only derive speedups due to true parallelism and not due to more efficient code design. The code processes the pre-constructed heap `node[]`, where each node (e.g. `node[3]`) is a pointer to a string (line 2). As each top node is removed and replaced from the remaining heap, a count is kept of the number of instances of each string dereferenced by the nodes (line 17). Each string and its count are written into a (previously allocated) result string (line 2) at the position pointed to by `inRes` (lines 9-16). To do this, the top node of the heap (`node[0]`, which points to the alphabetically first string) is removed and compared to the string

```
#define COLWID (30)
char *result, *node[];

void compileResults() {
 char *last, *inRes;
 long cmpPt, oldCmpPt, cnt;
 int sLen;

 //  INITIALIZATION
 inRes = result; last = node[0]; cnt = 0;

 //  OUTER LOOP - REMOVES ONE NODE EACH ITERATION
 while (node[0]) {
  //  IF NEW STRING, WRITE LAST STRING AND COUNT
  //  TO RESULT STRING AND RESET COUNT
  if (strcmp(node[cmpPt=0], last)) {
   strcpy(inRes, last);
   sLen = strlen(last);
   memset(inRes+sLen, ' ', COLWID-sLen);
   inRes += sprintf(inRes+=COLWID,"%5ld\n",cnt);
   cnt = 0;
   last = node[0];
  }

  cnt++;

  //INNER LOOP - UPDATE THE HEAP, REPLACE TOP NODE
  while (node[oldCmpPt=cmpPt] != NULL) {
   cmpPt = cmpPt*2 + 2;
   if (node[cmpPt-1] && !(node[cmpPt] &&
    strcmp(node[cmpPt-1], node[cmpPt]) >= 0))
    --cmpPt;
   node[oldCmpPt] = node[cmpPt];
  }
 }

 //  WRITE FINAL STRING AND COUNT TO RESULT STRING
 strcpy(inRes, last);
 sLen = strlen(last);
 memset(inRes+sLen, ' ', COLWID-sLen);
 sprintf(inRes+=COLWID, "%5ld\n", cnt);
}
```

Figure 3-3: Code for top node removal and heap update

pointed to by the previous top node removed (lines 8 and 9). If they point to dissimilar strings, then all nodes pointing to the previous string have been removed and counted, so the string and its count are written to the result string and the count is reset (lines 9-16). In all cases, the count for the current string is incremented (line 17) and the heap is updated/sorted in the manner described above (lines 18-23). The heap is structured so that below the last valid child on any tree descent, the left and right child are always two `NULL` pointer nodes (line 18). This whole counting and sorting process is conducted until the heap is empty (line 8). Then the results for the last string are written to the result string (lines 25-28).

3.2.2. Parallelizing with TLS

When parallelizing with TLS, the programmer first looks for parts of the application with some or all of the following qualities. These parts should dominate the execution time of the application with that time concentrated in one or more loops, preferably with a number of iterations equal to or greater than the number of processors in the TLS CMP. These loops should contain fairly independent tasks (few inter-task data dependences), with each task requiring from 200 to 10,000 cycles to complete, and all tasks being approximately equal in length for good load balancing. For the example program, the two loop levels where this code can be parallelized are either the inner loop or the outer loop, i.e. within a single event of removing `node[0]` and updating the heap (lines 8-24), or across multiple such events. The first is not good due to the small parallel task sizes involved, which are better targeted with techniques that exploit ILP. The second level is much better suited to the per-iteration overheads of the TLS system. But, parallelizing across multiple node removals and heap updates requires each thread to synchronize the reading of any node (lines 8, 9, 15, 18, 20, 22) with the possible updates of that node by the previous threads (line 22). The top node will always require synchronization, while nodes at lower levels will conflict across threads with a decreasing likelihood at each level.

This example can be parallelized using TLS simply by choosing and specifying the correct loop to parallelize. In this example, changing line 8 to use the special keyword `pwhile` rather than while can be used with a fairly simple source-to-source translator to trigger the automatic generation of TLS parallel code [26]. The translator performs several operations. First, it analyses the loop to determine loop-carried dependences, i.e. dependences that span iteration boundaries. In this example, these can occur for the variables `node`, `last`, `inRes`, and `cnt`, and also for any access to data dereferenced from a pointer. All variables and accesses that can have loop-carried dependences appear in boldface type in Figure 3-3. Then, it transforms the code so that during every iteration the initial load from and the final store to these variables or to dereferenced pointers occur from or to memory, preventing the data from being register-allocated across iteration boundaries. By forcing data to memory, the transformed source code ensures that the TLS system can detect inter-thread data dependence violations. Meanwhile, all variables without loop-carried dependences are made private to each thread to prevent false sharing and violations. Additionally, for peak performance, the source code is transformed to register-allocate variables having loop-carried dependences in all places other than the first load and the final store in each iteration.

This parallelized TLS code was executed upon a heap comprising the approximately 7800 words in the U.S. Constitution and its amendments. The TLS CMP provides a speedup of 2.6 over a single-processor system with the same, unscaled, realistic memory system. Very little of the difference between the achieved speedup and a “perfect” speedup of 4 is due to not scaling the memory system, as the speedup when both have a perfect memory system is only 2.7. Likewise, the requirement that shared variables not be register-allocated causes only a 2% slowdown, if the code is executed sequentially. This is termed the base TLS parallelization.

3.2.3. Ease of TLS Parallelization

The base case illustrates the simplicity of TLS programming and the efficiency of its resultant programs, in contrast to the complexity and overheads of conventional parallelization. Like TLS, conventional parallelization requires that loop-carried dependences be identified. However, once this has been done, the difficult part of conventional parallelization begins.

Accesses to any dereferenced pointer or variable with loop-carried dependences could cause data races between processors executing different iterations in parallel. While synchronization must be considered for each access, to avoid poor performance only accesses that could actually cause data races should be synchronized with each other. However, determining which accesses conflict requires either a good understanding of the algorithm and its use of pointers or a detailed understanding of the memory behavior of the algorithm. Pointer aliasing and control flow dependences can make these difficult. Finally, a method for synchronizing the accesses must be devised and implemented. This typically requires changes in the data structures or algorithms and must be carefully considered to provide good performance. None of this is necessary when parallelizing with TLS.

In this example, one set of accesses that must be explicitly synchronized when using conventional parallelization are the read accesses of the nodes (lines 8, 9, 15, 18, 20, 22) with the possible updates of those nodes by earlier iterations (line 22). To do this a new array of locks could be added, one for each node in the heap. However, this would introduce large overheads. Extra storage would be required to store the locks. Each time a comparison of child nodes and an update of the parent node were to occur, an additional locking and unlocking of the parent and testing of locks for each of the child nodes would need to be done. Furthermore, doing this correctly would require careful analysis. The ordering of these operations would be critical. For example, unlocking the parent before locking the child to be transferred to the parent node would allow for race conditions between processors. Worse yet, these races would be challenging to correct because they would be difficult to detect, to repeat and to understand.

One could attempt a different synchronization scheme to lower the overheads. For example, each processor could specify the level of the heap that it is currently modifying, and processors executing later iterations could be prevented from accessing nodes at or below this level of the heap. While this would reduce the storage requirements for the locks to just one per processor, it would introduce unnecessary serialization between accesses to nodes in different branches of the heap. Another alternative would be to have each processor specify only the node which is being updated, so processors executing later iterations would stall only on accesses to this node. But, locking overheads would still exist in either case, and care would still need to be taken to prevent data races. Alternatively, the choice could be made to completely replace the uniprocessor heap sort with a new algorithm designed for parallelism from the start. But, this would likely be more complex than any solution discussed so far, and the support for parallelism will still introduce overheads into any algorithm that has inter-thread dependences. As this example shows, parallelization without TLS can be much more complex and error-prone than parallelization with TLS. Because the complexity of redesign versus incremental modification becomes greater for larger, more complex programs, its simplicity is even more of a benefit for real-world applications.

3.2.4. Performance of TLS Parallelization

The base case also illustrates the second point of this section, that parallelization with TLS can often yield better performance than parallelization without TLS [27]. This occurs for two reasons. First, the hardware-assisted automatic detection and correction of dynamic dependence violations reduces communication overheads. Furthermore, it is often possible to speculate beyond potential dependences, eliminating all synchronization stall time when the potential violations do not actually occur. This is termed optimistic parallelization. It can be much more efficient than the pessimistic static synchronization used in conventional parallelization, which synchronizes on all possible dependences, no matter how unlikely.

It is worth considering this point further. Very often, TLS can improve the performance of an application that has already been manually parallelized by allowing some optimistic parallelization [22]. Less apparent is that a single-threaded application only incrementally modified using manual TLS parallelization can sometimes provide better performance than an application that has been completely redesigned for optimal parallel performance using only conventional manual parallelization. This is because code optimized for non-TLS parallel performance introduces overhead over uniprocessor code to support low-contention parallel structures, algorithms and synchronization. The advantage that results from this redesign for conventional parallelism can be less than the combined advantages of using TLS and starting with more efficient, optimal uniprocessor code. Given the difficulty of redesigning legacy code and of parallel programming, this can make manual parallelization with TLS a better alternative than application redesign using conventional manual parallelization.

3.2.5. Optimizing TLS Performance

I will now cover a variety of methods for achieving better TLS parallel performance. This will allow us to focus on three main points: 1) introducing the reader to the process of parallel programming using TLS, which is substantially different from conventional parallel programming; 2) demonstrating several categories of source code transformations that allow extraction of more of the inherent TLP from applications; and 3) indicating situations in which a minor manual adjustment can substantially outperform the automatic base parallelization. I will show how the programmer can detect and understand sources of performance loss and use this to conduct incremental changes to the original source code to improve performance. This process is repeated until no further TLP can be exposed to the TLS hardware.

First, a programmer conducts the base TLS parallelization, as described above, and then executes the resultant code against a representative data set. The TLS hardware is capable of reporting instances of dependence violations, including data on which processors were involved, the address of the violating data element, which load and store pairs triggered the violation, and how much speculative work was discarded. This data is then sorted by each load-store violation pair. By totaling the cycles discarded for each pair and sorting the pairs by these totals, the causes of the largest losses can be known. Using this ranking, a programmer can better understand the dynamic behavior of the parallel program and more easily reduce violation losses.

Compared to non-TLS parallel programming, parallelization with TLS allows the programmer to more quickly transform a portion of code. The key to this is that TLS provides the ability to easily test the dynamic behavior of speculatively parallel code (while it correctly executes in spite of dependence violations) and get specific information about the violations most affecting performance. The programmer can then focus only on those violations that most hamper performance, rather than being required to synchronize each potentially violating dependence to avoid introducing data races into the program.

Before discussing specific code transformations for performance enhancement, I will summarize the general approach to reducing performance-limiting violations. Typically parallel performance is most severely impacted by a small number of inter-thread data dependences. Moving the writes as early as possible within the less speculative thread and the reads as late as possible within the more speculative thread usually reduces the chance of experiencing a data dependence violation. For loop-based TLS, this corresponds to moving performance-limiting writes toward the top of the loop and delaying performance-limiting reads toward the end of the loop; in the limit, the first load of a dependent variable occurs just prior to the last store, forming a tiny critical region. Furthermore, moving this critical region as close as possible to the top of the loop minimizes the execution discarded when violations do occur. Finally, constructing the loop body to ensure that the critical region always occurs approximately the same number of cycles into the execution of the loop and requires a fairly constant time to complete allows the speculative threads to follow each other with a fixed inter-thread delay without experiencing violations. In contrast, critical sections that occur sometimes early and sometimes late increase violations due to late stores in less speculative threads violating early reads in more speculative ones.

3.2.6. Automatic Optimization

I will now consider optimizations that can be done automatically. More than three violations per committed thread occur while executing the base parallelization. The store of last in line 15 often violates the speculative read of last in line 9. The same occurs with `cnt` (the store in line 17 violates the load in line 13), `inRes`, and several other variables. To reduce these violations, the length of the critical regions from first load to last store can be minimized. For example, the store of `last` in line 15 can be moved right after the load in line 9. Because each thread optimally executes with a lag of one-quarter iteration from the previous thread on a four-processor CMP, this makes it unlikely that any other thread will be concurrently executing the same critical region. To hoist the store of `last`, the previous value must first be saved in a temporary variable for lines 10 and 11. Research shows that this transformation can be automated [35]. We can also move these critical regions as early in each thread as possible For example, line 17 (the increment of `cnt`) can be moved above the conditional block (lines 9-16). Automatically determining and conducting this is complex [35]. However, we will assume that automated parallelization can conduct all these transformations optimally to strengthen the argument that manual TLS programming can still further improve performance.

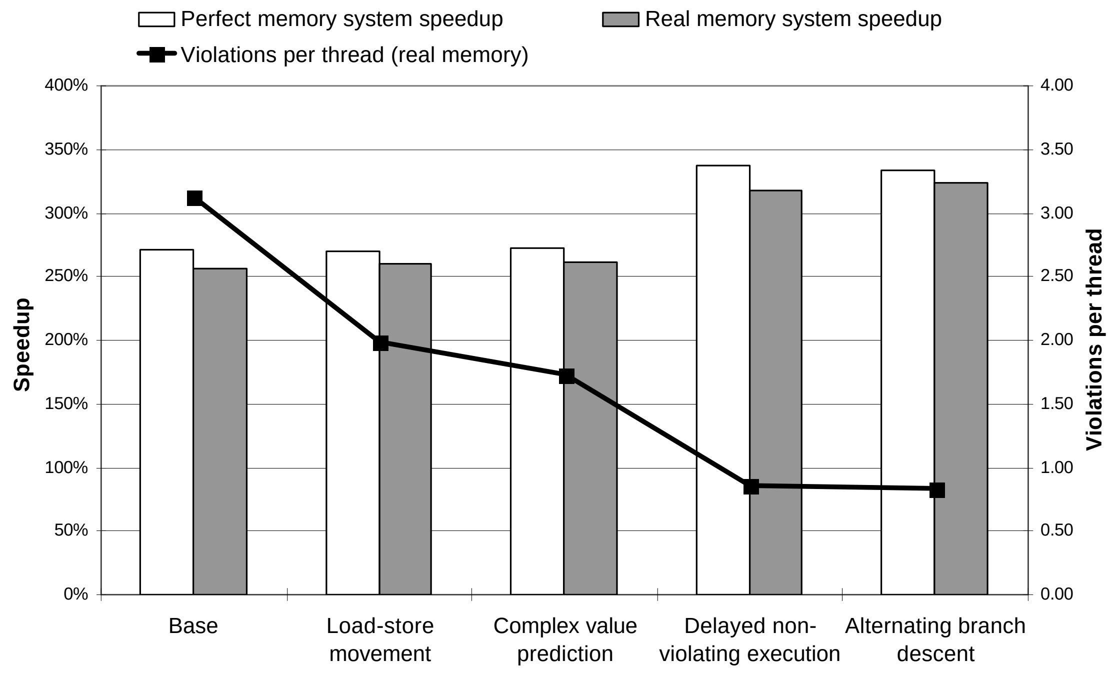


Figure 3-4: Performance of incremental optimizations

When these transformations have been completed for all variables that can benefit, surprisingly the performance remains virtually unchanged. Upon inspecting the violation report, we see that most of the lines which were causing violations before are no longer significant sources of losses, but now previously unimportant load-store violation pairs dominate performance by causing much larger losses than before. Threads now progress farther per violation, but nonetheless violate anyway before they can successfully commit. This results in a lower violation count, but more discarded execution time per violation. This is shown in Figure 3-4, which shows speedup results with real and perfect memory systems and the number of violations per committed thread, for each version of the example application.

Unfortunately, the performance at this point (a speedup of 2.6) represents an optimistic upper bound on the current capability of automated TLS parallelization. We have optimally used all the automated methods of which we are aware that can benefit this example. However, manual TLS parallelization can provide still more speedup (a final speedup of 3.4) with a minimum of code transformation. This is because a programmer can do more complex value prediction than an automated parallelizer. Also, automated parallelization is constrained to allow only transformations that appear to preserve the original execution ordering and data structures, even if a minor, obvious change in them could enhance performance. This arises because the original program was targeted to a uniprocessor, where data contention or value prediction was never an issue, so often a small and obvious change can lessen contention or reduce dependences.

The techniques to be discussed below require an increasingly detailed understanding of the application. However, it should be noted again that these performance optimizations are optional and for improved performance only.

3.2.7. Complex Value Prediction

In the current example, one of the main variables suffering violations is `inRes`. Complex value prediction can reduce these violations. Note that the result string is constructed out of fixed width columns. The first column is `COLWID` characters wide and contains the word (lines 10-12). The next column is five characters wide and contains the final count of the number of instances of the string, followed by a carriage return (line 13). From the code a programmer can determine that the final value of `inRes` will always be `COLWID+5+1` characters greater after line 13 than it was in line 10, but an automated parallelizer would have difficulty deciphering this. Using this prediction of the final value of `inRes`, the programmer is able to hoist the final update of `inRes` above the many function calls in lines 10-13, once again reducing both the chance of a violation occurring and the execution time discarded if a violation does occur.

This violation could perhaps be alleviated automatically using a combination of profiling, violation tracking and a stride predictor. A more challenging example would be if the count of instances were printed to a variable, rather than fixed, length field. Complex value prediction could quickly determine the final value of `inRes` based upon the number of digits used to print `cnt`, but this would be difficult to do automatically using a stride predictor.

Likewise, if the count had been printed to a variable-length field, the programmer could have chosen to change the format to a fixed length to allow for complex value prediction.

This could occur if the output format was not critical and could tolerate a change. If so, this would also show how a small change in the algorithm and data structures can allow further optimization on a program exhibiting contention due to its having been designed without parallel execution in mind. This change would not generally be allowed for an automatic parallelizer.

3.2.8. Algorithm Adjustments

By this point almost all loads and stores to the same variable are placed close to each other and close to the top of each iteration, and yet the performance has not improved significantly. Upon closer examination, we see that many of the violations would never occur if each thread did not execute lines 9-16 and if it maintained a spacing of one quarter iteration from the threads immediately previous to and following it. The problem is that when lines 9-16 are executed, a large number of cycles are consumed to store a word and its count to the result string. Only after completing this, the thread updates the top of the heap (line 22). This violates all more speculative processors, due to the load in line 8, and causes them to discard all their execution during the time the result string was being updated. While conducting an early update on the top node of the heap could yield some benefit, nodes further down would still likely cause violations.

The optimization to alleviate this problem is to move as much of the execution in lines 9-16 to the position following line 23. By minimizing the work conducted before lines 17-23, we can reduce or eliminate many of the violations. In particular, only the updates of data locations with loop-carried dependences should occur before line 17, i.e. updates to `inRes`, `last` and `cnt`. The `strcpy`, `strlen`, `memset` and `sprintf` functions can be conducted later, after lines 17-23, without causing violations. This is similar to moving load-store pairs closer to the start of each iteration, but instead we are making algorithm changes to move non-violating work closer to the end of each iteration. Specifically, we are moving these four functions from before to after the heap update, which repeatedly dereferences dynamically determined pointers. It may be obvious to the programmer that `resultString` and the heap are never intended to have intersecting addresses; hence no violations should occur. However, it appears the compiler would need to conduct either an advanced general analysis or an analysis quite specific to this situation to assert this non-intersection in all cases. Therefore, this is a change in algorithms that may not be possible for an automated compiler to conduct. As Figure 3-4 shows, this optimization greatly improves performance, raising the speedup from 2.6 to 3.2 and also halving the number of violations.

After this optimization, we observe that the dominant remaining violations are the loads in line 20 with the store in line 22. We observe that this is in part due to the fact that when the two child nodes point to equal strings (a common occurrence at the top of the heap), the second (right, higher-index) node is always selected. This leads to frequent contention for all nodes near the top of the heap and resultant violations, as each thread descends down the same path through the heap.

We can easily change the algorithm so that each speculative thread chooses the opposite direction from the thread immediately before it. Consecutive threads will alternate between always selecting the left or always selecting the right node in cases of equality, thereby descending down the opposite path from the immediately previous thread.

Again, a parallelizing compiler could not make this change, because it alters the behavior of the program, even though in this case the program will still produce exactly the same final result string. This final optimization results in slightly improved performance and less frequent violations. Note that including all the transformations so far would yield a 4% slowdown if the code were executed on a uniprocessor. Hence, a perfect, linear speedup on this code would correspond to a speedup of only 3.85 versus the original sequential program.

Further attempts at optimization were unsuccessful. Yet, violations do remain, because they occur infrequently enough that their losses are less than the overheads of reducing them. For example, attempts at synchronizing on the most frequent violations, using locks similar to those used in conventional parallelization, generated excessive waiting times. This supports the assertion that TLS parallelization often performs better than manual parallelization without TLS due to its optimistic execution of code that only occasionally causes violations.

3.2.9. Additional Automatic Optimizations

Several additional automatic techniques exist for improving TLS parallel performance. These were not used in the heap sort example, but will be discussed briefly here for completeness; more details can be found in [5][7][9][20][25][27][30][32]. The techniques comprise loop chunking, loop slicing, parallel reductions and explicit synchronization. Loop chunking refers to unrolling multiple small loop iterations to form each TLS iteration, usually to amortize per-iteration overheads. Loop slicing is the opposite, i.e. splitting each large iteration into multiple, more manageable ones, a technique that represents a simple and automatically transformable case of speculative pipelining described below. Parallel reduction transformations allow certain iterative functions to be parallelized. For example, iterative accumulations into a single summation variable could be instead transformed into four parallel summations that are combined at the end of the loop. Explicit synchronization works much like locks in conventional parallelization to protect a variable and can be used on a frequently violated variable to reduce the violation frequency and the associated discarding of execution [5][7][25]. Unlike its use in conventional parallelization, it is used for performance and not correctness. If the violation data for a TLS parallel application indicates that a read is frequently violated by a write from a less speculative thread, then these two instructions can be explicitly synchronized. By eliminating frequent violations, it trades a large quantity of discarded execution time for a smaller quantity of waiting time.

3.2.10. Speculative Pipelining

Finally we will describe one other very important code transformation, speculative pipelining. Until now, we have focused on single-level, loop-based speculation, because loops are an obvious and easy form of parallelism to extract and because the TLS software speculation overheads for single-level loop-only speculation are low. However, TLS can also extract parallelism from tasks that are not associated with a single loop. For example, Figure 3-5 shows how parallelism can exist at multiple levels within a set of nested loops, making single-level parallelization suboptimal. Here we assume that each of the thousand-cycle routines is a fairly independent task. If only either the outer loop or

```
  for (x=0; x<1000; x++) {
    if ((x%10) == 0)
      for (y=0; y<10; y++)
        InnerLoopOneThousandCyclesOfWork();
    OuterLoopOneThousandCyclesOfWork();
  }
```

Figure 3-5: Original code with independent tasks

```
  for (x=0; x<1000; x++) {
    if ((x%10) == 0)
      for (y=0; y<10; y++)
        InnerLoopOneThousandCyclesOfWork();
    OuterLoopOneThousandCyclesOfWork();
  }

shared_threadChoice = shared_y = 0;
for (shared_x=0; shared_x<1000;) {
   threadChoice = shared_threadChoice;
   x = shared_x;
   if ((x%10) == 0) {
      y = shared_y++;
      if (y == 0) {
         threadChoice=shared_threadChoice=1;
      } else if (y >= 10) {
         threadChoice=shared_threadChoice=0;
         shared_x++;
         shared_y = 0;
      }
   } else
      shared_x++;

   switch (threadChoice) {
      case 0:
         OuterLoopOneThousandCyclesOfWork();
         break;
      case 1:
         InnerLoopOneThousandCyclesOfWork();
         break;
   }
}
```

Figure 3-6: Speculatively pipelined code ready for loop-only TLS

the inner loop is parallelized using single-level parallelization, half the TLP that exists will not be extracted.

With speculative pipelining we break the dynamic execution path of a uniprocessor program between fairly independent tasks and make each task an iteration of a newly constructed loop. To do this, we create a loop shell that chooses between the tasks each iteration by using a `switch-case` statement directed by a dynamically updated thread-choice variable. Figure 3-6 demonstrates how multi-level speculation can be implemented. The outer loop body is represented by `case 0` and the inner loop body by `case 1`.

The selection between them is made by the `threadChoice` variable, which is updated each time program flow switches between executing iterations of the outer loop and the inner loop. New shared variables allow each thread to update early the next thread's value of `x`, `y` and `threadChoice` while maintaining a private copy of the variables to be used for conducting the thread's remaining execution.

The overhead of speculative pipelining is very small; this example has less than 12 extra dynamic instructions per thread, or roughly 1% overhead. But, speculative pipelining allows great flexibility in constructing threads. Unlike regular loop-based speculation, it can create threads that start and end in different functions or that are from portions of the program that do not iterate at all. As a result, speculative pipelining is one of the most powerful and complex techniques for enhancing TLS performance.

As another example, TLP can exist between a procedure and the code following the procedure call. In the past this has been exploited with procedural speculation [26], but speculative pipelining can extract this parallelism with lower overhead. Finally, fairly independent, sequential tasks that are not part of a loop can be parallelized. This is similar to the TLS conducted by Multiscalar [25][37], but because the programmer explicitly selects the parallel tasks and the TLS hardware support is less closely coupled to the processor cores, speculative pipelining focuses on longer threads. In some cases, speculative pipelining can be automatically applied (loop slicing, procedural speculation), but in other cases the technique must be conducted manually.

This chapter has demonstrated the process of manual parallelization using TLS using a couple of simple example applications. In the next chapter, this same process is applied to applications within the SPEC2000 suite of benchmark applications to show the capability of TLS to enable good parallel performance with fairly little programmer effort on a variety of larger applications that are difficult to parallelize automatically, even with TLS.

4 Manual TLS Parallelization of Whole Applications

In Chapter 3 the TLS programming process was discussed, along with useful transformations, focusing on small sequences of code as examples. This chapter discusses how these same techniques can be applied to larger, more realistic applications. The applications selected are from SPEC CPU2000, commonly referred to as SPEC2000.

I start by explaining the reasons why SPEC2000 was selected for this research, and the applications that were parallelized from this suite of benchmarks. Also discussed is the methodology used for sampling the execution of the benchmarks, as the selected inputs to the applications (the reference data sets) result in prohibitively long execution paths for our simulation environment.

Following this, for each application parallelized, I provide a detailed explanation of its parallelization. I tell where thread-level parallelism was found in the application, how it was extracted, which transformations were utilized and what performance was achieved using TLS. While describing the transformations required to parallelize each application, an indication is given for each transformation as to whether it could have been conducted automatically, and if not, why not. Also provided are performance results for the intermediate stages of parallelization leading to the final parallel form of the application. As with the previous microbenchmark example, this provides insight into the parallelization process and the necessity and the usefulness of each of the transformations. Also discussed are some factors that limit the parallel performance that can be achieved.

Having described the parallelism in each application, I present further results and observations derived from simulations of the benchmarks once the transformations discussed above were performed. Patterns in the performance data across benchmarks and memory models is considered first. This is followed by a discussion of the thread sizes utilized and a breakdown of the time spent in different activities during speculative execution. Finally, the focus turns to ease-of-programming issues, looking at the breakdown of time spent parallelizing each SPEC2000 benchmark and the number of regions and lines of unique code that needed to be written for each application by the TLS parallel programmer.

4.1 SPEC2000, benchmark selection and execution sampling

The SPEC2000 benchmark suite was chosen for this study, as it contains a selection of applications with standard input datasets that are widely understood and accepted to be representative of CPU-intensive workloads executed on high-performance processors and memory systems. The SPEC2000 suite contains 14 floating point and 12 integer benchmarks representative of compute-intensive applications. SPEC2000 is specifically designed to test the performance of the processor, the memory architecture and the compiler. For this study of TLP, this is the correct benchmark suite to use. For a study of TLP, the threads under consideration should last for hundreds to a few thousands of instructions. This is too short of an interval to include interactions with more peripheral systems such as input/output devices.

An important reason to use SPEC2000 is that each benchmark is a whole application, except for the exclusion of input and output routines. These are applications in actual use in various locations. They are not optimized or contrived microbenchmarks. They contain some inefficient code and unusual or complex data structures. Therefore, parallelizing these benchmarks gives a good indication of the performance and difficulty of parallelizing all general-purpose, legacy applications of this size, as they were not specifically created for this research or by me. Another reason for using SPEC2000 is that a large number of research publications use SPEC2000 when studying the performance advantages of proposed new approaches to computation. While results are not completely comparable between studies, the use of common benchmarks does allow a high-level assessment of the performance of manual TLS parallelization relative to other methods of enhancing uniprocessor or parallel performance. Furthermore, the analysis conducted in this research of the parallelism within the SPEC2000 applications parallelized can be used by other researchers to improve their techniques of automatically parallelizing these well-known and often utilized applications.

A brief description of each of the benchmarks is provided in Table 4-1, with the applications we selected for parallelization in bold font. It is important to understand the scope of the applications selected as this defines the applicability and the limitations of the research experience gained with using manual TLS parallelization on general-purpose applications. For TLS parallelization, we selected the four floating point applications that are coded in C, `art`, `equake`, `mesa` and `ammp`, since they are more difficult to parallelize than the Fortran benchmarks. We then selected three of the integer benchmarks, `mcf`, `twolf` and `vpr`, based upon their smaller source code size and indications from profiling that they would be amenable to manual parallelization with TLS. Each of these applications spends much of its execution time in a few easily understood routines, or at least the application is fairly short in source code length, which allowed us to easily understand the behavior of the program in order to effectively

Table 4-1: Benchmarks comprising SPEC CPU2000

Benchmark		**Source code language**	**Application description**
CFP 2000	168.wupwise	Fortran 77	Physics: quantum chromodynamics
	171.swim	Fortran 77	Shallow water modeling
	172.mgrid	Fortran 77	Multi-grid solver: 3D potential field
	173.applu	Fortran 77	Parabolic/elliptic partial differential equations
	177.mesa	**C**	**3D graphics library**
	178.galgel	Fortran 90	Computational fluid dynamics: analysis of oscillatory instability
	179.art	**C**	**Image recognition; neural network simulation; adaptive resonance theory**
	183.equake	**C**	**Seismic wave propagation simulation**
	187.facerec	Fortran 90	Image processing: face recognition
	188.ammp	**C**	**Computational chemistry**
	189.lucas	Fortran 90	Number theory: primality testing
	191.fma3d	Fortran 90	Finite element crash simulation
	200.sixtrack	Fortran 77	Particle accelerator model; high energy nuclear physics accelerator design
	301.apsi	Fortran 77	Meteorology: pollutant distribution
CINT 2000	164.gzip	C	Data compression utility
	175.vpr	**C**	**FPGA circuit placement and routing**
	176.gcc	C	C programming language compiler
	181.mcf	**C**	**Minimum cost network flow solver; combinatorial optimization**
	186.crafty	C	Game playing: chess program
	197.parser	C	Natural language processing
	252.eon	C++	Ray tracing; computer visualization
	253.perlbmk	C	PERL programming language
	254.gap	C	Computational group theory; interpreter
	255.vortex	C	Object-oriented database
	256.bzip2	C	Data compression utility
	300.twolf	**C**	**Place and route simulator**

Table 4-2: Source code lengths of the SPEC CPU2000 benchmarks selected

Benchmark		Application category	Lines of code
CFP 2000	177.mesa	3-D graphics library	61,343
	179.art	Image recognition/neural networks	1,270
	183.equake	Seismic wave propagation simulation	1,513
	188.ammp	Computational chemistry	14,657
CINT 2000	175.vpr	FPGA circuit placement and routing	17,729
	181.mcf	Combinatorial optimization	2,412
	300.twolf	Place and route simulator	20,459

parallelize it. Details of the source code lengths of the applications we parallelized are provided in Table 4-2. While a few of the other integer benchmarks look amenable to manual parallelization, it is clear that several would be very difficult or impossible to manually parallelize without an extensive understanding of the algorithms and data structures in use.

Each of the SPEC2000 benchmarks comes with two or three standard input data sets (and execution parameters): a test data set, a training data set and a reference (full-size) data set. While the test and training data sets can enable smaller execution times, the behavior of the applications under these input sets is substantially different from the behavior on the full-size reference data sets. Therefore, the reference input data sets were used for this research. Due to the long execution times of these data sets, complete execution was not possible for any of the benchmarks. Concurrent research on SPEC benchmarks [31] has demonstrated both the difficulty and the importance of choosing carefully the portion of execution to simulate for applications that exhibit large-time-scale cyclic behavior. The sampling strategy utilized in the research for this dissertation is supported by their recommendation to simulate one or more whole application cycles. The total of all simulation samples was at least 100 million instructions from each original (non-parallelized) application. One should note that all speedup and coverage results presented below are based upon an extrapolation of these samples of whole application cycles back to the entire application. The extrapolation was conducted by first profiling the full application using similar real hardware and the same compiler as the Hydra CMP. Full application speedup was then calculated assuming the simulated speedup on the portion of execution time corresponding to the application cycles, and assuming no speedup on the portion of the original execution time that was not a part of the application cycles we sampled.

The benchmarks were profiled on real hardware to determine the percentage of time spent in each basic block and function. Simulations were then executed on evenly distributed samples of execution from throughout the entire reference runs. These samples were generated by first selecting a set of functions that do not overlap in execution, but that span almost the complete execution time. In other words, effectively the entire execution time of the program can be attributed to these functions and the functions they call. Also, these non-overlapping functions were chosen so that they are either called many times themselves or they contain loops that occur frequently and dominate the execution time of the functions. That way, a fixed portion of these calls, say one out of four thousand, could be executed in simulation mode, and the rest could be executed in native mode.

The length of each simulated sample necessarily depends upon the structure of the application, but the total of all samples run in simulation mode was at least 100 million instructions from each original (non-parallelized) application. This sampling approach was validated by comparing the execution cycle breakdown by function of the simulated sections with the native execution profile of the entire application. All results presented below are based upon an extrapolation of these samples back to the entire application, i.e. speedups, coverages, etc. are for the full application, not just the samples.

4.2 SPEC2000 parallelization

In the remainder of this chapter, I describe the ways in which each application was parallelized and the characteristics that limited speedup. Useful parallelism was located within many applications. Because many researchers are very familiar with the applications in SPEC2000, it is useful to be specific here, even listing the names of subroutines and variables within each application.

Table 4-3: Code transformations

Transformation	SPEC CFP2000				SPEC CINT2000		
	177 mesa	179 art	183 equake	188 ammp	175 vpr	181 mcf	300 twolf
Loop chunking/slicing		X	X	X		X	
Parallel reductions		X			X	X	X
Explicit synchronization					X		X
Speculative pipelining				X	X	X	X
Adapt algorithms or data structures					X	X	X
Complex value prediction					X	X	X

Each application was initially parallelized using base parallelization of loops and automatic load-store placement. Table 4-3 lists the additional transformations that were then used. The first three are simple and can be automated; the second three are complex, requiring manual programming.

The transformations were not applied in the order shown in Table 4-3. The actual order in which the transformations were applied is shown in Table 4-4, which details the speedups achieved for each speculative region in each application as the transformations were sequentially added. Ideally, the incremental speedup due to each transformation could be listed. However, the transformations interact with each other. For example, on `vpr (place)` explicit synchronization yielded no speedup after base parallelization with additional value prediction. However, applying it together with the parallel reduction transformation provided a sizeable advantage. Due to the interactions and the many permutations of transformations, we have instead listed the speedups along the single path of transformations we actually followed. Note that because `vpr` is a place and route application and the two portions of the application are very different, we have listed results for them separately.

At each stage, the total speedup and the incremental speedup are provided for a realistic memory model and also a perfect one with and without TLS software handler overheads. All results utilized the reference input data sets. As discussed in Section 1.4.3, because of long simulation times and the large-time-scale cyclic behavior of the reference runs, representative samples encompassing whole application cycles were selected for simulation, similar to the strategy suggested by [31]. It should be noted that in the following discussions, whenever we discuss the execution time, violation frequency or some other statistic for a subroutine, we are including the contributions not only of that subroutine, but of all other subroutines that it calls, as well.

Table 4-4: Speedup resulting from each additional transformation

Application		Specu-lative regions	Location of top level of speculative region(s). Line numbers are for SPEC CPU2000, version 1.00.	Percent execution time coverage	Last transformation applied	Cumul. speedup, real	Increm. speedup, real	Cumul. speedup, perfect	Increm. speedup, perfect	Cumul. speedup, perfect, no overhead	Increm. speedup, perfect, no overhead
CFP 2000	177. mesa	1	vbrender.c, lines 897-901	84%	Basic	175%	175%	179%	179%	179%	179%
	179. art	7	scanner.c, lines 405-477 (7 loops) and 545-617 (7 loops)	95%	Basic	60%	60%	0%	0%	0%	0%
					Parallel reductions	122%	39%	43%	43%	112%	112%
					Loop chunking/slicing	154%	14%	282%	167%	294%	86%
	183. equake	6	quake.c, lines 449-478 (5 loops) and 1195-1220	100%	Basic	135%	135%	185%	185%	195%	195%
					Loop chunking/slicing	145%	4%	196%	4%	200%	2%
	188. ammp	1	rectmm.c, lines 562-1123	86%	Basic	61%	61%	59%	59%	62%	62%
					Speculative pipelining, loop chunking/slicing	99%	24%	69%	6%	76%	9%
CINT 2000	175. vpr (place)	1	place.c, lines 506-513	100%	Basic	7%	7%	16%	16%	17%	17%
					Complex value prediction	55%	45%	67%	44%	68%	44%
					Parallel reductions, explicit synchronization	111%	36%	128%	37%	129%	36%
	175. vpr (route)	1	route.c, lines 518-541	97%	Speculative pipelining	17%	17%	16%	16%	27%	27%
					Algorithm/data structure changes	67%	43%	60%	38%	72%	35%
					Complex value prediction	88%	13%	113%	33%	128%	33%
	181. mcf	6	implicit.c, lines 246-272	44%	Loop chunking/slicing, algorithm/data structure changes	70%	70%	126%	126%	151%	151%
			mcfutil.c, lines 75-76	5%	Loop chunking/slicing, complex value prediction	24%	24%	>300%	>300%	>300%	>300%
			mcfutil.c, lines 80-109	19%	Parallel reductions, speculative slices, speculative pipelining, complex value prediction	55%	55%	10%	10%	16%	16%
			pbeampp.c, lines 96-121	7%	Speculative pipelining, algorithm/data structure changes	84%	84%	95%	95%	119%	119%
			pbeampp.c, lines 161-174	4%	Basic	64%	64%	146%	146%	197%	197%
			pbeampp.c, lines 181-195	20%	Loop chunking/slicing	89%	89%	150%	150%	211%	211%
	300. twolf	1	uloop.c, lines 154-361	100%	Speculative pipelining	18%	18%	21%	21%	23%	23%
					Parallel reductions, explicit synchronization, algorithm/data structure changes	43%	21%	53%	26%	59%	29%
					Complex value prediction	60%	12%	67%	9%	72%	8%

4.2.1. Parallelization of 183.equake

`Equake` is a seismic wave propagation simulation. It is extremely simple to parallelize. The five main calculation loops in the time integration loop were parallelized, along with the loop in the `smvp` subroutine. For the five loops nested in the time integration loop, each loop was individually parallelized, and ten iterations of each loop were chunked together to form each thread, providing a slight additional increase in performance. The parallelization of `smvp` used one iteration per thread.

4.2.2. Parallelization of 179.art

Art is an image recognition application that uses neural networks. Once again, it is extremely simple to parallelize, with 95% of the execution time split between two subroutines, `match` and `train_match`. In these, each loop to update a member of the `f1_layer` (`P`, `Q`, `U`, `V`, `W`, `X`, `Y`) was treated as a speculative region, with ten iterations of each loop chunked together as a single thread. Additionally, the `P`, `V`, `W` and `Y` loops used parallel reductions for summations, and the `Y` loop also utilized a parallel reduction for a logical `AND`. Each of the techniques provided a substantial gain in performance. The poor performance of the basic parallel version under the perfect memory model results from instruction count increases due to both the TLS software handler overheads and the need to force register allocated variables into the caches to allow the detection of violations. These instruction count increases affect the real memory speedups less than the perfect memory speedups, since their effects are masked by the memory delays in the real memory model.

4.2.3. Parallelization of 177.mesa

`Mesa` is a 3-D graphics library. The bulk of execution (84%) occurs within `general_textured_triangle`, which is called many times per execution of `gl_render_vb` in `vbrender.c`. By simply making each call to `general_textured_triangle` a speculative thread, a 175% speedup is gained on this code. However, there is a problem with this approach. The problem arises from the fact that each call to `general_textured_triangle` calls `gl_write_texture_span` several times (an average of 3.5), where most of the execution time actually occurs. This subroutine invokes `gl_depth_test_span_less` in `depth.c` once each time, where reads and writes to an array `zptr[]` occur, causing occasional violations. These violations are especially expensive because each one can force the discarding of all more speculative threads, each thread containing multiple calls to `gl_write_texture_span`, as shown in Figure 4-1A.

One might theorize that less execution time would be discarded if threads were formed at a finer granularity. For example, using speculative pipelining, we could break the original threads into new, smaller threads along the dotted lines in each thread shown in Figure 4-1A. This results in the threads shown in Figure 4-1B, where some may even span multiple calls to `general_textured_triangle`. This could potentially eliminate the dominant violation due to updates to the `zptr` array. However, the recoding necessary to do this is much more complex. Also, it may not provide better

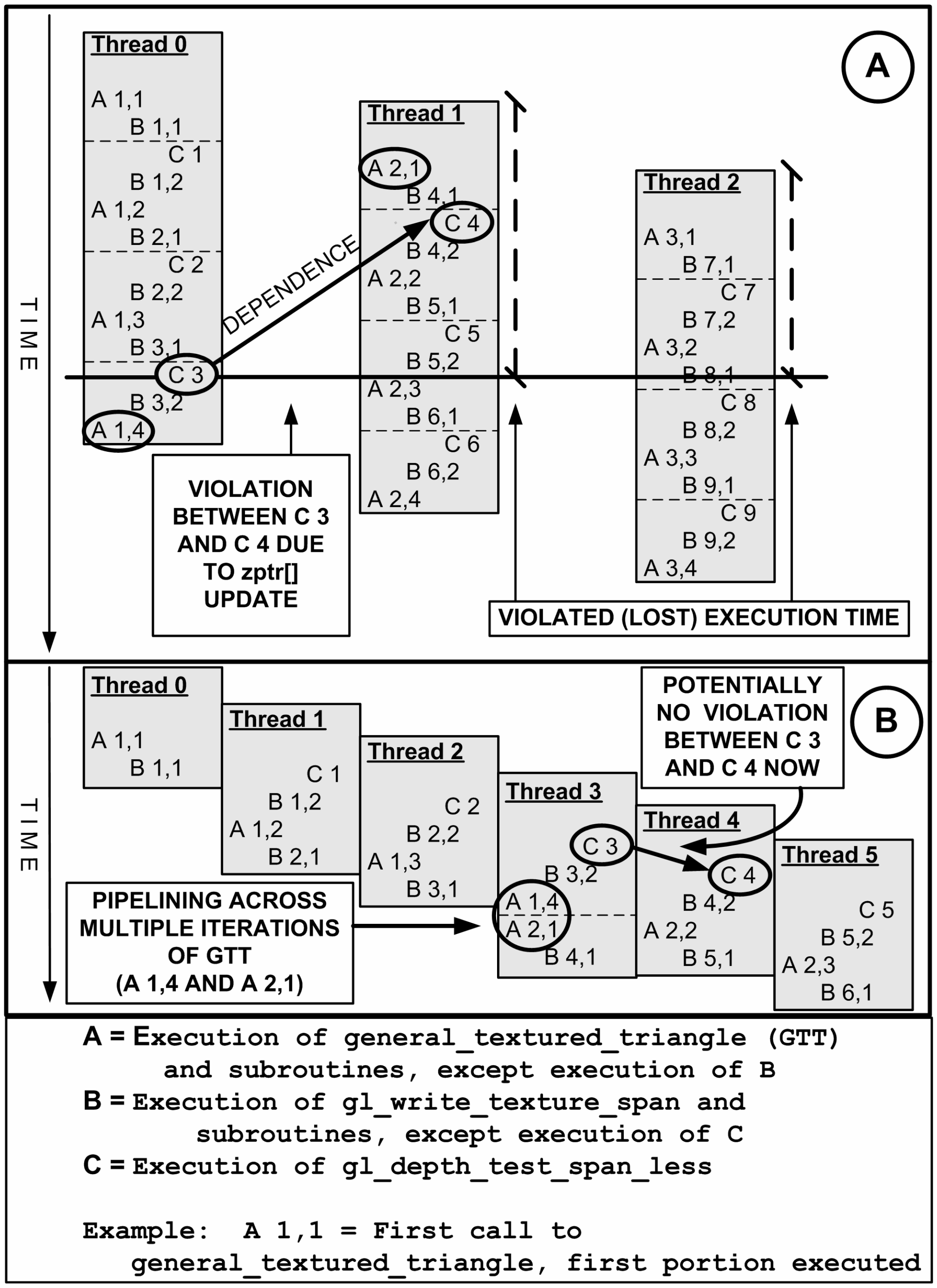

A
TIME
Thread 0
A 1,1
B 1,1
C 1
B 1,2
A 1,2
B 2,1
C 2
B 2,2
A 1,3
B 3,1
C 3
B 3,2
A 1,4
DEPENDENCE
Thread 1
A 2,1
B 4,1
C 4
B 4,2
A 2,2
B 5,1
C 5
B 5,2
A 2,3
B 6,1
C 6
B 6,2
A 2,4
Thread 2
A 3,1
B 7,1
C 7
B 7,2
A 3,2
B 8,1
C 8
B 8,2
A 3,3
B 9,1
C 9
B 9,2
A 3,4
VIOLATION BETWEEN C 3 AND C 4 DUE TO zptr[] UPDATE
VIOLATED (LOST) EXECUTION TIME
B
TIME
Thread 0
A 1,1
B 1,1
Thread 1
C 1
B 1,2
A 1,2
B 2,1
Thread 2
C 2
B 2,2
A 1,3
B 3,1
Thread 3
C 3
B 3,2
A 1,4
A 2,1
B 4,1
Thread 4
C 4
B 4,2
A 2,2
B 5,1
Thread 5
C 5
B 5,2
A 2,3
B 6,1
POTENTIALLY NO VIOLATION BETWEEN C 3 AND C 4 NOW
PIPELINING ACROSS MULTIPLE ITERATIONS OF GTT (A 1,4 AND A 2,1)
A = Execution of general_textured_triangle (GTT) and subroutines, except execution of B
B = Execution of gl_write_texture_span and subroutines, except execution of C
C = Execution of gl_depth_test_span_less
Example: A 1,1 = First call to general_textured_triangle, first portion executed

Figure 4-1: Execution pattern and violations of 177.mesa

performance, if violations increase in frequency due to the sharing of the numerous variables local to `gl_write_texture_span`. Even if violations are not a problem, this still introduces more overhead because of the need to force shared variables out of the registers and into the caches to allow for dependence tracking. This is an example of where a programmer may decide between potentially better performance with a large programming effort, or an adequate speedup with far less effort, depending on the requirements for the performance and the cost of redesign for the final parallel program.

4.2.4. Parallelization of 188.ammp

`Ammp` is a computational chemistry application. 87% of the execution time is devoted to the subroutine `mm_fv_update_nonbon` in `rectmm.c`, which conducts an update of the nonbonded potentials of the atoms. While this subroutine possesses considerable thread-level parallelism (TLP), it is spread between two levels of nested loops, with program execution frequently switching between the two levels, as shown in Figure 4-2. About one-third of the execution time of this subroutine is spent at the outer level, while the remaining two-thirds are spent on the inner loops, which are invoked by 8% of the outer loop iterations. Parallelizing just the inner loops misses the parallelism in the outer loop. On the other hand, using only the outer loop iterations to construct threads results in 8% of the threads being enormous, leading to thread stalls caused by load imbalance.

While it is optimal to parallelize both levels together, our TLS system is not capable of speculating on multiple concurrent regions. However, simple modifications to the source code using a `switch-case` statement allow nested loops to be rewritten as a single-level loop using speculative pipelining [29]. With this transformation, threads can be constructed from different segments of the code, as shown in Figure 4-2. These four different types of threads can be executed in parallel, and this provides a significant performance boost, as shown in Table 4-4. Loop chunking was required to provide good

Code segment	Line numbers in rectmm.c	Percent of total application execution time
A	562 - 567	5%
B	571 - 955	27%
C	957 - 978	5%
D	979 - 986	3%
E	987 - 993	4%
F	994 - 1121	42%

```
outer loop
{
    A ( 5% execution time)                                THREAD TYPE 0
    if (92% true)
        if (68% true * 92% = 62% combined)
            B (27% execution time)
    else (8%)
        if (90% true *  8% =  8% combined)
        {
            inner loop
            {
                C ( 5% execution time)
            }

            inner loop                                    THREAD TYPE 1
            {
                D ( 3% execution time)
            }

            inner loop                                    THREAD TYPE 2
            {
                E ( 4% execution time)
            }

            inner loop                                    THREAD TYPE 3
            {
                F (42% execution time)
            }
        }
}
```

Figure 4-2: Thread formulation for 188.ammp

load balancing by making most instances of the four types of threads similar in size. With this formulation of threads, the remaining limitations to speedup are due to violations caused by late updates of the variable `i_max`.

4.2.5. Parallelization of 175.vpr (place)

`Vpr` is a FPGA circuit place and route application. Because the place and route portions of the program are so different, we will discuss them here as two separate applications. Execution of `vpr (place)` almost entirely comprises repeated calls within `try_place` to the subroutine `try_swap`. Parallelizing this loop yields only a tiny speedup, because of frequent violations due to the serial nature of the pseudorandom number generator used by the routine. Each pseudorandom number is used to generate the next one, and this occurs throughout each loop iteration.

This problem was overcome by determining that the generator is typically called four times by each thread. To improve performance, the rewritten application performs value prediction by assuming the generator function will be called four times. At the start of each thread, the thread computes a predicted seed value for the next thread and writes this prediction to a separate shared variable. This prediction is used by the next thread to set the seed for its random number generator, after which it likewise makes a prediction for the next thread after it. At the end of all possible calls to the generator within a thread, this prediction is checked against the true final value of the seed. If a misprediction has occurred, the prediction is updated to the correct value, resulting in a violation in the next thread and re-exeution with the correctly predicted seed. However, because the prediction is usually correct, violations are unlikely.

This example illustrates the standard way in which complex value prediction can be done in a TLS system. This TLS performance technique may be possible to automate using an advanced compiler in conjunction with application profiling. An alternative solution for this application would have been to remove the artificial dependency by just redesigning the random number generator to avoid this obvious serialization. While this may be apparent to a programmer, unfortunately compilers cannot generally redesign programs.

In addition to using complex value prediction, performance for `vpr (place)` was augmented via parallel reductions for the summation of `success_sum`, `av_cost` and `sum_of_squares`.

4.2.6. Parallelization of 175.vpr (route)

`Vpr (route)` executes almost entirely within the subroutine `route_net` in the nested loop that routes each individual pin. This loop considers the cost of each new path and expands the search around any that are lower cost. Basic parallelization using each iteration of this nested loop to form a single thread is not useful for a number of reasons. Load imbalances occur because approximately half the iterations call `expand_neighbours`, and these iterations are approximately twice as long as iterations in which this does not occur. Speculative pipelining can be used to split iterations that call `expand_neighbours` into multiple threads of a similar length. Once this is done, late updates to shared variables, especially those related to the heap structures, such as `heap_free_head` and `heap_tail`, become a problem. Because the late updates occur within subroutines called from within `route_net`, the best solution was to inline the functions and then promote late updates to the earliest possible point in each iteration.

With these modifications, many of the remaining violations are due to stores that do change the value of a variable, but in a way that only infrequently affects the control flow of concurrently executing threads. For example, when there are no more free elements to add to the heap (`heap_free_head == NULL`) or the heap is empty (`heap_tail == 1`), special actions must be taken. However, most updates to the head or tail of these structures, such as adding or removing a member, do not create these special situations. However, every time the tail or head are updated, a violation will occur on all conditionals predicated on these variables in more speculative threads, anyway.

The solution is to recognize that there are two uses for each of these variables. For example, `heap_tail` is used both to point to the next element to be processed and to indicate when the heap is empty. The next element to be processed changes with a high frequency and is difficult to predict in advance. In contrast to this, whether the heap is empty changes with a low frequency and is easy to predict. By creating an additional Boolean variable to store the value of this low frequency information and then using it to control conditional execution in more speculative threads, we achieve a reduction in violations and therefore better performance.

4.2.7. Parallelization of 300.twolf

`Twolf` is a place and route simulator. Almost all execution is contained within the main loop in `uloop.c`, where the algorithm attempts to place moveable cells in different blocks and measure the cost function. While each of these iterations could form a thread of appropriate length, these threads suffer too many violations.

Approximately three quarters of the execution is spent in calls to the subroutine `ucxx2` in `ucxx2.c`. We inlined this code and separated each iteration of the loop in `uloop` into eight fairly independent portions, some of which are only conditionally executed. Speculative pipelining was then used to dynamically assign one of these eight different portions to each thread. The first portion is all of the code from the main loop in `uloop.c` leading up to the call to `ucxx2`. The last two portions are the code in `uloop.c` following the call to `ucxx2`. The middle five portions are formed from the inlined code from `ucxx2`. This speculative pipelining provided some speedup. However, violations due to shared variables are severely limiting.

After threads have been created with speculative pipelining, the most significant violations between them occur on summation variables such as `cost` and `delta_vert_cost` and on accesses to `netarray` in various subroutines in `dimbox.c` called from `ucxx2`. By using parallel reductions for the summation variables and synchronizing threads that are accessing `netarray`, performance was significantly enhanced. Finally, as in `vpr (place)`, the pseudorandom number generator introduces an unnecessary serialization into the application. Advanced value prediction and communication of the expected final seed value is conducted early in each thread to mitigate this serialization.

4.2.8. Parallelization of 181.mcf

`Mcf` is a combinatorial optimization application. Essentially the entire application can be parallelized by parallelizing four subroutines, `price_out_impl` in `implicit.c`, `refresh_potential` in `mcfutil.c` and `primal_bea_mpp` and `sort_basket` in `pbeampp.c`.

Parallelizing `price_out_impl` requires parallelizing a short inner loop. We chunked four iterations together to reduce the speculation overheads for these short loops. Pointer chasing on the `arcin` variable causes violations due to late updates, but by using simple code motion, these updates can be hoisted to the top of each thread. Prior to executing each of the four iterations within each thread, a check is done of whether the `arcin` pointer is `NULL`. If so, the thread terminates before completing any further iterations, and this condition causes completion of the speculative region.

`Refresh_potential` comprises two loops. The first loop, resetting the nodes, was parallelized using chunks of 100 iterations to form each thread. In the original application, once each iteration the node induction variable must be tested. In the speculative version, advanced value prediction is used to reduce this to just one conditional test per chunk of 100 iterations in most cases. This explains the superlinear speedups under the perfect memory model in Table 4-4. Under the real memory model, memory stall time limits the achievable speedup.

The second loop processes a tree in a depth-first manner. The critical path is the tree traversal code, especially due to memory delays under the real memory model. Hence, to parallelize it we made every alternate thread a speculative slice, which is a thread containing only instructions from the critical path [37]. The intervening threads conduct the actual computation at each node. Unlike the parallelization of `price_out_impl`, the pointer chasing in `refresh_potential` consumes a large percentage of the total execution time. For this reason, it was assigned to its own separate speculative-slice threads, rather than simply moved to the beginning of each thread. Advanced value prediction is used in the speculative slices to predict the final value of the sibling search early, and this prediction is checked and updated, if necessary, when the thread completes its search. We also used parallel reductions for the checksum in the computation threads. The short lengths of both the computation and the node traversal threads, in conjunction with the prefetching effectively conducted by the traversal threads, explain the larger speedups obtained under the real memory model than the perfect memory models.

The two loops in the non-initialization section of `primal_bea_mpp` were trivial to parallelize, with three iterations per thread used for the second loop. Parallelizing `sort_basket` was not easy, due to the recursive nature of the algorithm. However, because the recursion is not deep, typically recursing less than ten times, and because the number of instructions executed at each level of the recursion is very small, it can be parallelized well. This recursive sorting can be modeled as a binary tree, where each node of the tree represents a sorting operation. This tree has the special property that the sorting operation represented by a node does not affect the sorting operation of any node of the tree that is not a descendent of that node. Speculative pipelining was used to create eleven iterations. The first seven iterations conduct the sorting operations in the top seven nodes of this tree that represents the recursive algorithm. This yields eight nodes of sorting operations at the third/fourth level of the tree of sorting operations. These eight sort baskets are assigned to the four processors in the remaining four threads. The first and last baskets of the array to be sorted are assigned to the first processor, the second and the next-to-last baskets to the second processor, etc., in order to provide better load balancing, as the number of elements to be sorted and the degree of sortedness of the baskets varies across the array. The approach could obviously be extended to allow for scalability to more processors.

4.3 Performance-related observations

As expected, the floating point applications were very simple to parallelize compared to the integer ones. Few complex techniques were used on them, and they were very uniform in the nature of their threads. This is in contrast to the many, varying speculative regions in the integer applications, each of which was substantially different from the speculative regions in the rest of the application. While high parallel coverage was managed for all the applications, the integer applications required more effort and more complex parallelization techniques, and each region parallelized required a different approach to parallelization. The complexity of the parallelization of the integer applications made speculative pipelining an essential technique, as threads often had to be constructed from execution segments that did not belong to simple loops, the form of parallelism required by a simple TLS system. In spite of these challenges, very good speedup was managed even on these integer applications.

The data in Table 4-3 demonstrate that the simple transformations are beneficial for both floating point and integer applications. However, the complex ones are beneficial mainly for the integer applications. This was because the execution times of the floating point applications were all dominated by easily parallelizable loops, except for `ammp`.

Therefore, the complex transformations added little or no benefit. In contrast, all the integer applications benefited from the code transformations, and two of the three benefited from complex ones. Notably, explicit synchronization was not very valuable, enhancing performance for just two applications and both times only when used in combination with some other technique. This is for two reasons. First, it does not work well for infrequent violations, as in this case it is better to allow the TLS system to optimistically execute speculative work rather than to force data accesses to be synchronized. Second, many of the violations typically prevented by explicit synchronization are instead better eliminated through the use of complex methods that do not cause serialization.

Likewise, a close look at the incremental performance numbers in Table 4-4 clearly demonstrates that simple transformations parallelize floating point applications well, but that integer applications require complex transformations. In fact, most if not all of the speedup for each floating point application is already realized using only basic parallelization, while the opposite holds true for the integer applications, which often require complex transformations to get any significant speedup at all.

Table 4-4 shows the full-application speedups that were achieved using the three TLS/memory systems. The first is a realistic system, the second assumes a perfect memory system and the third assumes a perfect memory system with a zero-overhead TLS implementation. The average (arithmetic mean) floating point speedup with the real memory system is 2.2, and the average integer speedup is 1.7. Comparison of these speedups with those generated by previous studies on automatic parallelization with TLS is difficult, due to the different architectures, compilers and execution segments utilized. Since TLP is mostly orthogonal to ILP, a rough comparison of speedups can be done using systems with different processor cores and compilers, but different memory systems will still affect the results. With these caveats, a comparison with results from [32][33][37] indicates that the manual parallelization has provided very good parallel performance, well in excess of automatic extraction of TLP at similar thread granularities.

The whole-application speedups for the realistic and perfect memory systems in Figure 4-3 indicate a sensitivity to memory system delay that varies, with some application speedups fairly insensitive to the characteristics of the memory system and others more strongly affected. The perfect memory system results usually provide an upper bound on the performance that can be achieved by scaling the memory system with the number of processors. An unusual exception occurs for `ammp`, because the faster memory system causes a large number of violations on a load-store pair that would otherwise have experienced far fewer violations. Likewise, the results for the perfect memory systems with and without speculation overheads indicate the performance losses caused by the use of a TLS system with speculation software handlers. These results indicate that fully hardware-based speculation would improve performance fairly little for these applications.

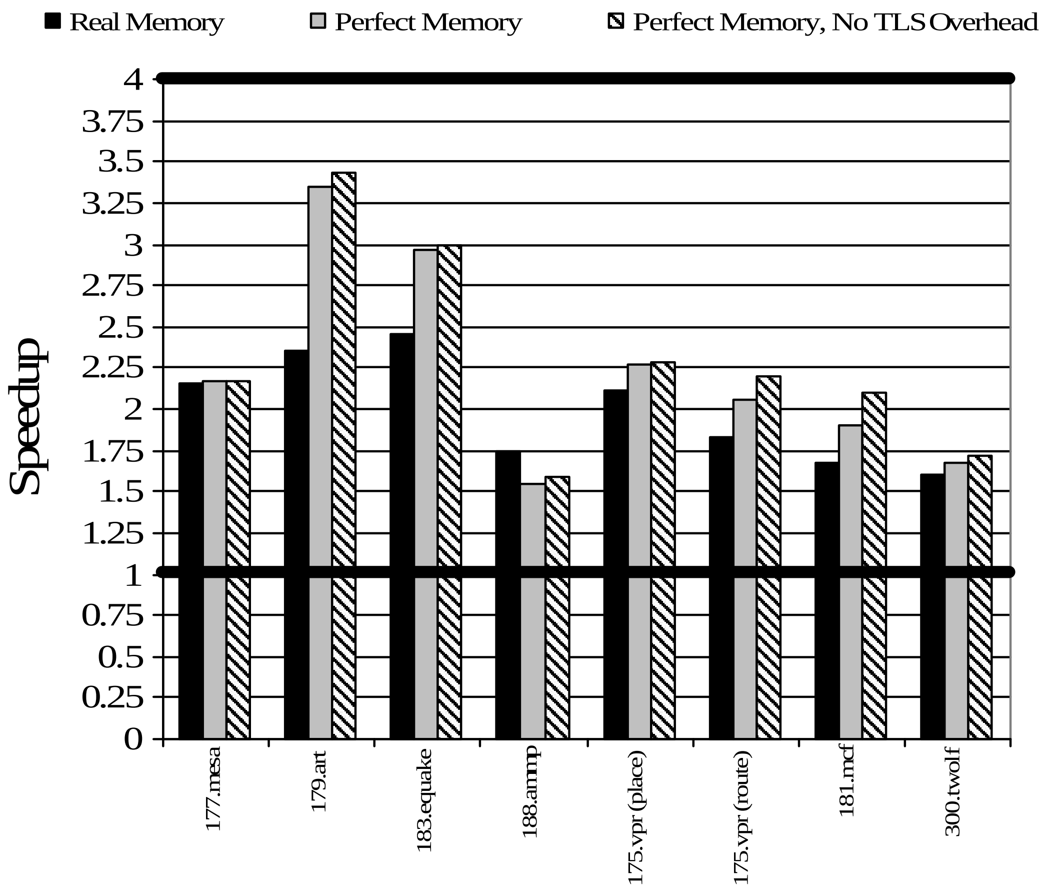


Figure 4-3: Whole application speedups under various memory and TLS models

4.4 Additional simulator results

Table 4-5 characterizes the speculative threads created within each application. These thread lengths represent just the number of instructions in the original, unmodified applications. Specifically, they do not include any of the overheads of executing the TLS software handlers, adapting the applications to expose parallelism or forcing register-allocated variables into the caches to allow the detection of violations. The thread sizes vary considerably, but all are in the range of hundreds to thousands of instructions long. As a result, the parallelism extracted from these applications is thread-level parallelism

Table 4-5: Speculative thread lengths, regions and coverage

Application		Dynamic thread length (instructions)	Number of speculative regions	Percent execution time coverage
CFP 2000	177.mesa	7,800	1	84%
	179.art	450	7	95%
	183.equake	1,300	6	100%
	188.ammp	450	1	86%
CINT 2000	175.vpr (place)	5,100	1	100%
	175.vpr (route)	200	1	97%
	181.mcf	250	6	91%
	300.twolf	700	1	100%
Column mean		2,030	3	94%

and is orthogonal to the instruction-level parallelism (ILP) that could also be extracted from these same applications. Because of the lengths of the threads chosen to parallelize these applications, the extraction of ILP within each thread, for example by an aggressive out-of-order processor, would be expected to have little effect upon the speedups generated by TLS parallelization.

The number of distinct speculative regions is small, demonstrating that for many representative applications a large portion of the total execution time can be parallelized by selecting only a few locations in the code. The parallel coverage of the original sequential execution time is uniformly high, even for the integer applications. Parallel coverage typically increases as more sophisticated transformations are applied to the applications. Similarly, thread lengths can also increase safely as violations are reduced. Because longer threads expose more work to losses from violations, as violations become less frequent, thread lengths can be safely increased, for example, by loop chunking.

Table 4-6: Breakdown of parallelized execution times

Application		Useful	Discarded	Waiting	Overhead
CFP 2000	177.mesa	70%	28%	2%	0%
	179.art	98%	0%	1%	1%
	183.equake	78%	16%	5%	1%
	188.ammp	50%	40%	6%	4%
CINT 2000	175.vpr (place)	63%	36%	0%	1%
	175.vpr (route)	47%	35%	10%	8%
	181.mcf	65%	24%	6%	5%
	300.twolf	55%	23%	20%	2%
Column mean		66%	25%	6%	3%

This reduces speculation overheads and the serialization enforced by the in-order commit at the end of each thread. Correspondingly, applications for which the TLS thread lengths are large and the parallel coverage high tend to have good speedups. But, if either quality is absent, then the performance will usually be substantially diminished. Amdahl's Law explains why coverage must be high, while the poorer performance for small thread lengths can be explained by their correlation with high violation rates and greater speculation overheads and commit serialization.

Table 4-6 provides the breakdown of execution times spent in the parallelized sections of code. The useful work done is generally quite high, the TLS system overhead is negligible, and violations (discarded time) waste over four times as many cycles as load imbalances (waiting time). The remaining violations tended to be due to variables that had frequent accesses distributed amongst the threads, where each access unpredictably caused a violation a small percentage of the times it dynamically occurred. This prevented any benefits from explicit synchronization, because it causes too much execution serialization. Likewise, these qualities would prevent the dynamic dependence detector described in [5] from providing any benefit, although the one proposed in [25] could work, but only if the dependence distances defined in their paper could be used to develop a reliable dependence predictor for these specific dependences. Dynamic load imbalance, due to size-mismatched threads that must wait to be committed in order, is the source of the waiting losses. Both `vpr (route)` and `twolf` show large losses due to load imbalances. This is especially a problem for applications that have been parallelized with small thread sizes.

Part of the useful work done includes the execution of the additional instructions required for the parallel transformations and to support the interprocessor communication and control. In general, the programmer must make a choice between the cost of supporting each additional transformation and the cost of the violations that occur from not using it, instead. These extra instructions limit the maximum speedup, even though for these benchmarks the losses due to the extra work were fairly small.

4.5 Programmer effort required

Table 4-7 provides an indication of the programmer effort required for the parallelization of these benchmarks. It lists the number of lines of code added and the total number of hours spent analyzing, parallelizing and debugging each application. While the hours required are highly dependent on the capabilities of the programmer, these data provide at least an order-of-magnitude gauge of programmer effort involved, and no better metric is apparent. We only counted lines of code that were new and unique, or at least

Table 4-7: Lines of code added to parallelize applications

Application		Original lines	Lines added	Percent added	Prog. hours required
CFP 2000	177.mesa	61,343	20	0%	33
	179.art	1,270	140	11%	24
	183.equake	1,513	130	9%	18
	188.ammp	14,657	130	1%	107
CINT 2000	175.vpr	17,729	160	1%	102
	181.mcf	2,412	120	5%	165
	300.twolf	20,459	320	2%	112
Column mean			146	4%	80

substantially changed. We did not count lines of code that were changed or added to implement the automatic base parallelization, i.e. we did not count lines of code that were effectively replicated from the original application or lines of template code that were inserted purely to support the simulator.

The number of lines of code added remains fairly small and constant across applications, almost always less than two hundred. For these applications, the number of lines required has little correlation with the size of the application. However, this may not hold true for larger, more complex applications, which may require the parallelization of more speculative regions, i.e. loops. The number of hours required to parallelize each application was also quite small, in comparison to the number of hours that were originally required to develop them. This strongly supports our claim that manual parallelization with TLS allows programmers to code for a uniprocessor target in a straightforward way, and then with minimal effort port the entire optimized application to a TLS CMP platform to realize good parallel performance.

TLS parallelization depends primarily on an application's algorithms and source code and the TLS and memory systems, rather than the processor architecture. Therefore, the final parallelized application should port easily to other CMP systems that support loop-based TLS and explicit synchronization. For equivalent speedups, they should have low interprocessor communication delays, low speculation overheads and similarly sized caches and write buffers.

Instead, if a TLS CMP system has very different delays, overheads or cache or write buffer sizes, this could necessitate a modification to the way in which applications should be parallelized for it. A TLS with smaller delays and overheads and larger caches and write buffers would be expected to be able to utilize the same speculatively parallelized code (i.e., the same thread formulation) for each application and gain a roughly equivalent or better speedup. The applications, however, could be parallelized anew using different thread lengths in order to take advantage of the more powerful capabilities of this TLS CMP.

However, if a TLS CMP system with larger delays or overheads were to be utilized, the applications with shorter TLS thread lengths would be expected to be most negatively affected. Use of such a CMP system could necessitate a different parallelization of these applications to use longer thread lengths to reduce the effects of the larger delays and overheads. In contrast, a TLS CMP with smaller caches or write buffers could require the use of smaller thread sizes. This could be the case for applications with large thread lengths, in order to prevent the stalling which could occur due to the TLS system’s being unable to buffer the speculative state for each thread that cannot yet be committed.

However, these are only general expectations. Whether a specific application would need to be parallelized differently for a different TLS CMP is highly dependent on the specific data access patterns of the application, and cannot in general be determined purely from the length of the threads that were used to speculatively parallelize it.

Concerning the effect on TLS parallelization of the number of processors in the CMP, a CMP with fewer processors will generally not require significant code changes in order to run efficiently. But a CMP with more processors may necessitate modifications to the parallelization employed so that the application takes advantage of the larger number of processors available. This would be useful to examine in future research.

This chapter has provided a guide to the thread-level parallelism that exists within a variety of the SPEC2000 benchmarks and the methods by which it can be extracted. It has also discussed the data on parallel performance and on the programmer effort required for parallelization that was generated as a result of conducting this research. This leads to a discussion in the next chapter of obstacles to using TLS to extract TLP, which in turn leads to listing simple programming guidelines for uniprocessor programmers that allow uniprocessor code to be more easily ported at a later date to a chip multiprocessor with hardware support for thread-level speculation.

5 Observations and Conclusions

This chapter opens with a discussion of what hinders and what helps the porting of uniprocessor applications to a TLS platform. Based upon this, guidelines are provided for uniprocessor programmers to follow to enable their applications to be more easily ported to a TLS CMP in the future. Finally, a summary is provided of the research presented in this dissertation and some promising opportunities to conduct further research on this subject are listed.

5.1 Hindrances to TLS parallelization

From our experience parallelizing SPEC2000 applications, we observed a number of common hindrances. We will briefly summarize them here, starting with those that are inherent to the application and working down to those that pertain to our specific TLS system.

Many parts of integer applications are inherently difficult to statically parallelize into threads. While speculative pipelining allows for some dynamic adaptation of the way in which code is divided into threads, parallelizing this kind of code may benefit from hardware that provides more support for dynamic thread creation, similar to [4].

Some algorithms within applications interact badly with TLS in general, for example, deeply recursive algorithms with extensive execution at each level of the recursion. This is especially true if the recursion depth is very different at different parts of a tree structure and the tree changes often, as there is no indication of proper places to cut the algorithm into load-balanced speculative threads. Even worse, reuse of global variables during recursion generally obscures all parallelism and completely serializes the subroutine.

Some applications interact badly with our TLS implementation. For example, our choice to use less complex TLS hardware results in longer communication delays between processors than in more tightly coupled TLS architectures. This makes some thread formulations infeasible, because the thread lengths are too short to amortize the TLS overheads. Another problem is applications with iterations that vary greatly in length. They overflow speculative buffers, causing stalls, for long threads and suffer from load imbalance on short threads. While the former problem can be avoided with most applications, the latter is an issue that frequently arises. Methods have been proposed to address this problem [6]. However, speculative pipelining can be used to dynamically redistribute work between threads to conduct adequate load balancing, as shown for `ammp`.

Load imbalance stalls are caused by sudden decreases in thread length, but not sudden increases in thread length. This is because speculative state must be committed in order, so short threads may need to wait for longer, prior threads to complete. Figure 5-1 shows various thread length sequences. Figure 5-1A shows an ideal thread length sequence resulting in no stall time. Figure 5-1B shows that gradual decreases and even sudden increases in thread length also cause no stalls. However, as Figure 5-1C shows, sudden, large decreases in length do cause stall time. Figure 5-1D shows likewise that large variances in thread length can cause stalls.

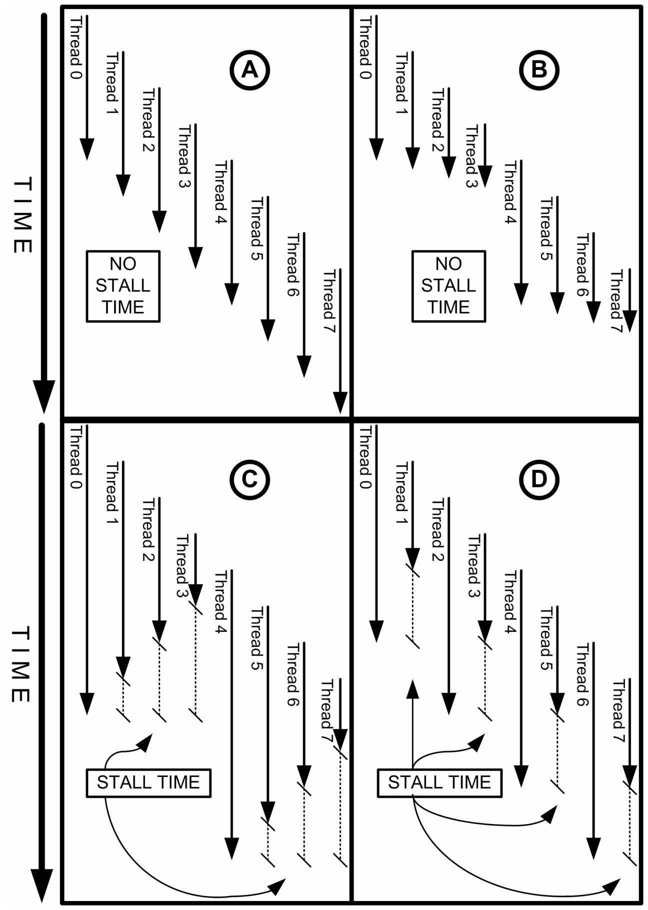


Figure 5-1: Good and bad thread length sequences

5.2 TLS-friendly uniprocessor programming

The manner in which a uniprocessor program is written can profoundly affect the ease with which it can be parallelized. We have identified several simple rules that are easy for a uniprocessor programmer to follow that allow for rapid porting of the final program to a TLS platform. Correspondingly, applications that do not follow one or more of these rules tend to be substantially more difficult to parallelize.

(1) Avoid recursion that returns or modifies values needed by parent calls

Because TLS systems need to commit threads in order, one can only place each call into its own thread if each level of recursion is not dependent upon its children, which must go into “later” threads. This is trivially the case if a recursive subroutine call is the last statement in the parent subroutine, and it does not return a value that is used (tail recursion), but this is rare. The problem is that TLS utilizes a FIFO of threads, while recursion uses a stack of calls. Likewise, some recursive subroutine calls that do not affect their parent or sibling recursive subroutine calls can be parallelized fairly easily. However, the need for good load balancing and the requirement to store speculative state require that the depth of recursion be both limited and similar for all children of a parent subroutine. This is the case for `sort_basket` in `mcf`, which allowed it to be parallelized.

(2) Avoid the use of tailored algorithms for standard purposes

For example, certain applications benefit marginally from complex sorting algorithms tailored to the characteristics of the data being sorted. Replacing these with calls to standard library sorting algorithms can facilitate their replacement with standard parallelized library calls.

(3) Avoid data structures and algorithms with undesirable communication patterns

Algorithms utilizing binary trees generate data “hot spots” that cause contention and frequent violations. Also, the use of complex sorting algorithms that swap distant array members rather than adjacent neighbors can be much more difficult than simpler algorithms to parallelize or synchronize. Often these complex data structures or algorithms are used to provide optimal uniprocessor performance. Because TLS systems can provide a good parallel speedup on many applications, programmers may wish to avoid the best uniprocessor data structure or algorithm, if it is only a little more efficient than a more easily parallelizable one. This is reinforced by the fact that essentially all future desktop and server microprocessors will be chip multiprocessors.

(4) Avoid unnecessary reuse of variables

Using a single variable for multiple purposes, especially a global variable, can often cause unnecessary violations. For example, the use of queues, stacks or heaps accessed in multiple spots during the same subroutine, but for different purposes, will cause violations when the head and tail pointers or the queue length are updated. Using separate data structures for independent portions of execution eliminates this problem. Likewise, using the queue length to check for an empty queue can cause unnecessary violations every time an element is added or removed. Creating a new Boolean variable for the condition of an empty queue, rather than reusing the integer queue length variable, can prevent this, as we demonstrated with `vpr (route)`.

(5) Encapsulate functionality and avoid global variables

Large, monolithic subroutines and applications that extensively use global variables make the task of identifying the communication patterns in a program very difficult, thereby complicating efforts to determine good separation points for threads.

(6) Avoid unnecessarily serializing algorithms

Algorithms such as random number generators that must pass a seed to the next generator call serialize the algorithm unnecessarily if the number of times the generator will be called within a thread cannot be accurately predicted. These algorithms should be replaced with more distributed versions, instead.

5.3 Summary

Hardware support for thread-level speculation provides the programmer with a powerful tool for the parallelization of applications. Because the hardware conducts dynamic dependence synchronization, the programmer is freed from the task of ensuring the appearance of sequential execution. Instead, the TLS hardware presents the appearance of sequential execution to the programmer, automatically ordering memory accesses. This is much simpler for the programmer, because the programmer is essentially following a uniprocessor programming model. The programmer merely has to specify the memory accesses (variables) that are accessed by multiple processors and for which at least one processor is conducting writes.

Due to the simplicity of letting the hardware enforce the sequential execution model, TLS systems enable a much different approach to parallel programming. While traditional non-TLS parallel programming requires planning for parallelism from the start, TLS parallel programming can realize much of the same parallel speedup without the same degree of advance planning and design. Traditional parallelization requires that all algorithms and data structures be designed for parallelism from the start of writing the program, so that parallel speedups can be generated without unnecessary serialization due to synchronization of shared memory locations that may only occasionally suffer dynamic dependences. Instead, TLS allows optimistic dependence synchronization, enabling execution to occur in parallel even when dependences may occur. Only in the event that they do occur does the TLS system dynamically synchronize and order the memory accesses. This provides speedup in the common case executing threads that do not often contend for memory, a speedup that cannot easily be generated when using the pessimistic static synchronization employed by non-TLS systems, that requires stalling at synchronization points every time a memory dependence could occur between ordered threads.

Non-TLS programmers must invest much greater effort in designing applications from the start to have algorithms and data structures that have few, if any, memory dependences, even if they only infrequently occur. TLS programmers are instead freed to concentrate their efforts on only the memory dependences that occur frequently. Less frequent dependences can be relegated to the TLS system for only a small cost in performance. In fact, for some applications, rather than this relegation to the TLS system resulting in a small cost in performance, it can result in a performance gain, as explained above. This is because most applications, once parallelized, will optimally not statically synchronize all dependences, but will instead leave the less frequent dependence violations to be detected and corrected by the TLS system. This allows TLS systems to realize performance benefits from the ability to speculate past potential dependences that do not in fact often occur, which is not the case for conventional, pessimistic, statically-determined synchronization.

This key difference between TLS and non-TLS programming enables a completely different approach to parallel programming. Instead of designing applications for parallelism from the start, applications can be written for a uniprocessor target first, and then ported with fairly little effort to a TLS multiprocessor platform to realize a substantial parallel speedup. As traditional parallel programming often requires significant modification of uniprocessor code, relatively few uniprocessor applications are worth parallelizing. With TLS, the hurdle for parallelization is reduced greatly, vastly increasing the number of applications that are amenable to parallelization and worth the effort to be parallelized.

The ease of parallelization is increased not only by requiring less redesign of applications. It is also greatly increased because of the elimination of data races in parallel programming. Having the TLS hardware ensure the ordering of accesses precludes data races from occurring, provided the programmer specifies which memory locations are shared. As debugging data races is arguably the most challenging task in parallel programming, while identifying shared memory locations is not, this also dramatically simplifies parallel programming. Not only is debugging simplified, but the programmer also does not need to have a good understanding of the memory access patterns and the algorithms in the application. When parallelizing with TLS, the programmer is freed from the initial major hurdles of redesigning data structures and algorithms, followed by extensive multiprocessor debugging. Instead, for the base parallelization, correctness is guaranteed if the programmer can simply specify which data is shared. The programmer is free to immediately begin optimizing parallel performance, and this only optionally need be done. On the other hand, the non-TLS programmer is instead required to possibly do an initial redesign and in all practical cases will need to do extensive debugging, before getting to even a base parallelization. Only then can a non-TLS programmer start to do the optional optimization of performance. For parallelizing legacy code written by other programmers, the ability to move almost immediately into a debugged parallel version with little understanding of the code being parallelized is an enormous benefit.

Because of the ease of TLS parallel programming for these reasons, the programmer can utilize a substantially new approach to parallel programming. Instead of designing programs for parallelism from the start or substantially redesigning a uniprocessor algorithm, applications can be written much more simply for a uniprocessor target and then rapidly and fairly easily ported to the TLS multiprocessor with no advance planning or design. This is simply not possible to do for most applications using the current methods of parallelization, i.e. without hardware support for TLS programming.

While rapid parallelization is possible for far more applications using a TLS system, even more applications can be parallelized and even better speedups generated if programmers follow a few simple guidelines for exposing parallelism when writing the original applications for a uniprocessor system. These guidelines are not very restrictive, and do not require nearly the effort of designing parallel data structures and algorithms for traditional parallelization. However, they do allow parallelism to be more easily detected and extracted by a TLS system, either manually or even automatically. Rather than requiring a programmer to design for parallelism, these guidelines simply attempt to prevent a programmer from obscuring parallelism that is already present in the applications they are writing. In a way, these guidelines are concerned more with preventing a programmer from doing harm than requiring a programmer to do good.

Previous studies have shown that conventional parallelization is improved [22][27] and manual parallelization vastly simplified [29] by the availability of TLS support. To illustrate all the points above, I have first presented two simple examples of application code, and the way in which they can be parallelized with TLS. I provided a short description of the process by which a TLS programmer profiles, analyzes and parallelizes an unfamiliar legacy application. By using basic-block-level, butterfly profiles and run-time violation performance monitoring, the programmer can rapidly locate parallelism and identify variables that limit parallel performance. I then described the way in which the TLS manual programming is done and three of the most useful manual TLS code transformations: complex value prediction, data structure /algorithm changes and speculative pipelining.

These techniques were then applied to several applications in SPEC CPU2000 to assess the performance and difficulty of using TLS on well-known processor benchmarks. I provided a detailed look at the parallelism in a number of SPEC2000 applications. For each application, I described the specific location of this parallelism as a roadmap to other researchers who may wish to utilize this information for their own purposes. I also told what are the key impediments in each application that block extraction of the parallelism. Finally, I discussed the methods we used to overcome these impediments, and I provided performance results for each of these parallelized applications.

While `equake`, `art` and `mesa` are relatively simple to parallelize and can be done via automatic parallelization, the rest of the applications benefit from the expertise of a programmer. In `ammp` this is due to having the parallelism evenly divided between two levels of a nested loop, and having complex load imbalances based upon the outcome of a conditional statement. `Vpr (route)` also exhibits complex load balance issues based upon a conditional branch. In addition, it contains frequently updated variables that are utilized as both integer and Boolean variables. Splitting these uses into two separate variables enhances the parallelism. Redesigning the recursion and shortening the critical paths in parts of `vpr (route)` also enable its parallelization. `Vpr (place)` suffers from artificial serialization due to a random number generator, as does `twolf`. Finally, several of the parallelized applications benefit from synchronization to eliminate a number of preventable violations.

While simple (automatic) transformations were useful for many applications, complex transformations were able to provide further large performance benefits. This was especially true for integer applications, some of which would have experienced no significant speedup with only automatic parallelization. This research provides evidence that real-world applications can be parallelized with very little effort with manual TLS programming.

My experience shows that TLS can dramatically reduce the programmer effort required for application parallelization, while yielding performance gains similar to, if not exceeding, those obtainable using conventional manual parallelization. This supports the assertion that the use of TLS enables a new approach to parallel programming. In this paradigm, the majority of the programming effort can focus on conventional single-threaded application design, with a relatively small effort at the end to port the application to a multiprocessor platform using manual parallelization with TLS.

With the strong movement of mainstream computing toward single-chip multiprocessors, the potential exists for TLS support to be added to future CMPs. This would simplify the process of parallel programming and enable higher performance on applications with parallelism that is difficult to extract using static methods. With this in mind, I have provided broad guidelines to uniprocessor programmers on how to design programs that will port easily to TLS CMP platforms. While this was done specifically with TLS platforms in mind, all of these guidelines also facilitate porting applications to non-TLS parallel platforms.

In conclusion, by providing this detailed explanation of the parallelism in several SPEC2000 applications, I have shown that significant parallelism can be extracted using TLS, even from several integer applications. Furthermore, I have shown the way in which this can be done manually, with the hope that these examples will help inform future efforts on automated TLS parallelization.

5.4 Future research

The use of TLS for aiding manual parallelization holds great potential for allowing significantly more applications to be parallelized than can be done either without TLS or with only automated TLS parallelization. To further this opportunity, the following research would be valuable. More difficult integer applications within SPEC2000 should be parallelized, to learn more about code transformation techniques that can be utilized and to gain more insight into the limitations to the extraction of TLP. For this purpose, it would also be valuable to provide better tools than simple basic-block application profiles to assist the programmer in finding promising regions of TLP. Some research on this has already been done, but much more remains to be completed to provide the TLS parallel programmer with a simple way of finding the best spots at which to parallelize. In particular it would be helpful to be able to see visually the dependences in the code, both prior to and following various code transformations to expose the TLP. Research should be done to determine how to automate the manual code transformations that I have utilized. While it is not currently possible to automate many of them, at least some of them would be amenable to automation, at least in certain constrained situations. Finally, as more research is conducted into what limits the extraction of TLP from applications, more precise guidelines could be provided to uniprocessor programmers to enable them to create applications that would port to TLS platforms and exhibit much better speedups with even less effort.

References

[1] V.S. Adve, et al., “An integrated compilation and performance analysis environment for data parallel programs,” Supercomputing 1995, San Diego, California, pp. 1370-1404, Nov. 1995.

[2] L. Barroso, et al., “Piranha: a scalable architecture based on single-chip multiprocessing,” Proc. 27th Annual Intl. Sym. on Computer Architecture (ISCA), Vancouver, Canada, June 2000.

[3] B. Blume, et. al, “Restructuring programs for high-speed computers with Polaris,” Proc. 1996 ICPP Workshop on. Challenges for Parallel Processing, pp. 149-161, Aug. 1996.

[4] M. Chen and K. Olukotun, “The JRPM system for dynamically parallelizing Java programs,” Proc. 30th Annual Intl. Sym. on Computer Architecture (ISCA), San Diego, CA, pp. 434-445, June 2003.

[5] G.Z. Chrysos and J.S. Emer, “Memory dependence prediction using store sets,” Proc. 25th Annual Intl. Sym. on Computer Architecture (ISCA), Barcelona, Spain, pp. 142-153, June 1998.

[6] M. Cintra, J. Martínez and J. Torrellas, “Architectural support for scalable speculative parallelization in shared-memory multiprocessors,” ISCA-27, Vancouver, Canada, pp. 13-24, June 2000.

[7] M. Cintra and J. Torrellas, “Eliminating squashes through learning cross-thread violations in speculative parallelization for Multiprocessors,” Proc. 8th Intl. Sym. on High-Performance Computer Architecture (HPCA), Cambridge, Massachusetts, Feb. 2002.

[8] J. Clabes, et al., “Design and implementation of the POWER5 microprocessor,” IEEE Intl. Solid-State Circuits Conference (ISSCC), San Francisco, CA, Feb. 15-19, 2004.

[9] F. Gabbay and A. Mendelson, “Using value prediction to increase the power of speculative execution hardware,” ACM Transactions on Computer Systems, vol. 16, pp. 234-270, Aug. 1998.

[10] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam, “Maximizing Multiprocessor Performance with the SUIF Compiler,” IEEE Computer, December 1996.

[11] L. Hammond, et al., “The Stanford Hydra CMP,” IEEE Micro, pp. 71-84, Mar.-Apr. 2000.

[12] L. Hammond, M. Willey and K. Olukotun, “Data Speculation Support for a Chip Multiprocessor,” Proc. 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, California, Oct. 1998.

[13] “International technology roadmap for semiconductors,” hyperlinked to http://public.itrs.net.

[14] S.W. Keckler et al., “Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor,” ISCA-25, Barcelona, Spain, pp. 306-317, June 1998.

[15] D. Kwon, S. Han and H. Kim. "MPI Backend for an Automatic Parallelizing Compiler," 1999 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '99), p. 152, 1999.

[16] P. Kongetira, “A 32-way multithreaded SPARC® processor,” Hot Chips 16, Stanford, California, Aug. 22-24, 2004.

[17] K. Krewell, “AMD vs. Intel in dual-core duel,” Microprocessor Report, Scottsdale, AZ, July 6, 2004.

[18] M. S. Lam and R. P. Wilson, “Limits of Control Flow on Parallelism,” Proceedings of the 19th Annual International Symposium on Computer Architecture, May 1992.

[19] D. Lammers, “Intel cancels Tejas, moves to dual-core designs,” EETimes, Manhasset, New York, May 7, 2004.

[20] K.M. Lepak, G.B. Bell, and M.H. Lipasti, “Silent stores and store value locality,” IEEE Transactions on Computers, vol. 50, pp. 1174-1190, Nov. 2001.

[21] S.W. Liao, et al., “SUIF Explorer: An Interactive and Interprocedural Parallelizer,” Proc. Sym. PPOPP 1999, pp. 37-48, Atlanta, Georgia, Aug. 1999.

[22] J.F. Martinez and J. Torrellas, “Speculative synchronization: applying thread-level speculation to explicitly parallel applications,” Proc. 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, California, Oct. 2002.

[23] C. McNairy and R. Bhatia, “Montecito - The next product in the Itanium® Processor Family,” Hot Chips 16, Stanford, California, Aug. 22-24, 2004.

[24] B.P. Miller, et al., “The Paradyn Parallel Performance Measurement Tools,” IEEE Computer, 28(11):37-46, Nov. 1995.

[25] A. Moshovos, S.E. Breach, T.N. Vijaykumar, G.S. Sohi, “Dynamic speculation and synchronization of data dependences,” ISCA-24, Denver, Colorado, pp. 181-193, June 1997.

[26] K. Olukotun, L. Hammond, and M. Willey, “Improving the performance of speculatively parallel applications on the Hydra CMP,” Proc. 13th ACM International Conference on Supercomputing (ICS), Rhodes, Greece, pp. 21-30, June 1999.

[27] C.-L. Ooi, et al., “Multiplex: unifying conventional and speculative thread-level parallelism on a chip multiprocessor,” ICS-15, June 2001.

[28] S. Naffziger, T. Grutkowski and B. Stackhouse, “The implementation of a 2-core Multi-Threaded Itanium® Family Processor,” IEEE Intl. Solid-State Circuits Conference (ISSCC), San Francisco, CA, Feb. 6-10, 2005.

[29] M. Prabhu and K. Olukotun, “Using thread-level speculation to simplify manual parallelization,” Proc. Sym. PPOPP’03, San Diego, CA, pp. 1-12, June 11-13, 2003.

[30] L. Rauchwerger, N. Amato, and D. Padua, “Run–time methods for parallelizing partially parallel loops,” ICS-9, Barcelona, Spain, pp. 137-146, July 1995.

[31] T. Sherwood and B. Calder, “Time varying behavior of programs,” Tech. Rep. No. CS99-630, Dept. of Computer Science and Eng., UCSD, Aug. 1999.

[32] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry, “Improving value communication for thread-level speculation,” HPCA-8, Cambridge, Massachusetts, Feb. 2002.

[33] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, “A scalable approach to thread-level speculation,” ISCA-27, Vancouver, Canada, pp. 1-12, June 2000.

[34] D. W. Wall, “Limits of Instructional-Level Parallelism,” WRL Technical Note TN-15, December 1990.

[35] A. Zhai, C.B. Colohan, J.G. Steffan, and T.C. Mowry, “Compiler optimization of scalar value communication between speculative threads,” ASPLOS-10, San Jose, California, Oct. 2002.

[36] Y. Zhang, L. Rauchwerger, and J. Torrellas, “Hardware for speculative parallelization of partially-parallel loops in DSM multiprocessors,” HPCA-5., Orlando, Florida, pp. 135-141, Jan. 1999.

[37] C. Zilles and G. Sohi, “Execution-based prediction using speculative slices,” ISCA-28, Goteborg, Sweden, pp. 2-13, July 2001.