ABSTRACT

HAN, LIANG. Using Integrated Compiler and Architectural Techniques to Handle Data Dependences for Thread-Level Speculative Parallelization on Multi-Core Processors. (Under the direction of James Tuck.)

Thread-Level Speculation (TLS) is a promising technique for improving performance of serial codes on multi-cores by automatically extracting threads and running them in parallel. However, the power efficiency as well as the performance gain of TLS systems are reduced due to frequent data dependence violations. Handling cross-thread dependences in the right way is key to improving the efficiency of TLS. This thesis studies efficient dependence handling techniques.

The first study targets a class of dependences with reduction-like patterns. Reduction variables are an important class of cross-thread dependences that can be parallelized by exploiting the associativity and commutativity of their operation. In this work, we define a class of shared variables called *partial reduction variables (PRV)*. These variables either cannot be proven to be reductions at compile time or they may be accessed outside their reduction operation statements and thus violate the requirements of a reduction variable. We describe an algorithm that allows the compiler to detect *PRVs*, and we also discuss the necessary requirements to parallelize detected *PRVs*. Based on these requirements, we propose an implementation in a TLS system to parallelize *PRVs* that works by a combination of techniques at compile time and in the hardware. The compiler transforms the variable under the assumption that the reduction-like behavior proven statically will hold true at runtime. However, if a thread reads or updates the shared variable as a result of an alias or unlikely control path, a lightweight hardware mechanism will detect the access and synchronize it to ensure correct execution.

The second study broadly targets dependence prediction and synchronization. Prior work targeted frequently occurring dependences. But, in our well optimized TLS system, we find violations are caused by many infrequently occurring dependences too. In this work, we propose a new technique that is able to synchronize both frequently and infrequently occurring dependences in irregular tasking patterns without introducing excess synchronization. First, we enlist the help of the compiler to find and mark store-load pairs (SLPs) that generate data dependences. In this way, no training is needed to identify SLPs and irregular patterns are handled. Second, a *hint* operation informs hardware of a possible pending write so that a load only synchronizes when a store to the same address may occur in the future. The compiler schedules the *hint* as early as possible to warn successor threads. Finally, our mechanism wakes up a load as soon as possible using a *release* operation. *Release* is scheduled just after a store and on every path leading away from the hint in which the store does not occur. Together,

they form our proposal called *HiRe*.

We implement our compiler analysis and transformation in GCC, and analyze their potentials on a set of SPEC CPU 2000 benchmarks. We find that supporting *PRVs* provides up to 46% performance gain over a highly optimized TLS system and on average 10.7% performance improvement. And a TLS system supporting *HiRe* suffers only 22% of the violations that occur in our base TLS system, and it cuts the instruction waste rate of TLS in half. Furthermore, it outperforms prior approaches we studied by 3%.

Using Integrated Compiler and Architectural Techniques to Handle Data Dependences for Thread-Level Speculative Parallelization on Multi-Core Processors

by
Liang Han

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2011

APPROVED BY:

Eric Rotenberg	Frank Mueller
Gregory Byrd Co-chair of Advisory Committee	James Tuck Chair of Advisory Committee

DEDICATION

To my parents:
It is always your love that glorify me.

BIOGRAPHY

Liang Han was born in Tai Yuan, Shan Xi province, China, in 1979, and spent most of his childhood in there. Liang received his B.E. from Tianjin University in 2001, and received his M.S. from Chinese Academy of Sciences (CAS) in 2004, under the direction of Dr. Jie Chen. He worked as an assistant research scientist in Institute of Microelectronics of Chinese Academy of Sciences from 2004 to 2006. During his years with CAS, Liang involved in major national projects and published 9 papers. They designed and taped out the first generation of high-performance programable DSP processors in China. Liang received his Ph.D. in Computer Engineering at North Carolina State University in 2011, under the direction of Dr. James Tuck. His overall research focus is in computer architecture, microarchitecture, CPU/SoC/ASIC design, and compiler design, with the main focus on chip multiprocessors (CMPs) and hardware and software supports for speculative parallelization. Liang joined Qualcomm Innovation Center at San Diego, CA in 2011.

ACKNOWLEDGEMENTS

It is many people's help and support that have made this work possible. First, I would like to thank my advisor, Dr. James Tuck, for his help and everything he taught me. From him, I learned not only how to do prefessional research and teaching, but also how to be a responsible and dependable person. His efficiency, persistency, and patience have affected me deeply and will eventually lead me to success in the future.

In my early years in CESR, Balaji Iyer and Chad Rosier were instrumental in guiding me through the difficult process of hacking GCC. ECE-521 and ECE-721 were my favorite courses, Dr. Rotenberg is one of the best instructors I have ever met in my life. Attending his lectures was always joyful. But being his TA was not that easy, because he is very responsible and patient to the students. I like and respect Dr. Byrd and Dr. Mueller very much, not only because of their broad knowledge but also because of their gentleness and patience to students. Shaolin Peng was always my golden partner for many course projects. We had a lot of disagreements on methodology, but we were always among the teams with the highest scores. Xiaowei Jiang and Sanghoon Lee shared a lot of ideas with me – not only on research but also on life. Thanks to George Patsilaras and Rajeshwar Vanka for help on the usage of simulators, tools and softwares, and for discussing research ideas.

I thank my Ph.D. committee for their time and due diligence in their service on my committee. I also thank them for their patient and supportive advice and help.

Last but not least, I thank my parents and friends. Their love always gives me courage and power.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Given the abundance of multi-core architectures and the relatively few programs capable of exploiting parallel architectures effectively, techniques for automatic and semi-automatic parallelization are increasingly important. Parallelizing compilers attempt to decompose a program into threads (or tasks) that can execute in parallel when there are no cross-thread control or data dependences. Despite their many advances, such compilers still fail to parallelize many codes. Examples of such codes are accesses through pointers, subscripted subscripts, inter-procedural dependences or input dependent access patterns.

To move beyond the limitations of classical parallelizing optimizations, new parallelizing compilers must look to analyses and transformations that do not rely on conservative assumptions for correctness, but rather consider new ways to speculate on the likely behavior of data and control dependences at runtime. Parallelizing compilers [41, 25, 33, 54] that target architectures with support for dynamic speculation of data dependences [17, 21, 39, 40, 44, 45, 55] have shown promise for coping with these kinds of problems.

1.1 Thread-Level Speculation

Thread-Level Speculation (TLS) is a promising technique to run sequential code in parallel to take advantage of the multi-core architectures. A TLS compiler breaks a hard-to-analyze sequential code into tasks, and speculatively executes them in parallel, hoping not to violate sequential semantics (e.g. [2, 5, 10, 20, 46, 47, 51]). The control flow of the sequential code imposes a control dependence relation between the tasks. This relation establishes an order of the tasks, and we can use the terms predecessor and successor to express this order. The sequential code also yields a data dependence relation on the memory accesses issued by the different tasks that parallel execution cannot violate.

A task is *speculative* when it may perform or may have performed operations that violate

data or control dependences with its predecessor tasks. When a non-speculative task finishes execution, it is ready to *commit*. The role of commit is to inform the rest of the system that the data generated by the task are now part of the safe (non-speculative) program state. There is only one task in the whole system that can be in non-speculative status; all others are in speculative status. Among other operations, committing always involves passing the non-speculative status to a successor task. Tasks must commit in strict order from predecessor to successor. If a task reaches its end and is still speculative, it cannot commit until it acquires non-speculative status.

As tasks execute in parallel, the system must identify any violations of cross-task data dependences. Typically, this is done with special hardware support that tracks, for each individual task, the data written and the data read without first writing it. A data dependence violation is flagged when a task modifies a version of a datum that may have been loaded earlier by a successor task. At this point, the successor (consumer) task is *squashed* and all the state that it has produced is discarded. Its successor tasks are also squashed. Then, the task is re-executed.

TLS architectures can discard the state produced by a task and re-start the task thanks to special hardware that buffers all speculative modifications, and a checkpointing mechanism that enables rollback. Discussion of such hardware (e.g. [16, 21, 39, 40, 44, 45]) is beyond this thesis's scope. Note that, thanks to these buffers, anti and output dependences across tasks do not cause squashes.

1.2 Cross-Thread Dependences in TLS

Loads and stores in a TLS program must read and write memory in the same order they would in a serial execution of the program. If a load executes for a memory location before the preceding write to that location, a dependence violation happens. Then the load's thread must rollback and re-execute in the correct order. In addition, if successor threads had already consumed data created by the load's task, then they would have to re-execute on a mis-speculation as well. Although the compiler and hardware are capable of speculating on unknown data dependences, blind speculation is not a good idea, because the power efficiency as well as the performance gain of TLS systems is reduced by frequent mis-speculations.

Avoiding such a rollback is desirable in most cases since it would prevent wasted work in the load's own task and possibly in successor tasks as well. But the story is more complicated than this. In fact, a cross-thread dependence creates a dilemma when executing a load: has the last write to the load's effective address already occurred or has it not? If it has not yet occurred, is it better to synchronize the load or let it speculate? In general, such a decision is non-trivial. Overall, there are a few possible behaviors. (1) The last write has occurred, so the load may proceed. This case occurs much of the time. (2) The last write will occur after the

load, so the load should wait. Or, (3) the last write will occur after the load, but the load goes ahead using a predicted value [22, 23, 26, 33] for either prefetching benefit or parallelism in the event the prediction is correct. This last case is problematic because it demonstrates that some benefit can be extracted from speculating even when a dependence is violated.

What is a right way to handle a cross-thread dependence? Not well answered by current techniques, this question has become a significant impediment for TLS systems.

1.3 Existing Techniques to Handle Dependences in TLS

Existing techniques to handle cross-thread data dependences for TLS include: (1) dependence prediction and synchronization, (2) value speculation, prediction, and precomputation slices, and (3) parallelization techniques to handle dependences with special patterns such as reduction variables.

1.3.1 Prediction and Synchronization

Synchronization, such as signal&wait, can guarantee a dependence by suspending the thread loading from a memory location until that location has been updated by the storing thread. Although this technique can avoid dependence violations in a TLS system, the performance could suffer from two issues. (1) The synchronization time could become a critical path of the program. (2) Over-synchronizations, or unnecessary synchronizations, could slow down the program when the dependence doesn't occur at run-time.

Issue (1) is relatively easier to attack. Instruction scheduling techniques that reduce the critical forwarding path are proposed [36,37,38]. The key idea is to hoist the store instruction and sink the load instruction involving the dependence. To handle issue (2), we really need flexible strategies to insert and manage the synchronizations smartly. This involves the design of efficient dependence predictors.

There are two overall strategies that have been proposed for synchronizing/predicting dependences: store-load-pair (SLP) centric [27, 53] and address centric synchronization [6]. SLP-centric techniques are based on the observation that data dependences usually occur between a small set of loads and stores in a program, and that often a specific store communicates to a specific load. Hence, if these SLPs are identified, then only loads that participate in an SLP need to be synchronized. Furthermore, if their associated store has already occurred, then it likely falls into case (1) in Section 1.2, otherwise it likely falls into class (2). Address-centric predictors first determine which memory locations tend to suffer from violations. Then, any load to such a location is forced to synchronize.

Moshovos *et al.*'s hardware predictor [27] adopts an SLP-centric approach. When a squash occurs, an SLP is stored in a centralized table in the processor, called the Memory Dependence

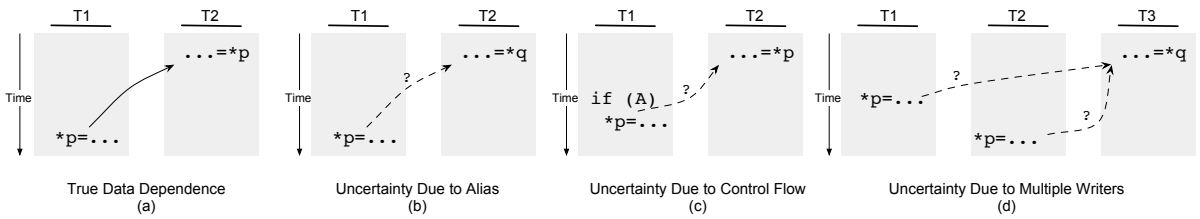


Figure 1.1: Data Dependence Patterns in TLS Programs.

Predictor Table (MDPT), and trained. If a load instruction's PC is found in the MDPT as part of an SLP, then it checks the Memory Dependence Synchronization Table (MDST) to see if its matching store has already occurred. If it has, the load accesses memory and continues executing. If the store has not occurred yet, then the load stalls to wait for it. If the SLP synchronizes properly, the entry for the SLP in the MDPT is re-enforced by incrementing a saturating counter. If the SLP fails to synchronize because the store never occurred, then the counter in the MDPT for the SLP is decremented. Over time, this mechanism learns which SLPs are common and synchronizes them. For example, Figure 1.1(a) is a case detected and handled by this predictor, assuming it occurs frequently. However, it may over-synchronize loads when their store never occurs or when the load and store occur to different memory locations (Fig. 1.1(b)). To mitigate the case in which the store never occurs, they propose using a timeout and tracking dependence distance. However, it is hard to find an efficient timeout time, since the sync time varies greatly from SLP to SLP. (Figure 1.2 gives average sync time in our study.) Also, this predictor will not work well in Figure 1.1(c) since the store may not occur frequently enough to train the MDPT.

Besides being applied to synchronization, hardware dependence predictor can also be used to fast forward the value communicated between a store and a load. In Speculative Memory Cloaking [28], the PCs and other information of a SLP are saved in a hardware table once a dependence are detected via the hardware predictor. The store instruction, if reached again, will then forward its address and value into its entry in this table. And, the load instruction will then speculatively use this value from the table entry when it is reached again and predicted to hit on this dependence. A verification will validate or squash this speculating load afterwards.

Zhai *et al.* [51, 53, 54] provide a compiler based approach that schedules signal&wait instructions at SLPs to force synchronization between tasks. Using high-speed core-to-core communication queues, they send the address of the store and its value to the synchronizing load. If the load's address matches the store address, the sent value is used. Otherwise, the load proceeds by accessing memory. This scheme forces the load to wait until the store executes. If the store and load are accessing different addresses, then the load stalled for nothing, as is

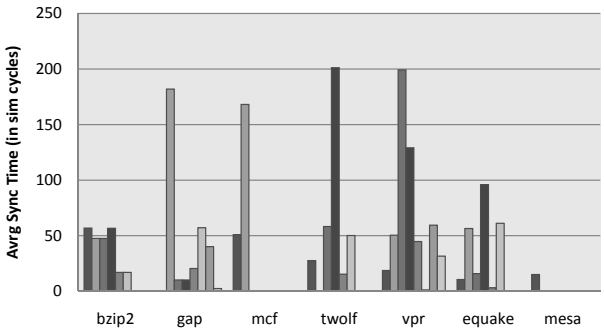


Figure 1.2: Average Synchronization Time in SPEC-2000.

the case in Fig. 1.1(b). Furthermore, it is necessary that communication is guaranteed between the store and the load. For example, in Fig. 1.1(c), the signal must be placed before the `if` condition. However, in general, such placement is not always possible with irregular task patterns.

Finally, Cintra and Torrellas [6] propose an address-centric scheme called Stall&Release and Stall&Wait. If a memory location is frequently involved in a dependence violation, then Stall&Release forces a load to wait until a write occurs to an address, and Stall&Wait forces a load to wait until the load becomes safe. Stall&Release would work in Figures 1.1(a)-(c) if `p` usually has the same address (or one of a few addresses). Stall&Wait will work in Fig. 1.1(d) where there are multiple writers. However, if the locations involved in the dependence change frequently, then this approach will not catch many dependences.

1.3.2 Value Speculation and Precomputation Slices

Value Speculation is also a common technique for eliminating data dependences [22, 23, 26, 33]. Rather than waiting for the last write, a prediction is computed (either in hardware or software) for the likely value of that write. In many cases, the same value already stored in memory is written back. This is known as a silent store [22], and in all the analysis and experiments we perform, we already assume silent stores cause no dependence violations.

Value predictors work best when dependences exhibit a predictable value stream or cannot be synchronized without a large delay. If the value stream is predictable, then simple predictors, like last value predictors or stride predictors, are possible [26]. However, this technique is not always efficient alone, because (1) value updating pattern is not always predictable, (2) a single wrong prediction among the multiple input values can squash the whole thread. Generally, it is combined with other dependence handling techniques.

Precomputation slices (p-slice) [33] can be used to predict or compute input values of a

speculative thread, so that some dependences can be broken and fewer violations happen. The p-slices must calculate the values accurately but not necessarily correct, because the underlying TLS architecture can still handle mis-speculations. This allows the compiler to use aggressive optimizations and reduce the overhead. This technique is more efficient than value prediction to handle input values without a regular pattern. However, this technique is efficient only for small, highly accurate slices. Otherwise it suffers the high cost of big slices or squashes due to the wrongly fed values. If prediction saves a large sync cost and can afford to spend cycles on an accurate prediction, then precomputation slices can be used, as done in Mitosis [33].

These techniques are compelling, and the compiler we are using employs value prediction for some frequently occurring dependences with good value locality. These are usually induction-like variables and function return values.

1.3.3 Reduction Variable Parallelization

Some techniques handle dependences with special patterns such as reduction variables. Special schemes, such as privatization and parallelization, are applied to take advantage of the natures of these special patterns.

Reduction variables are an important class of loop-carried dependences in programs. They are characterized by an expression of the following form inside a loop: $r = r \bigotimes expression$, where r is the reduction variable and *expression* is a value computed independently from r. $\bigotimes$ is assumed to be commutative and associative. Furthermore, r is not used or defined anywhere else in the loop. Even though the computation results in a loop-carried (and likely cross-task) dependence, it can still be parallelized by exploiting the commutative and associative properties of the operation to avoid a cross-task dependence. Each thread can accumulate part of the computation privately; only when all threads are finished do they synchronize and calculate the final sum. Scientific codes exhibit these patterns in abundance and are parallelizable using this technique. However, it is harder to parallelize these patterns when they occur in highly irregular codes, pointer-intensive applications, and general purpose code with library and cross module function calls because the classical definition of reduction variable is too restrictive. However, by relaxing the definition of reduction variables and relying on additional compiler and speculation support, better parallelization is possible.

The work in this area focused on a variety of issues including detection of reductions [1, 4, 30, 15], parallelization/scheduling strategies [12, 31, 43, 42, 35, 49, 50], speculative approaches [34, 8, 7, 54], and architectural support [13].

LRPD test [34] is proposed as a way of overcoming the limitations of static analysis. Instead of requiring a complete static analysis, some disambiguation tests were delayed until runtime. Their scheme works by first creating private versions of each array participating in the reduction

computation, and calculating read and write sets for each array at runtime. Their technique is able to detect if the operations were in fact reductions during the actual execution, even though static analysis could not verify it either due to control or aliasing. Refinements of the LRPD algorithm [8] aim to increase coverage and reduce overhead of this approach.

Hardware support for reductions has been proposed in the past that extends beyond support for efficient synchronization primitives. These schemes include PCLR [13] proposed by Garzaran *et al.*, and UPAR[55] proposed by Zhang *et al.*

1.4 Major Proposals of This Thesis

1.4.1 *PRV*: Partial Reduction Variable Parallelization

The first work of this thesis targets a type of dependence with a special pattern. In this work [18], we formulate a model for a class of dependence patterns that we call *partial reduction variables* (PRV). We call them *partial* because they do not necessarily fit the pattern or *all* of the requirements of reduction variables as defined above. They do exhibit the load-operate-update pattern on at least one path, but they may not exhibit this pattern on all paths. We found that *PRVs* are 3 times more frequently occurring than *RVs* in SPEC CPU 2000 applications. Given the frequency of these variables, it is important to consider hardware and software mechanisms that can exploit parallelism from them. We relax the requirement that all potential uses of the reduction variable can be analyzed. This allows us to label a variable or memory location as a PRV despite must- or may-aliased references elsewhere in the code. Furthermore, we describe how to parallelize *PRVs*, and propose a novel architecture that supports *PRV* parallelization as part of a system with Thread-Level Speculation.

Overall, this work makes the following contributions:

- Defines a new class of variables called *partial reduction variables* and characterizes their presence in SPEC CPU 2000 applications.
- Describes an algorithm to detect *PRVs* automatically and validate manually marked *PRVs* to promote programmability.
- Identifies the needed hardware and software support for *PRV* parallelization.
- Evaluates the impact of *PRV* parallelization with our proposed architectural support on a set of SPEC CPU 2000 applications, and find that supporting *PRVs* provides up to 46% and on average 10.7% performance gain over a highly optimized TLS system.

1.4.2 *HiRe*: Hint & Release Synchronization Scheme

The second work of this thesis targets dependence prediction and synchronization. While existing dependence prediction and synchronization mechanisms cover important behaviors, they struggle to catch a broad set of dependence patterns. Importantly, infrequent dependences are not well supported. Hardware predictors must first learn a behavior, and this cannot easily happen with medium to low probability dependences. For example, the MDPT will un-train when a load occurs a few times without synchronizing with a store. Also, compiler-forced synchronization is unwise if the dependence is not guaranteed to occur with high frequency since the stall cycles serve merely as overhead. Hence, infrequently occurring dependences are disadvantaged in these proposals. Also, these techniques have trouble determining when or how long a load should stall. *They often synchronize loads even when a dependence will not occur.* It is difficult to determine when the store has not yet occurred versus when it will never occur. Using a timeout can be effective for some SLPs, but it must be carefully tuned and provides no real assurance that the store will not occur after the timeout period.

Our goal is the design of a system that achieves the same or similar capabilities as these predictors while robustly covering a larger set of dependence patterns. In particular, our design directly targets both *infrequent* and *frequent* dependence violations. Furthermore, we build support for determining how long a load should wait directly into our strategy. Altogether, we develop a new strategy that is more effective at preventing mis-speculation as well as over-synchronization.

In this work, we propose a new technique that is able to synchronize both frequently and infrequently occurring dependences in irregular tasking patterns. For the infrequent dependences, we optimize for that case in three ways. First, we enlist the help of the compiler to find and mark stores that generate dependence violations on occasion. In this way, no training is needed to identify SLPs and irregular patterns are handled. Second, rather than synchronize every time the load in such a pair executes, our mechanism only synchronizes a load when it detects that a likely store to the same address may occur in the future. We achieve this using a *hint* operation that informs hardware of a possible pending write. The compiler schedules the *hint* as early as possible to warn successor threads. Third, our mechanism also provides a way to wakeup the load and let it resume execution as soon as the store *does or will not* occur. We achieve this with the *release* operation. *release* is scheduled just after a store and on every path leading away from a hint in which the store does not occur. Together, they form the *Hint* & *Release*, or *HiRe*, mechanism.

Overall, this work makes the following contributions:

- Proposes a new data dependence synchronization technique called *HiRe*, that provides novel synchronization properties in the context of speculative parallelization.

- Describes the architectural and compiler support needed to support *HiRe*.
- Evaluates *HiRe* on a set of SPEC 2000 and find that *HiRe* suffers from only 22% of the violations that occur in our base TLS system, and it cuts the waste rate of TLS in half. Furthermore, it outperforms prior approaches we studied by 3%.

1.5 Organization of This Thesis

The rest of this thesis is organized as follows. Chapter 2 presents PRV and the schemes to parallelize it on a TLS system. Chapter 3 presents the HiRe schemes and how it is used to improve data dependence prediction. Chapter 4 describes the methodology, and Chapter 5 shows evaluation of the proposed schemes. Chapter 6 compares with related work. Chapter 7 concludes the thesis.

Chapter 2

Speculative Parallelization of Partial Reduction Variables

Reduction variables are an important class of loop-carried dependences in programs. They are characterized by an expression of the following form inside a loop: $r = r \bigotimes expression$, where r is the reduction variable and *expression* is a value computed independently form r. $\bigotimes$ is assumed to be commutative and associative. Furthermore, r is not used or defined anywhere else in the loop. Even though the computation results in a loop-carried (and likely cross-task) dependence, it can still be parallelized by exploiting the commutative and associative properties of the operation to avoid a cross-task dependence. Each thread can accumulate part of the computation privately; only when all threads are finished do they synchronize and calculate the final sum. Scientific codes exhibit these patterns in abundance and are parallelizable using this technique. However, it is harder to parallelize these patterns when they occur in highly irregular codes, pointer-intensive applications, and general purpose code with library and cross module function calls because the classical definition of reduction variable is too restrictive. However, by relaxing the definition of reduction variables and relying on additional compiler and speculation support, better parallelization is possible.

In this work, we formulate a model for a class of dependence patterns that we call *partial reduction variables* (PRV). We call them *partial* because they do not necessarily fit the pattern or *all* of the requirements of reduction variables as defined above. They do exhibit the load-operate-update pattern on at least one path, but they may not exhibit this pattern on all paths. Furthermore, we relax the requirement that all potential uses of the reduction variable can be analyzed. This allows us to label a variable or memory location as a PRV despite must- or may-aliased references elsewhere in the code. Furthermore, we describe how to parallelize *PRVs*, and propose a novel architecture that supports *PRV* parallelization as part of a system with Thread-Level Speculation.

This chapter is organized as follows: Section 2.1 defines *PRV* and characterizes their occurrence in SPEC CPU 2000 applications; Section 2.2 describes the compiler algorithm for detecting *PRVs*, and Section 2.3 describes how to parallelize them; Section 2.4 discusses the architectural support added to enable *PRV* parallelization.

```
     for( ... ) {

          for( ... ) {
               // alias with delta_vert_cost
1 ---------->  *costptr += ABS( newx - new_mean ) ... ;
          }

          ...

          // tmp_rows is global array
          rowsptr = tmp_rows[net] ;

          // rowsptr aliases with delta_vert_cost and *costptr
2 --------> for( row = 0 ; rowsptr[row] == 0 ; row++ ) ;
          ...

          // alias with *costptr and delta_vert_cost
3 --------> tmp_missing_rows[net] = -m ;

          // alias with *costptr
4 --------> delta_vert_cost += ...
     }
```

Figure 2.1: The new_dbox_a Function Taken from 300.twolf.

2.1 Partial Reduction Variables

A *reduction* is kind of loop recurrence that can be parallelized on a multi-core (or multiprocessor) system easily using conventional hardware. A reduction variable has the following form inside a loop: $r = r \bigotimes expression$. Here, $\bigotimes$ is a suitable reduction operation, and expression is independent of r. There are additional constraints placed on r. It cannot be read or written in any other statement in the loop. With these restrictions in place, the computation of r can be parallelized by exploiting the associativity and commutativity of the $\bigotimes$ to group expressions so that there are no cross thread dependences.

Reduction variables are an important case to handle when parallelizing code, even irregular codes like SPEC CPU [32, 54]. However, the restrictions placed on a reduction variable can be

prohibitive in integer or other highly irregular codes. In particular, the requirement that an RV cannot be read or written outside of the reduction operation prevents the inclusion of many variables. There are many reasons that prevent inclusion. First, conservative pointer analysis results in the appearance of possible reads or writes outside of the reduction statement that are unlikely to happen at run-time. Second, cross-module or library calls that cannot be fully analyzed must be treated conservatively as an update to any RV that may escape to that call. Finally, rarely executed code paths result in a known read or write outside of the reduction statement that will usually not occur at runtime. While the compiler is not able to parallelize these cases, they may behave like reductions at runtime. Instead of letting these opportunities pass by, we want to take advantage of the associativity and commutativity of these operations. Since they do not conform to strict definition of a reduction variable, we call them Partial Reduction Variables (PRVs).

Part of our inspiration for pursuing PRVs came from this loop in the `new_dbox_a` function in `dimbox.c` which is part of 300.twolf in the SPEC CPU 2000 suite. There are two variables (labeled 1 and 4) that have clear reduction patterns in the loop shown in Figure 2.1 (N.B. some code has been omitted). It would appear from inspection that they are ideal candidates for traditional RV optimization. There appear to be no other reads or writes to the variables in the loop, and the operation type is addition. However, aliases are present in this loop (shown at 2 and 3). The aliasing write at 3 may be a definition of one of the potential RVs, and the aliasing read at 2 may use one of the potential RVs. The presence of either one is enough to disqualify both variables. However, even if 2 and 3 are removed, the problem remains since 1 and 4 may alias each other. To preserve correctness of the program at compile time, neither variable can be labeled as an RV since an alias may occur among the two RVs or with other variables in the loop.

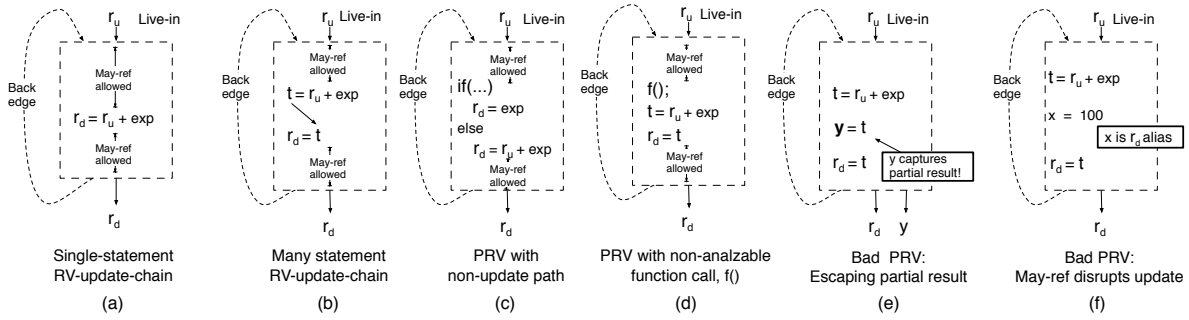


Figure 2.2: Examples of Allowed and Dis-Allowed *PRVs*.

In the remainder of this section, we will provide a definition of *PRV* and will characterize the frequency of *PRVs* in the SPEC CPU 2000 applications.

2.1.1 Definition of Partial Reduction Variables

To encompass the behavior shown in *twolf* we will relax some of the restrictions on an RV to create a formal definition for Partial Reduction Variable. However, first, we specify the definition of RV we will use for the remainder of the article.

A variety of techniques have been explored for detecting reduction variables [1, 4, 30, 15]. Our approach builds on the body of work that searches for specific patterns in the data dependence graph (DDG) that indicate a reduction variable. A reduction operation forms a cycle in the DDG that begins with the load of the reduction variable (*RV-load*), includes a series of true dependences that perform the operation, and updates the reduction variable (*RV-store*) . The cycle is complete because the last store feeds back to the first load along a back-edge in the control flow graph (CFG). For convenience, we will call this an *RV-cycle*. For the computed *RV-cycle*, we identify a sub-graph from the *RV-load* to the *RV-store*, called the *RV-update-chain*. In order to detect a qualified *RV-cycle*, a few other properties are enforced: (1) none of the intermediate computations in the cycle can propagate to other variables, (2) there are no reads or writes of the reduction variable outside of the cycle, and (3) the chain of operations in the cycle form an associative and commutative operation. Note that this definition allows multiple *RV-update-chains* in the *RV-cycle* as long as they all must update the same variable using the same reduction operator. In addition, we tolerate *RV-update-chains* on multiple paths (in a sub-loop or in both an if-then and if-else statement) as long as all paths through the *RV-cycle* form a sequence of valid updates.

For PRVs we start with the same definition as RV, but we relax two properties, (1) and (2), of our previous definition. First, we allow both explicit and implicit references to the PRV outside of the *RV-cycle*. This means that reads or writes of the PRV are allowed that are inconsistent with the reduction operation. These reads and/or writes may come from any of the sources described earlier: aliased memory references, un-analyzed control paths that reference the PRV, or known references on analyzed paths. Even though the reads or writes may be implicit or explicit, we will refer to them all as a *PRV-may-ref*. Second, we relax the requirement that all paths through the *RV-cycle* form a sequence of valid updates. We must relax this property in order to allow a *PRV-may-ref* outside of an *RV-update-chain*. However, it also gives us considerable freedom when labeling *PRVs*. Instead of restricting the static control flow patterns, our definition is now flexible enough to identify reductions that may occur at runtime.

We will place some restrictions on where a *PRV-may-ref* occurs in the code. The *RV-update-*

chain covers the full set of paths that begin with an *RV-load* and end with an *RV-store*. We allow a *PRV-may-ref* along any path incident with *RV-cycle* except for those in the *RV-update-chain*. These restrictions are not arbitrary, rather they are introduced to reduce the complexity of our proposed support or to improve performance. The full rationale for these restrictions may not be apparent until we describe our entire system.

Figure 2.2 presents several examples of our definition. In all of the charts in this figure, the outlined box represents a loop, r_u is live-in, r_d is live-out, and there exists an *RV-cycle*. Fig.2.2(a) shows single-statement *RV-update-chain* that makes a suitable *PRV*. In the case of Fig.2.2(b), the *RV-update-chain* is a sequence of instructions. When performing RV detection on an intermediate representation of a program after optimization, it is more likely to find such a chain than a single statement as in Fig.2.2(a). Also, no *PRV-may-refs* occur inside the *RV-update-chain*, but they are allowed elsewhere in the loop. In the case of aliased references, this is reasonable since they may not actually alias dynamically. Fig.2.2(c) shows a case in which the *RV-update-chain* takes one control flow path but the other is just a write. This is allowed as a PRV but not as an RV. We hope to capitalize on the dynamic behavior of the PRV when its operation along the frequent path is like a reduction. Fig.2.2(d) has a call to a function that has not been analyzed by the compiler either because it is in a separate module or a library. If the address of r escapes to `f`, we can still mark it as a *PRV* in this region.

Fig.2.2(e,f) show cases that aren't allowed. Fig.2.2(e) shows the case that an intermediate result is propagated to another variable. Fig.2.2(f) shows the case that an aliasing or explicit write is present in the middle of the *RV-update-chain*. Neither of these cases are allowed for either *RVs* or *PRVs*.

2.1.2 Characterization of PRVs in SPEC CPU 2000

We implemented our definitions of *RV* and *PRV* in an analysis pass in GCC 4.3 [14] to assess the frequency of these variables in SPEC CPU 2000 applications. Table 2.1 shows the number of *RVs* and *PRVs* found for each application we analyzed. This is not an upper bound on the number of *PRVs*, but it is the number detected by our algorithm (Section 2.2). The second column shows the number of *RVs*, and the third column shows the number of additional *PRVs* that are detected. Note that the columns are disjoint sets even though a *PRV* is a superset of *RV*.

There is considerable variation of both *RV* and *PRV* presence in these applications. Interestingly, our definition allowed us to classify significantly more variables as possible reductions than before, including the ones in *twolf* that inspired our definition. Based on the average number of *RVs* and *PRVs*, we increased the number of reduction variables considered for parallelization by a factor of 3.

Table 2.1: Frequency of *PRVs* and *RVs* in SPEC CPU 2000 subset.

Application	**RV**	**PRV**
crafty	26	6
gap	48	88
gzip	9	27
mcf	6	2
parser	28	12
perlbmk	14	28
twolf	45	56
ammp	8	17
art	7	2
equake	0	3
mesa	5	156
Average	18	36

2.2 *PRV* Detection

To effectively exploit *PRVs* in integer and irregular codes, we must first detect them. In this section, we will describe our algorithm for automatically detecting *PRVs* on an intermediate representation of the program. However, because our automatic detection pass occurs after many other optimizations, we also describe support for manually marking *PRVs* using pragmas inserted by the heroic programmer. The pragma is a hint not a mandate, and *PRVs* marked using the pragma must be validated by our automatic selection pass. We will describe both features, and how they work together to detect *PRVs*.

2.2.1 *PRV* Detection Algorithm

Our *PRV* detection algorithm is implemented in GCC 4.3 using the Tree-SSA intermediate representation [14, 9]. This is an SSA representation that additionally annotates the SSA web with points-to set information. Each separate points-to set is given a symbolic name, and the definitions and uses of that set are converted into valid symbolic SSA form. Figure 2.3 illustrates such annotation. Fig.2.3(a) shows the original code, and Fig.2.3(b) shows the corresponding CFG in SSA form with the memory name A representing a points-to set. References to A in angled brackets represent a definition or use as appropriate.

The *PRV* detection algorithm looks for an *RV-cycle*. As described in Section 2.1, the *RV-cycle* is a circuit of update operations on a single variable using a single reduction operator. The example in Figure 2.3 has such circuit of updates if you follow the edges from $1 \rightarrow 2 \rightarrow (6,7) \rightarrow 4 \rightarrow 5$. There is a use in block 8 and a definition in block 3 but these do not prevent

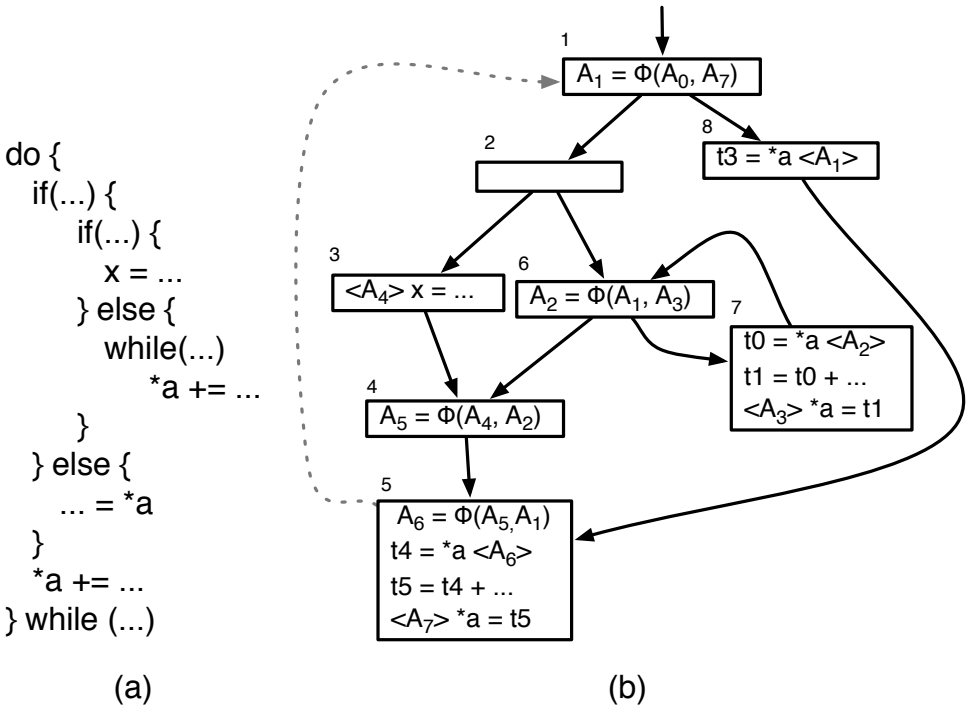


Figure 2.3: Example CFG.

classification as a *PRV*.

Since detecting recurrences on the CFG is well understood [1, 34], the algorithm will be described briefly. For each phi-node in a loop's header block, we will search to see if it's part of an *RV-cycle*. Figure 2.3 shows one such ϕ in the loop header in block 1. We recursively walk backward along the use-def chains of the SSA graph. Therefore, to begin our search, we start with the statement that defines the ϕ's argument along the back-edge of the loop (definition $< A_7 >$). From here, there are a few possibilities that we must handle.

Assignment statement. If it is an assignment to the reference we are searching for, then from this statement we search for an *RV-update-chain*. Detecting the update chain involves following the dependence chains of the assignment backward until finding a *PRV* load. During the traversal, all dependences are searched along use-def chains until they either reach the *PRV* load, a constant expression, a non-aliasing memory load, or exit the basic block. On the return path after reaching a terminating condition, the algorithm collects details about the operation. By merging the results from all operands in an expression and taking the expression operators into account, we can ensure the validity of each update operation. Considering block 5 in

Figure 2.3, it is clear that tracking back along the use-def chain will yield a valid chain quickly: $A7 \rightarrow t5 \rightarrow t4 \rightarrow A6$. After finding a chain, we continue searching backward for a complete cycle from the load operation.

If we fail to validate an *RV-update-chain* or determine the statement to be the definition of a possible alias (as is the case in block 3), we mark the store as a non-updating store and traverse back to its prior dominating definition (the one it kills) and look for an *RV-update-chain* starting there. This step is important since it allows us to overlook definitions caused by aliasing writes, un-analyzed functions, or even explicit writes that are not part of an update.

Conditional ϕ. If we find a conditional ϕ, we branch back along both paths recursively and continue searching for a statement that writes to the *PRV*. In order for the recursive call to return in the affirmative, one of the paths followed backward must include a valid *RV-update-chain*. From the example, the ϕ-node in block 4 would be the next statement encountered. This is a condition-ϕ, and so we search backward along both paths. Since the path through the $6 \rightarrow 7$ inner-loop includes an *RV-update-chain*, the eventual result of analyzing this statement is that a valid *RV-update-chain* was found. If *RV-update-chain* were found along both paths, then the type of reduction operations must match.

Inner-loop ϕ. If an inner-loop ϕ is encountered, the entire detection algorithm is called recursively on the inner loop and analyzed for the particular PRV of interest. In the example, we would apply the same detection mechanism described thus far, but restricted to the inner-loop $6 \rightarrow 7$. Inside the loop, the *RV-update-chain* $A3 \rightarrow t1 \rightarrow t0 \rightarrow A2$ is identified. In addition, the input argument to the inner-loop ϕ that is not from a back edge which is also searched recursively. If an *RV-update-chain* is found both in the inner-loop and along the backward use-def chain, then their operation types must match.

Header ϕ or outside region. The statement found could be the header ϕ we started from. This will not happen initially, but must eventually occur to terminate the cycle. If the header ϕ or a statement outside the loop is reached, the search terminates. As long as one path through the loop back to the header includes an *RV-update-chain*, we mark it as a potential *PRV*.

Final Validation. As the final step, we must verify that no *PRV-may-reference* occurs in any *RV-update-chain* in the *RV-cycle*. For each *RV-update-chain* detected during traversal, we iterate from the use to the definition and search for explicit or implicit (aliasing) references to the *PRV* in between. In addition, we also guarantee that each definition in the *RV-update-chain* is only consumed within the chain. This is trivially computed using the SSA graph.

2.2.2 Manual Detection & Auto-Validation

Manual marking of potential *PRVs* is done using pragmas added directly to the source code. Pragmas allow a heroic programmer the opportunity to suggest good PRVs. We found this support important in order to enable effective task selection for our automatic pass. Automatic parallelization environments often use many heuristics to decide on effective decompositions. With effective hints, these compilers can work better.

The syntax of the pragmas is very similar to OpenMP's reduction annotation. However, pragmas in our system are interpreted as a strong hint but not as a mandate. To validate *PRVs* marked using pragmas, we run the detection pass, as described in the previous section, on the marked loop and variable name. If it does not meet the definition, it is discarded and a warning is given to the programmer. This enables even not-so heroic programmers to suggest *PRVs* without breaking our system.

A key challenge in making this mechanism work is preserving the *PRV* until the *PRV* analysis pass runs. The pragma specifies the *PRV* using its lexical name, but these names are often lost after lowering to an intermediate representation and optimizations have been performed. To ensure that a manually marked *PRV* remains visible to our analysis pass, we replaced each static reference of a specified *PRV* (e.g. *realPRV*) with a volatile reference of equivalent type (e.g. *volaPRV*) since volatile variables are ignored by compiler optimizations that may eliminate, move, or rename memory references. We inserted assignments of *'volaPRV = realPRV;'* before and *'realPRV = volaPRV;'* after the pragma-marked region.

2.3 Speculative Parallelization of *PRVs*

In this section, we discuss the parallelization strategy for *PRVs*, and the kinds of systems that can support it.

2.3.1 Parallelization Requirements

PRV parallelization involves lowering the *RV-update-chains* found in the loop to run efficiently in parallel. If the *PRVs* behave only like classical *RVs*, then their parallelization can be highly efficient, and the strategies used can be similar to those that support *RVs*. Each reference to the *PRV* in an *RV-update-chain* is replaced with a reference to private variable of the same type. Before executing the parallel region, the private is initialized to a neutral value suitable for the *RV* operation. Either during or after the parallelized region, the private is accumulated with the *PRV*. Loop unrolling can be used to lengthen the region and increase the gains achieved from parallelization.

However, in the event that loads and stores to the *PRV* **do** occur outside of the *RV-update-chains*, the first requirement is that they have to be detected. Detection is not trivial since the compiler does not mark each *PRV* access. For explicit reads or writes, the compiler should handle them directly. For aliases analyzed by the compiler, runtime disambiguation tests can be used. For accesses in un-analyzed control paths, the parallelization environment must detect the access to the *PRV*.

Upon detection of *PRV* accesses outside an *RV-update-chain*, additional support is needed to ensure correct parallelization. The possible behaviors of *PRVs* can be divided into three cases based on the types of *PRV* reference that may occur: stores only, loads only, and a combination of loads and stores.

Stores only. In the case that only stores to the *PRV* may occur outside of the *RV-update-chain*, then care must be taken to ensure that the value of the *PRV* after executing the loop is equal to the last non-updating store accumulated with the private variable updates from all later iterations. This requires preserving the last store and ordering it with respect to all later iterations (and accumulations in the same thread). Fortunately, this does not require synchronization with other threads.

Loads only. In the case that only loads to the *PRV* may occur outside of the *RV-update-chain*, the potential overhead is much higher. When the load occurs, all private values accumulated from prior iterations must be merged with the *PRV* to provide the correct value. This requires that all prior iterations complete their last update to the private variable. If some of those iterations have not yet been scheduled, then the load must wait (or speculate on the loaded value).

Loads and stores. In the event that both a load and store may occur, then a combination of corrective actions are needed for both the load or store. Furthermore, the mechanisms must cooperate because a load must get the accumulated result since the last store. The overhead and efficiency of the mechanism can vary depending on the actual pattern of loads and stores in the loop. If stores frequently precede loads (especially in the same iteration), then the loads need not wait long to accumulate an up-to-date value. However, if loads and stores are far apart, then performance may appear more like the load-only case.

2.3.2 System for *PRV* Parallelization

Aspects of *PRV* parallelization are currently supported by some systems, but none fully support it. If *PRV* selection is restricted to the case of only stores occurring outside of the *RV-update-chain*, then a system like Thread-Level Speculation (TLS) [21, 39, 40, 44, 45] could support it

with few modifications. The latest store to the *PRV* would overwrite all previous stores. When the accumulation is done after the parallel region (or as part of each iteration), the most recent store would be used. However, somehow, the accumulations from iterations prior to the store still must be discarded. Transactional Memory [19, 48] (TM) could work similarly if a partial ordering were imposed on tasks that wrote to the PRV. However, these systems cannot fully support the case of loads or stores by default.

To fully support our definition of *PRV*, we extend a system that implements Thread-Level Speculation to record the link between the private variable and the *PRV*. By exposing this link to the TLS system, we can properly correct loads and optimize the handling of stores. Since TLS naturally supports thread ordering, managing the corrective actions for all task behaviors is straightforward.

Example

Figure 2.4 explains how we support PRV parallelization on a TLS-based system. Fig.2.4(a) shows the original code. The TLS compiler through heuristics or programmer hints chooses tasks for parallelization. Note that the TLS compiler can overlook any dependence due to hardware that will catch any dependence violation at runtime. The pragma at 1 in Fig.2.4(a) suggests to the compiler a potentially good TLS task (the region marked in between 1 and 1'), which includes a PRV named *sum* whose reduction operator is +. At location 2, *sum* operates as a traditional reduction variable, but it is read and written at locations 3 and 4 respectively, which make it a PRV.

Fig2.4(b) shows the transformed codes needed to create parallel tasks in TLS and support for the PRV. The *spawn()* at 1 will create a new thread beginning after the *commit()* at 1'. Similar to how a reduction is handled traditionally, *sum* is privatized as *priv* (as 2, 2', and 2" show). But in order to correctly support the load at 3 and store at 4, additional actions are inserted as 3' and 4'. In the case of the load, the first action is to prevent a squash by waiting to become safe. We do can do this explicitly using the *become_safe()* operation, and this guarantees the task will wait for the final version of sum from its predecessor task before reading it. Next, it updates sum with its accumulated value in *priv* and clears *priv*. After all this is done, it can use the value in *sum*. Also, because we cleared *priv*, additional accumulation into *priv* that could occur within the task will work correctly. In the case of the store at 4, we simply clear *priv* at 4'. At the end of the task, just before commit, *priv* is accumulated into *sum*. To ensure this does not happen prematurely, we synchronize the update using *become_safe()*.

If the compiler can fully analyze *sum* statically and make sure it has no alias, no hardware support is needed and Fig2.4(b) shows the transformed code added by the compiler. Otherwise, if the accesses at 3 and 4 were in fact aliases, then the instructions are not inserted by the compiler, instead hardware should detect the access to sum and take the same set of actions.

This requires the hardware knowing the link between *priv* and *sum* and interpreting the read or write as occurring outside of the *RV-update-chain* (see Section 2.4).

Fig.2.4(c) shows how program correctness is guaranteed at runtime. Time proceeds down the page while tasks to the right are more speculative. Iteration N+1 contains the store at 4. The block to the right of label 4 shows the correcting actions. Note that it stalls until it can load the final accumulated value of *sum* from its predecessor. Iteration N+2 contains the load at 3 and its correcting actions. The synchronization actions needed to bring *sum* up to date do lead to stalls, but these effects are ameliorated by a couple of factors. We expect these synchronization events to be infrequent in PRV tasks selected by the TLS compiler, and even when they do occur, it is better to synchronize than pay the price of a squash.

2.4 Architectural Support for *PRVs*

We add new architectural features to a TLS system to support *PRV* parallelization. Figure 2.5 shows the key pieces of our new architecture: (1) the PRV Lookup Table (PLUT) tracks the mapping from *PRV* to private variable and provides key details for synchronizing updates, (2) the *PRV* Signature stores a hash of the addresses of all the *PRV*'s for quick access, and the *PRV* Controller which implements the detection, correction, and update synchronization algorithm. The Load-Store Queue (LSQ) and Versioned Cache are assumed to be typical with no features specific to our scheme.

Table 2.2: Description of New Instructions.

Instruction	Description
`pair_addr.op.t Rv, Rp`	Pair reduction variable, whose address is in Rv, with the private variable whose address is in Rp. Op describes the operator (addition,multiplication), and *t* indicates whether the *PRV* is an int,float,double, or long long for proper initialization.
`unpair_addr Rv`	Unpair the reduction variable from any private variable.

2.4.1 Support for a *PRV* Access

If a *PRV* has or may have aliases, the compiler alone cannot totally handle it. Detecting and correcting such *PRV* accesses occurs at runtime when a reduction operation is in progress. Before entering a region of code with such a *PRV*, the compiler schedules a `pair_addr` instruction to notify the hardware that a partial reduction operation is under way and links the address of the *PRV* to the private variable holding the partial state. Hardware keeps a record of the *PRV* and private variable in the *PRV* Lookup Table. The hardware must assume that the reduction is underway until it receives an `unpair_addr` instruction. During this window of execution, hardware monitors for illegal accesses to the PRV. It is the compiler's responsibility to insert `pair_addr` and `unpair_addr` and ensure no illegal accesses occur to the *PRV* outside the region. Table 2.2 gives the descriptions of `pair_addr` and `unpair_addr` instructions.

When a `pair_addr` instruction is executed for the first time on a PRV, a new entry is created in the PRV Lookup Table, shown in Figure 2.6. The `pair_addr` instruction allocates an entry in the table, sets the Valid (V) bit, clears the Unpaired (U) bit (described later), sets the Private variable address, and initializes Accumulated Partial Result. The PRV address is also added to the PRV Signature. The signature is similar to a Bloom filter [3] and provides a summary of all the addresses in the lookup table. Searching a signature is much faster than searching a modest size lookup table, so this allows the lookup to be placed off the critical path.

The `unpair_addr` instruction indicates that the pairing of PRV to the private location should be stopped, and sets the *U* bit in the corresponding entry indicating that the private address is no longer paired with the PRV. Also, the value currently stored in the private variable is loaded from memory and accumulated with the Accum field in the PLUT entry. Even though the PRV's address is marked as unpaired, the entry for the *PRV* need not be removed from the PRV Lookup Table immediately. Instead, we allow it to continue monitoring this address since the partial state is preserved in Accum. This policy optimistically delays the memory update until the end of the task when it is safe to do so without causing a squash.

Detecting a Conflict

If there is at least one valid entry in the PRV Lookup Table, all loads and stores will be checked against the signature for a conflict. If no conflict is found with the signature, then the table need not be checked. However, if a conflict with the signature is found, then the PRV Lookup Table is searched for a matching PRV. Upon finding a matching entry in the table, the LSQ is temporarily stalled so that the state of the PRVcan be updated as necessary. If no match is found, then no action is taken and the cache access occurs as usual.

Correcting the State

When a conflict on a *PRV* is detected, the *PRV* Controller will take the necessary actions to fix the program's state. These actions mirror the code inserted by the compiler when it finds a non-*PRV* access. We will explain correcting a load and a store separately, but note that the mechanisms also suffice for the case when both loads and stores occur.

Correcting a Load. In the case of load access to the PRV, the Controller will request the current value stored in the PRV. If the Unpaired bit is empty, it also requests the private variable. Note these requests are equivalent to memory requests made by the processor and are handled in exactly the same way as standard TLS coherence request.

Then, it will accumulate, as determined by the operation kind stored in the PLUT entry, the PRV and the Accum field in the PLUT entry. Finally, it will store the result into the PRV's location in memory and reset the private variable to the correct initial value, depending on the type of variable and reduction operator. At the end of this sequence of operations, the PRV is fully up-to-date and the private variable is reset. Now the LSQ is un-stalled and allowed to complete its load operation on the PRV. When it loads, it will find the correct value in the cache.

Finally, if after correcting a load we discover that the U-bit is set, it is safe to clear the valid bit since the entry no longer contains any partial state. The only reason the entry was still in the PLUT was to delay the *PRV* update as long as possible. But since the update has occurred, no reason exists to keep the entry.

Correcting a Store. The case of a store access to the *PRV* is somewhat simpler. In this case, we are simply updating the current value of the PRV, and we could choose not to stall that update. However, we do stall the LSQ momentarily and insert a store to the private variable to return it to its initial value, and we reset the Accum field in the PLUT entry.

Multiple PLUT entries for the same *PRV* The compiler may aggressively select two *PRVs* in the same loop that may alias. Most of the time, they will not access the same location. However, if two entries in the PLUT are added with the same *PRV* and a different private location, the hardware forces both into a special mode of operation, indicated by the *A* bit in the PLUT. Instead of monitoring for accesses to the *PRV*, we monitor accesses of both privates. If there is a read to the private variable, such read is automatically replaced with a *PRV* read. Similarly, a write to the private is treated as a write to the *PRV*. This will guarantee that the *PRV* is updated correctly with respect to both *PRV* chains.

2.4.2 Synchronizing *PRV* Updates

With the support already described, synchronizing updates is relatively straightforward. When a task has become safe, finished executing all its instructions, and is ready to commit, it scans its PLUT for entries that have its Task Id (TID) and that are valid. For each such entry, it is handled by using the same logic as for correcting a load. This takes care of all actions necessary to bring the *PRV* up-to-date.

2.4.3 Compiler Support

This hardware mechanism requires that the PLUT never overflow, lest a mapping between *PRV* and private variable be lost. To avoid unnecessary complexity in the hardware, we place two restrictions on our technique: (i) we do not allow a *PRV* task to be nested within another *PRV* task, and (ii) we require that the compiler manages the table usage to prevent overflow. In the first requirement, disallowing *PRV*-task nesting ensures that the compiler can determine the number of active *PRVs* in the PLUT through global analysis of a single subroutine. However, this does complicate task selection because this property must be enforced conservatively – even if such a nesting was unlikely to occur dynamically, if it is ever possible, such a nested task cannot be selected. Such a limitation requires selecting among alternatives, and this selection is carried out explicitly by our profiler (see Section 4.1).

The second requirement ensures that a task will never use more entries than available in the PLUT, and this is enforced directly through analysis of each task region. We use a compiler parameter *NumPRV* to limit the maximum static number of PRVs per task. Our empirical evidence suggests that up to 4 static *PRVs* finally survive for an application. So we set *NumPRV* to 4 and use 4 PLUT entries for each core in the evaluation and found this to be sufficient.

2.4.4 Cost Estimation

The estimated per-core hardware cost added to support *PRVs* for a 64-bit system includes a 4-entry PLUT (4 * 34-Byte = 136 Bytes); 1 32-bit signature (Bloom filter with single hash function derived from lower bits of address excluding the two least significant); other control logics needed to control PLUT and utilize existing ALU/FPU during accumulate; and other logics added to support the 3 new instructions (*pair_addr*, *unpair_addr*, and *become_safe*).

The main performance cost added for a store or a non-conflicting load is the delay of a 32-bit signature which is negligible since it can be done in parallel with other logic. However, the cost for a conflicting load is the stalled cycles waiting for the correct value being fed from its predecessor thread (this delay is comparable to that of a synchronization operation), plus the

cycles to accumulate this value to that of the privatized variable. This may seem significant, but it is worth it compared to the squash-and-restart cycles that would otherwise occur.

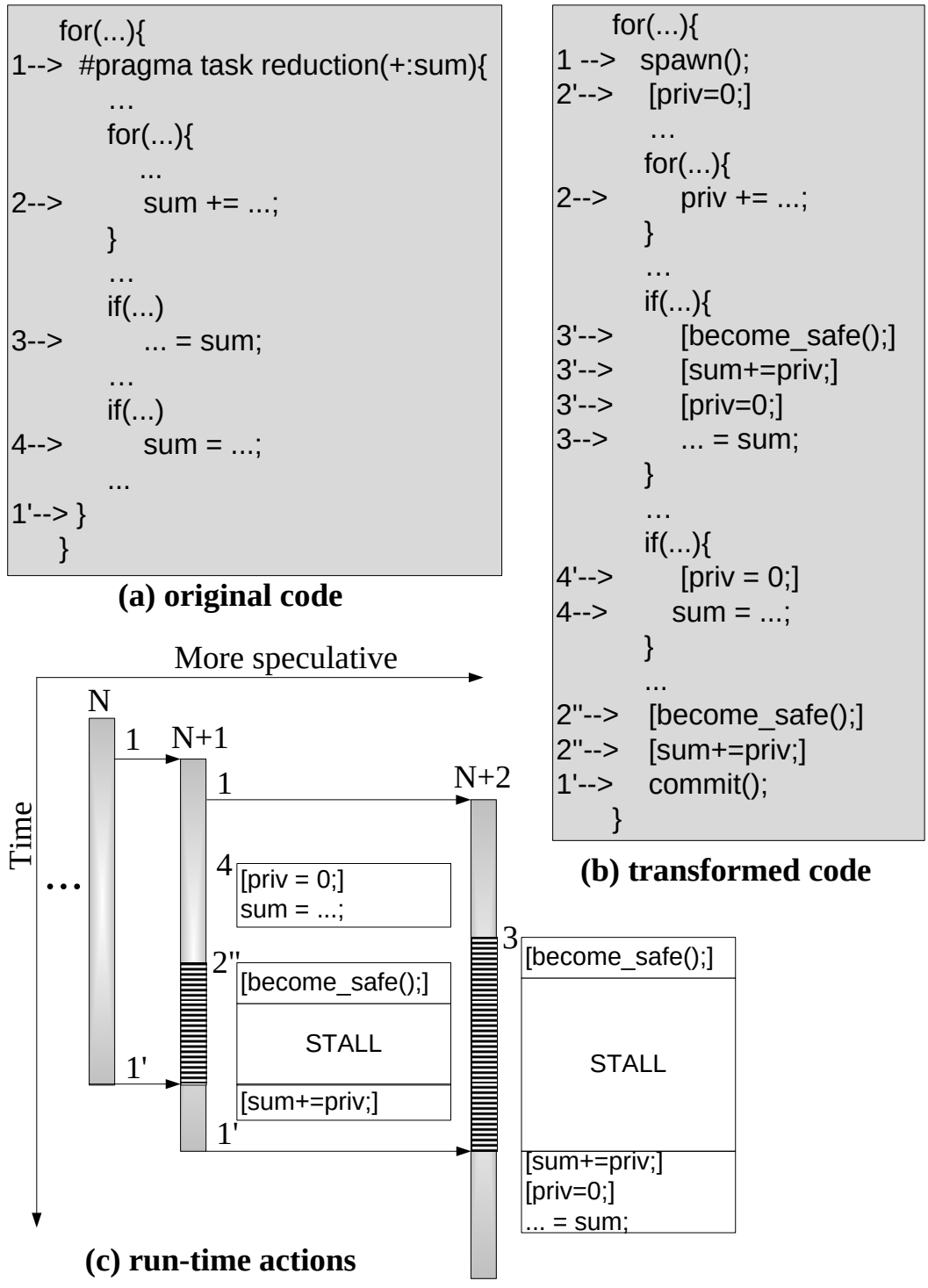


Figure 2.4: *PRV* Parallelization on a TLS System.

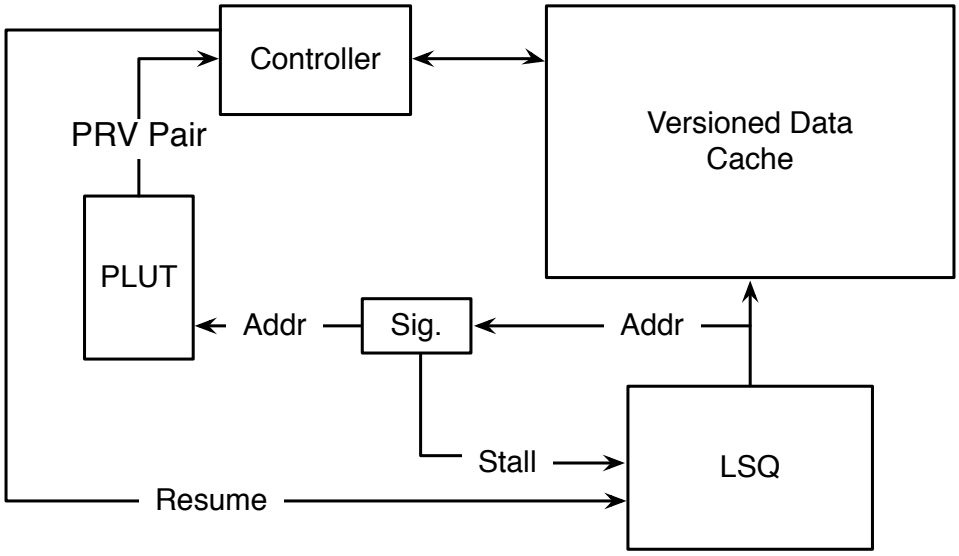


Figure 2.5: Architectural Modifications to Enable *PRV* Parallelization.

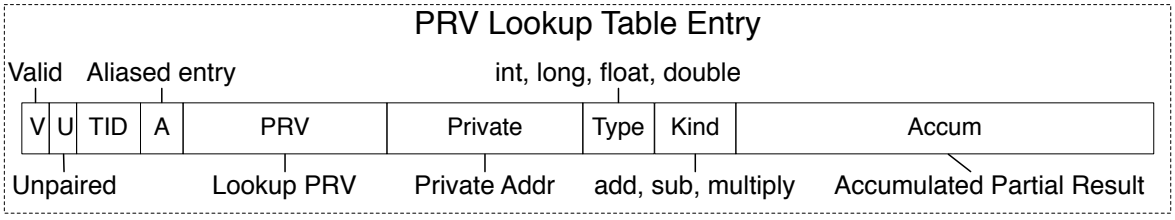


Figure 2.6: Fields in a PLUT Entry.

Chapter 3

Using Hint&Release to Improve Dependence Prediction for TLS

A variety of techniques for data dependence prediction and synchronization have identified common patterns and how to detect them on the fly. For example, some memory locations are updated repeatedly, and by detecting an access to such a location, threads can stall and handle the pending write [6] to prevent violations. Another behavior is that of store-load pairs (SLPs). SLPs can be identified and synchronized when they occur again [27]. Other proposals [53, 54] have exploited SLPs by using the compiler to insert signal&wait that force the load and store to synchronize and communicate their values.

However, these prior works have two major shortcomings. The first shortcoming is that they rely on data dependences to occur relatively frequently. Infrequent ones are hard to handle either because they introduce too much synchronization overhead or they are not detected by hardware as relevant. Ignoring infrequent dependences is detrimental because *good compilers for speculative parallelism are likely to produce code with relatively more infrequent dependences and fewer frequent ones.* Second, prior techniques imagined fairly regular tasking behavior, like loops. However, irregular codes (that you would want to parallelize using TLS or similar techniques) tend to need highly dynamic and irregular tasking patterns [29, 37] made up of loops, function calls, and continuations of loops and function calls. Therefore, simple dependence prediction techniques that rely on frequent easy to anticipate patterns will not work as well in such a system.

In this work, we propose a new technique that is able to synchronize both frequently and infrequently occurring dependences in irregular tasking patterns. Because the dependences are infrequent, we optimize for that case in three ways. First, we enlist the help of the compiler to find and mark stores that generate dependence violations on occasion. In this way, no training is needed to identify SLPs and irregular patterns are handled. Second, rather than synchronize

every time the load in such a pair executes, our mechanism only synchronizes a load when it detects that a likely store to the same address may occur in the future. We achieve this using a *hint* operation that informs hardware of a possible pending write. The compiler schedules the *hint* as early as possible to warn successor threads. Third, our mechanism also provides a way to wakeup the load and let it resume execution as soon as the store *will or will not* occur. We achieve this with the *release* operation. *release* is scheduled just after a store and on every path leading away from a hint in which the store does not occur. Together, they form the *Hint & Release*, or *HiRe*, mechanism.

The rest of this chapter is organized as follows. Section 3.1 provides a high level description of our idea. Section 3.2 describes the compiler algorithms, and Section 3.3 describes the new architectural mechanisms.

3.1 Main Idea: Hint & Release

Our idea is inspired by the strengths of both SLP-centric and Address-centric synchronization schemes introduced in Section 1.3.1. SLP-centric techniques are powerful because they identify specific locations in the code that may trigger a dependence violation. In particular, knowing which store is likely to produce the value needed by a load enables timely and efficient synchronization when the store actually executes. This is a feature worth preserving. Address-centric strategies are powerful in their ability to only stall loads that are accessing a location likely to be involved in a dependence. Stalling only those loads that are accessing a location likely to be written is a feature worth preserving.

Figures 3.1 shows the basic idea of *HiRe* scheme. Rather than trying to synchronize an SLP, *HiRe* identifies a window of execution that contains a store with the potential to trigger a mis-speculation. This window (a.k.a. the sync window) begins at a point in execution when a store is *anticipated* to occur and will be marked with a *hint*. The window extends until a point after the store occurs, called the *release*. The placement of *hint* and *release* is SLP-centric in the sense that is placed around a store for a specific SLP. However, unlike a SLP, the store need not occur for the region to be detected by a load.

Because *hint* and *release* both specify a specific address for synchronization, any load that tries to access the *hinted* address will be forced to stall. Hence, loads are synchronized in an address-centric manner. Once the address is *released*, loads are allowed to proceed with execution again.

The *hint* and *release* instructions delineate the execution of a load into 3 regions, as Figure 3.1 shows. The load will act differently at run time in different regions. In **Region A**, the hint has not been observed and the load will blindly speculate. This region occurs on any non-synchronized load or when a *hint* is not observed in a timely way. In **Region B**, the load

will wait to be released, providing synchronization for the load. We hope most loads that have a late arriving dependence execute in this region. Finally, in **Region C**, the load will proceed with execution since the dependence has been satisfied. From the perspective of the load, there is no difference between **Region A** and **Region C**. However, unlike hardware prediction schemes that do not know how long to stall the load, the transition from **Region B** to **Region C** is explicitly detected.

Because we want to handle dependences that are both frequent and infrequent, we optimize for the infrequent case in three ways. First, we enlist the help of the compiler to find and mark stores in SLPs that may generate data dependences. In this way, no online training is relied upon to identify SLPs on the fly. Second, rather than synchronize every time the load in such a pair executes, the load waits only when a store to the same address may occur, as indicated by the sync window. Finally, our mechanism also provides a way to wakeup the load and let it resume execution using *release*. *release* can be scheduled immediately after the dependent store and on any side-exits on the path from the *hint* to the store. Infrequent dependences are handled because we can identify their region of influence cheaply and only stall loads when they are likely to violate. Finally, this strategy is very tolerant of arbitrary tasking patterns since it makes no assumption on the relative ordering or location of stores and loads in the code.

3.1.1 Design Overview

To work effectively, *HiRe* must accomplish tasks which rely on a mix of compiler and hardware support. A compiler coordinates insertion of the *hint* and *release* to create the desired sync window, and the hardware must track the *hint* and *release* information and stall dependent loads if they are located within the sync window. We briefly consider these goals but leave detailed discussions for the next section.

A compiler will carry out several tasks in order to insert *hint* and *release* pairs. First, it will identify loads and stores that are likely to participate in a data dependence using program profiling. Given that few loads and stores (from 10s to 100s) actually cause violations, this is a relatively simple task through program profiling [27, 53]. For each store, the *hint* must be hoisted early enough in execution that it occurs before a dependent load in a successor. To do this, a backward slice for the store address can be hoisted to an earlier program point along with the *hint* instruction. Finally, the *release* follows the execution of *hint*. Ideally, it will occur just after the store itself or on a side-exit that guarantees the store will not occur.

The hardware is responsible for tracking the *hint* and *release* on any given address and synchronizing dependent loads. Ideally, such support should be cost effective for a large number of addresses and for many concurrent synchronizations. In addition, the *hint* and *release* should not add much overhead. To accomplish these goals, *HiRe* adds bits to the cache state to track

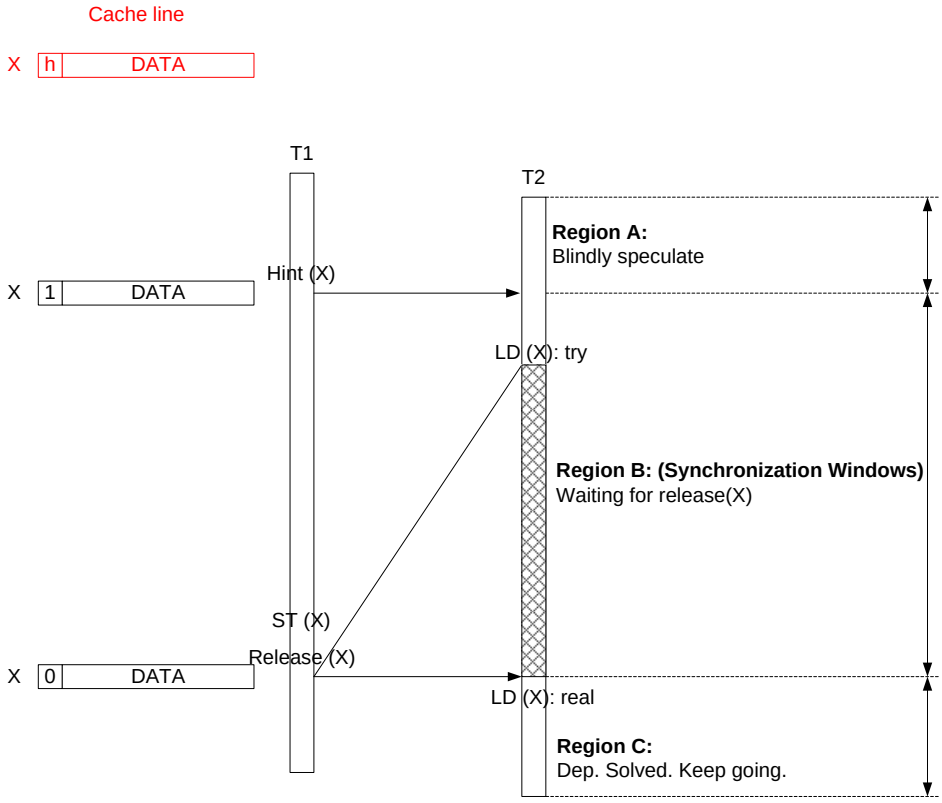


Figure 3.1: The Basic Idea of *HiRe*

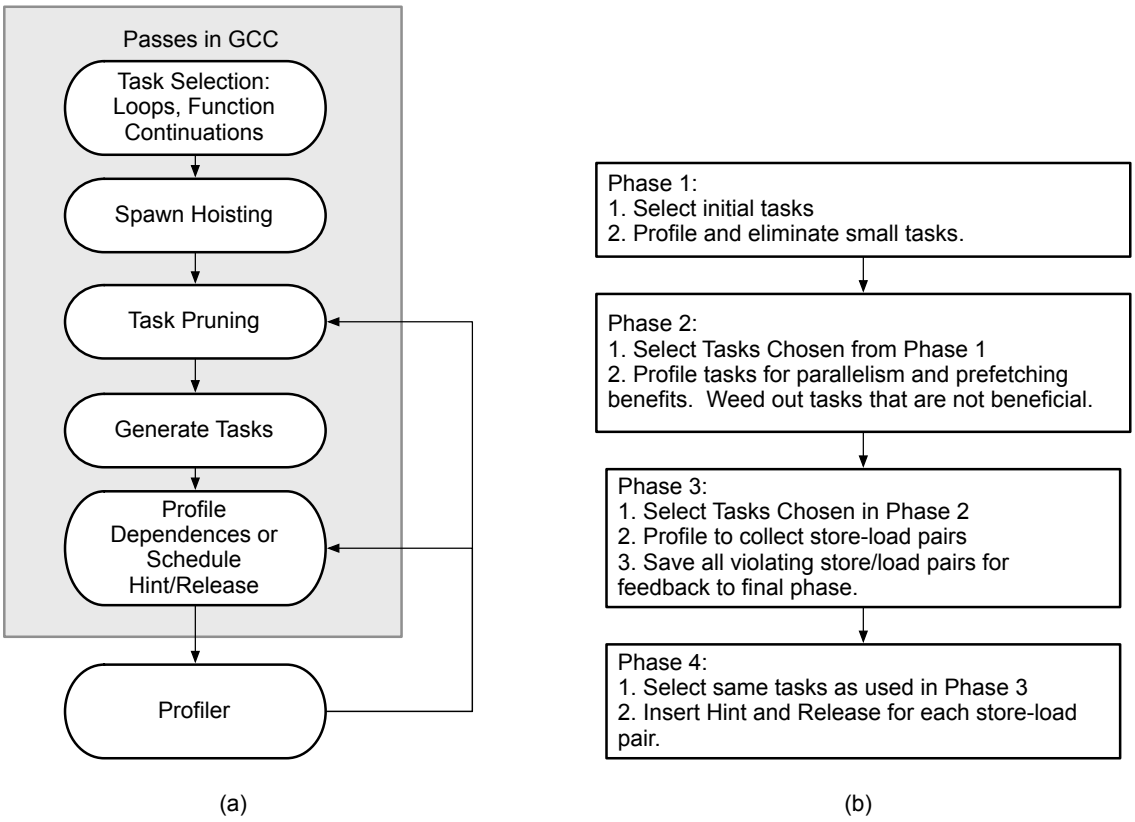


Figure 3.2: *HiRe* Compiler Organization (a), and Compilation Phases (b).

hint and *release* operations per address, as is shown Figures 3.1. Because the information is tracked with cache line state, the TLS coherence protocol can be leveraged to synchronize remote loads. Furthermore, since coherence operations are highly distributed, it can support many concurrent synchronizations. The following section describes these mechanisms in detail.

3.2 *HiRe* Compiler

HiRe's compiler support consists of a set of passes in GCC that perform TLS task selection and place the *hint* and *release* in the code; and, it includes a profiler which collects information about TLS tasks and data dependences for selection of SLPs and tasks. First, we will describe the overall compiler structure and then the specific support for *HiRe*.

3.2.1 Compiler Framework

The overall structure of the compiler is shown in Figure 3.2. Part (a) shows the pass structure added to GCC and the profiler. During compilation, two kinds of tasks are extracted from the program: loops and subroutine continuations. For loop selection, the compiler selects either outer or inner loops, depending on task characteristics. Subroutine-continuations with easy-to-speculate return values are also selected. For each task chosen, a spawn is inserted to initiate execution of the task. The spawn is placed at the earliest control independent location which does not violate any analyzable data dependence. Next, during *Task Pruning*, some tasks which are expected to offer little parallelism are eliminated. In *Generate Tasks*, the *spawn* and *commit* instructions are inserted along with all necessary code to communicate between tasks through memory. In the final stage, the compiler can optionally insert code to profile data dependences or to schedule *Hint* or *Release*.

We adopted a profiling strategy very similar to [24]. The profiler executes the TLS binary sequentially and analyzes tasks according to several key properties: (1) the size of the task (t_{size}), (2) the overlap (parallelism) achieved with its predecessors ($t_{overlap}$), (3) the likelihood of mis-speculation (p_{sq}), and ($t_{prefetch}$) the time saved due to prefetching from a squashed execution. Overall, we keep a task if $t_{size} > t_{szMin}$, $p_{sq} < p_{sqMax}$, and Benefit = $t_{overlap} + t_{prefetch} > 0$. However, these rules are not applied all at once. Figure 3.2(b) describes the three phases of compilation with the specific profiler actions taken in each phase. Now we consider in more detail the three aspects of the compiler that are novel with *HiRe*: selection of store-load pairs, *hint* hoisting, and *release* sinking.

3.2.2 Selection of Store-Load Pairs

To identify store-load pairs that need synchronization, all loads and stores are monitored during profiling. For each dependence violation that occurs, the profiler creates a record for the store-load pair and counts the number of occurrences. The record includes the PC for the store, the PC for the load, and the source and destination task IDs. Even if a load or store occurs in multiple pairs, the information for each pair is tracked separately. We adopt this policy because it favors synchronization of regularly occurring dependences between tasks.

Because *HiRe* is designed to work with both frequently and infrequently occurring dependences, all stores detected that ever occur in an SLP are candidates for *HiRe*. However, the profiler or compiler may choose to discard a pair for a variety of reasons. Table 3.1 shows a prioritized list of reasons why detected SLPs are not considered for *HiRe*. These are practical difficulties (1 and 2) or empirical common sense reasons (3 to 6). The empirical thresholds in 3, 4, and 5 are tuned for our system during previous experiments.

Table 3.1: Reasons some SLPs are not considered for *HiRe*.

Reasons to discard an SLP during Compilation	**Rationale**
1. The profiled store cannot be found at compile-time.	Generated by the back end of the compiler, no way to handle it with confidence.
2. The *hint* cannot be hoisted.	No benefit from *HiRe*.
Reasons to discard an SLP during Profiling	**Rationale**
3. Serial instruction distance > than 500M instructions	Not likely to occur.
4. Task distance between store and load > 8. (for 4-core)	Not likely to execute concurrently.
5. The load occurs in the first 1% of the task's execution time, and no new task is spawned before the load.	No benefit from sync.
6. The SLP is not beneficial	Does nothing but introduce cost.

3.2.3 Hint Scheduling

The Hint hoisting algorithm tries to schedule the Hint instruction as early as possible. The algorithm must overcome the difficulty of precomputing the store's address, and it must choose a location for the *hint* that increases the chances it will execute before the dependent load.

One problem is that the store's address calculation places a limit on the Hint's placement; therefore, we construct a backward slice for the store's address calculation to enable a greater motion. Figure 3.3(a) shows a store and its nearby address calculation. Using a precomputation slice, we can move the Hint a bit earlier (Fig. 3.3(b)). The slice computation algorithm, `compute_slice`, is shown in Figure 3.3(e). It takes as input a statement, an existing slice, and a location, `earliest`, above which we cannot hoist the *hint*. Starting from this statement, it traverses backward in the Program Dependence Graph (PDG) [11], recursively, gathering nodes to add into the slice. During the traversal, it will visit an ancestor at most once and decide how to process it. If the ancestor node is before `earliest`, a phi node, or a node that could cause an exception (i.e. memory load), it can move no further back and it determines the new earliest location which it cannot move beyond. Otherwise the node is added to the slice subject to data dependence constraints. When `compute_slice` finishes its PDG traversal, we can place the slice at `earliest`.

However, the earliest possible location in the serial program is not always the best choice when considering its parallel execution. The dynamic task structure must be considered to ensure that that *hint* actually executes before the load. In Figure 3.3(c), the *hint* is properly placed to execute before the load. However, Figure 3.3(d) shows what happens if we hoist the *hint* before the beginning of Task B. Now, it is actually located at the end of Task A and will occur *after* both the dependent load and the store! Hence, task boundaries must be considered as a limiting factor in *hint* placement. The `get_earliest_loc_for_hint` function determines the nearest task boundary which cannot be crossed. For a loop task, the boundary is placed at the beginning of the loop header. For other tasks, we traverse the dominator tree looking for

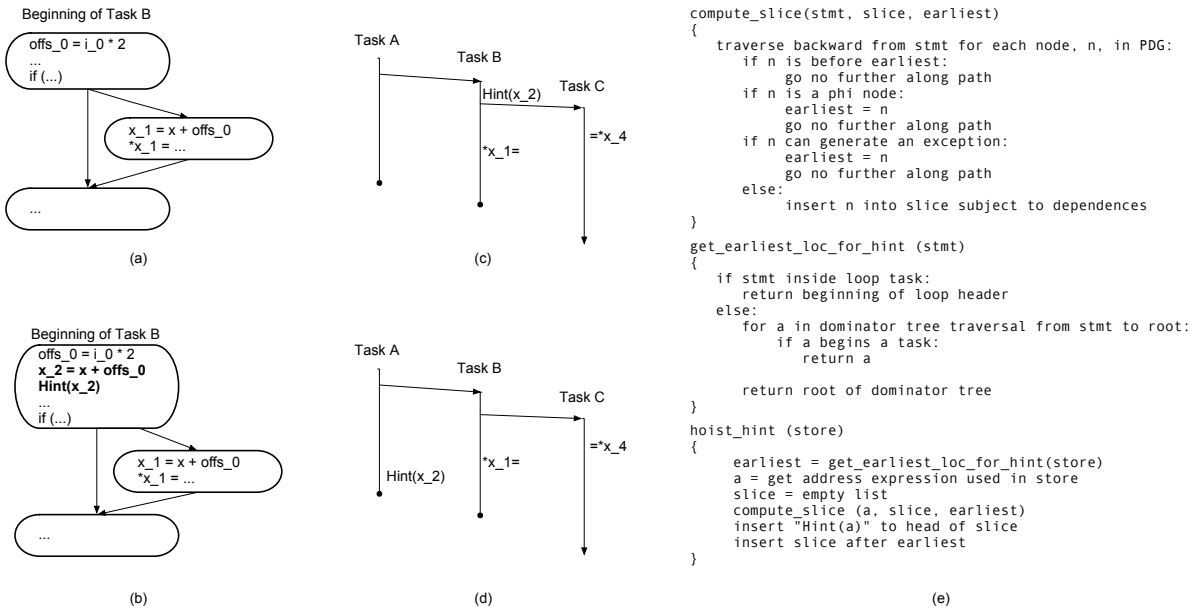


Figure 3.3: *Hint* Hoisting Examples and Pseudo Code.

parent task and stop when one is found. If one is found, the beginning of the task is set as the hoist limit; otherwise, the first block in the function (found at the root of the dominator tree) is chosen.

The `hoist_hint` function pulls all this functionality together. First, it determines the earliest possible location for the *hint* subject to task boundary constraints. Next, it computes the slice for the store address. Finally, the full slice, including the *hint* instruction, is scheduled just after the `earliest` location.

3.2.4 Release Scheduling

The purpose of *release* is to notify any stalled load that it may proceed. It is scheduled immediately following the store instruction and on every side-exit between the *hint* and *release*.

While it would seem that the store could serve as the *release*, this is unwise because the *hint* may calculate a wrong address some of the time. (For example, a function call in between the *hint(&X)* and the store(X) may update a global variable which is an operand of a statement in the slice.) It is important to release the same address that was hinted so that stalled loads on the incorrectly hinted address are released.

We would like to release the load as soon as possible. So, we devised an algorithm that places *release* immediately following the store and at the earliest point on every path that will not pass through the store. This is easily implemented using depth-first traversal of the CFG

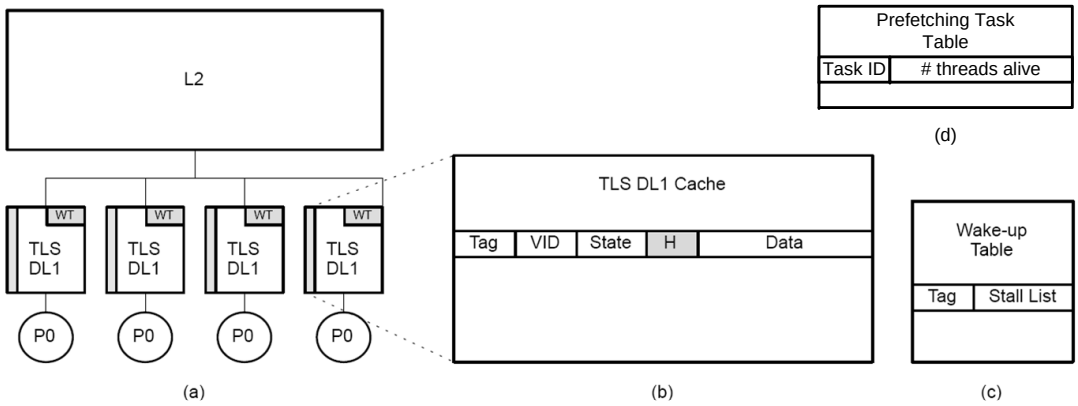


Figure 3.4: *HiRe* Architectural Support.

starting from the *hint* and enumerating all side-exits on the path to the store.

3.3 *HiRe* Architecture

We extend a CMP with a TLS coherence protocol and Versioned L1 data caches, much like [38]. The overall system is shown in Figure 3.4(a). We extend this system to support *HiRe* by adding three new instructions, extending the cache state to support a H bit, and modifying the TLS coherence protocol to stall a load when it encounters a set H bit. The following sections describe the architecture for these extensions in detail.

3.3.1 Hint and Release Support

We add an H-bit to each cache line to record the presence of a *hint*, and the H-bit is reset to indicate that *hint* has been released. Figure 3.4(b) shows the resulting organization for one cache line. It includes a tag, a speculative version identifier (VID) that determines the task, the speculative state for each byte in the line, support for *HiRe*, and the data.

The `hint` instruction executes as soon as its address is resolved in order to update the cache line status as early as possible. However, the `release` instruction waits until all prior store addresses are resolved and only executes after any preceding store. Once there are no preceding stores with the same address, it can execute. This policy ensures that it will follow a prior store to the same line, but otherwise will execute early and allow stalled loads to resume. Both the `hint` and `release` instructions work like a non-faulting prefetch and will not generate an exception if the input address is invalid. Due to aggressive slice construction, such an invalid address is possible.

Most interactions at the task level are kept as simple as possible. When a task completes and commits its data or when a task is squashed, all H bits are reset. They are not passed from one task to another, and they are not preserved across task boundaries. They are associated with the VID in the cache line they are written into. However, some additional interaction is needed to support stalling a load.

3.3.2 Stalled Load Support

In the TLS coherence protocol, when a thread reads an address for the first time, it must identify the most recent writer (in program order) and obtain a copy of the data from that task. We extend this mechanism with an additional rule — if an *executing* predecessor thread has set the H bit for the line, the request to read the line is nacked rather than returning stale data. When the requesting thread receives the nack, it knows to wait. Because such a mechanism is already provided in coherence protocols, adding this support is trivial.

Without any additional changes, we get the following behaviors. After waiting for a predetermined timeout period after a nack, the load is retried. If the same predecessor is found with its H bit cleared or if the same predecessor finished executing without clearing its H bit, the load will check if the dirty bit is set; if so, the load will now succeed since it has found the latest version. Otherwise, it will keep searching on an older predecessor. It is possible that a more recent predecessor had set its H bit during the previous load stall. On detection of such a case, the load will stall for that task's write (this is the multiple writer support). The load will continue to stall and retry as long it finds an H bit set for a running task. If the search reaches the oldest predecessor and still doesn't find any Hint, the load proceeds speculatively. Also, if the stalled task becomes safe, it must proceed since no violation is possible.

Optimizing Load Behavior Using the Wakeup Table

Requiring the load to busy wait is undesirable and inefficient. If we make it retry frequently, we waste power and energy; if it is infrequent, we sacrifice performance by unnecessarily delaying the load. The data in Section 5.2.2 shows that the average synchronization time fluctuates greatly from load to load. It is hard to set a unified time-out timer even for a single application. In order to reduce the wait time for the load, we add a small table near each DL1 cache that detects requests to cache lines with the H bit set. The Wakeup Table is shown in Figure 3.4(c). When such a load request occurs, the requesting task's id number is added to the table in an entry tagged with the requested line. When the H bit is cleared for that line or when the task finishes execution, the stalled task is sent a message to proceed. When the stalled task receives this message, it tries to load again in the method described in the previous section.

It would be nice to avoid re-executing the stalled load yet again by forwarding the entire

cache state immediately. However, such an approach would bypass the TLS coherence mechanism and could lead to an incorrect execution if another task, later in program order, wrote the data in the time since the load's first request. Therefore, it is necessary to check again for the nearest writer rather than forwarding the line.

Since the table will be small, it may not be able to hold all pending hints at all times during execution. Therefore, we support a timeout mechanism. We adopt a naive timeout in which a load waits at most for a fixed period of time before it tries again. We set the interval high enough that will not retry many times while waiting for the wakeup signal.

In an empirical study, we found that four entries is nearly identical to an unlimited-sized wakeup table. Therefore, we do not evaluate this part of the architecture in order to save space.

Optimizing for Prefetching

Prefetching is an important way to boost performance in TLS. However, synchronization tends to delay useful prefetches that would otherwise be beneficial. Recall that some tasks are selected by the compiler for their prefetching benefits even though they are frequently squashed. If we are not careful, the violating loads in these prefetching tasks will be stalled and will prevent the task from performing beneficial prefetches.

To prevent synchronization from happening in such cases, on the one hand, we need prevent loads from stalling when running in a task selected for prefetching. We could achieve this in two ways. First, tasks created for prefetching are spawned using a special opcode that tells the hardware to prefer to speculate. Second, when loads in a prefetching task generate coherence messages, they are labeled with a special bit that tells the coherence protocol to ignore the H bit. In this way, it will obtain a copy of the data quickly even if there is a pending write. On the other hand, once the task has performed the necessary prefetching, it should resume normal synchronization for loads.

However, it is not easy to fully achieve both requirements in a synchronization scheme, because there are generally multiple SLPs in a static task and many loads of these SLPs are accessing different locations in different dynamic threads. If we permit all SLP loads in all dynamic threads ignore sync, synchronization scheme will be totally useless for these tasks. And for many applications these prefetching tasks dominate the performance. So, we should not do prefetching in this way. We propose two schemes to partially perform the prefetching.

(1) If we only permit the first SLP load in all dynamic threads ignore sync for only one time and resume the *HiRe* scheme afterwards, only the L2-miss loads located in between the first LSP load and the 2nd SLP load are beneficial. To achieve this scheme, we just need add one ignore-sync bit for each hardware core. A load will ignore the H-bit only when this bit is set. Any prefetching thread will set this bit on its launching, and any violation in this thread will reset this bit. We evaluated this scheme in our early study and found the performance is

slowed down a little bit from the non-prefetching case. This is because L2-miss loads located in between the 1st and 2nd SLP load are not many and cannot bring much benefit, but the violations which could have been prevented hurts performance more and all dynamic threads will try to prefetch and will very likely violate. We thus didn't adopt this scheme in our final experiments.

(2) If we permit all SLP loads in the first dynamic threads in the system ignore sync to prefetch throughout that thread but don't let any other thread to do so, only variables shared by different dynamic threads of a same static task are prefetched. However, all such shared variables can be prefetched just with a sacrifice of a few first dynamic threads in the system. To achieve this scheme, we introduce a global Prefetching Task Table, as shown in Figure 3.4(d). Each entry records the ID of a static task, and the number of its dynamic threads alive in the system. An entry is dynamically assigned to the first thread of a prefetching task. A task spawn or commit operation increases or decreases its counter. Only the thread that increases its counter from 0 to 1 performs the prefetching. A four-entry table is big enough for most applications, as shown in Table 5.4. In case the table overflows, an entry whose counter is 0 will be kicked out, because it is very likely that the program is not going through this task at this moment. Otherwise, the newest prefetching thread will be ignored. We applied this scheme to our final evaluation.

3.3.3 A Runtime Example

Figure 3.5 shows how the *HiRe* scheme works at run-time. When a load instruction executes in thread C, it checks for any local write first (0). If it is exposed, it checks along predecessor's versioned cache in reversed speculative order (1). When the request reaches core 2, synchronization logic there checks the cache and finds that the H bit of location X is set (2). Entry "&X: thread C" is thus added into wakeup table on core 2 (3). On being notified this (3), core 3 suspends thread C (4).

When *release*(&X) instruction (without updating X) executes later on core 2 (5), it updates cache line by clearing the H-bit (6). When a hinted location is released, the wakeup table on core 2 is checked and thread C is found being suspended on this location (7). Then a release message is sent to core 3 to wake up thread C (8). Once resume, the load instruction re-executes and tries to check previous cache versions again (9). This time, it will not stop on core 2 since neither Hint bit nor Dirty bit is set, which means that no write to location X happened, although thread B thought there would be one (Otherwise, if the dirty bit is set, the read request would consume this value and stop searching further backwards since the latest store has been located.) The request continues to check an older version, say thread A running on core 1 (9). Similar to what happened on core 2 by the synchronization logic (10,11), thread

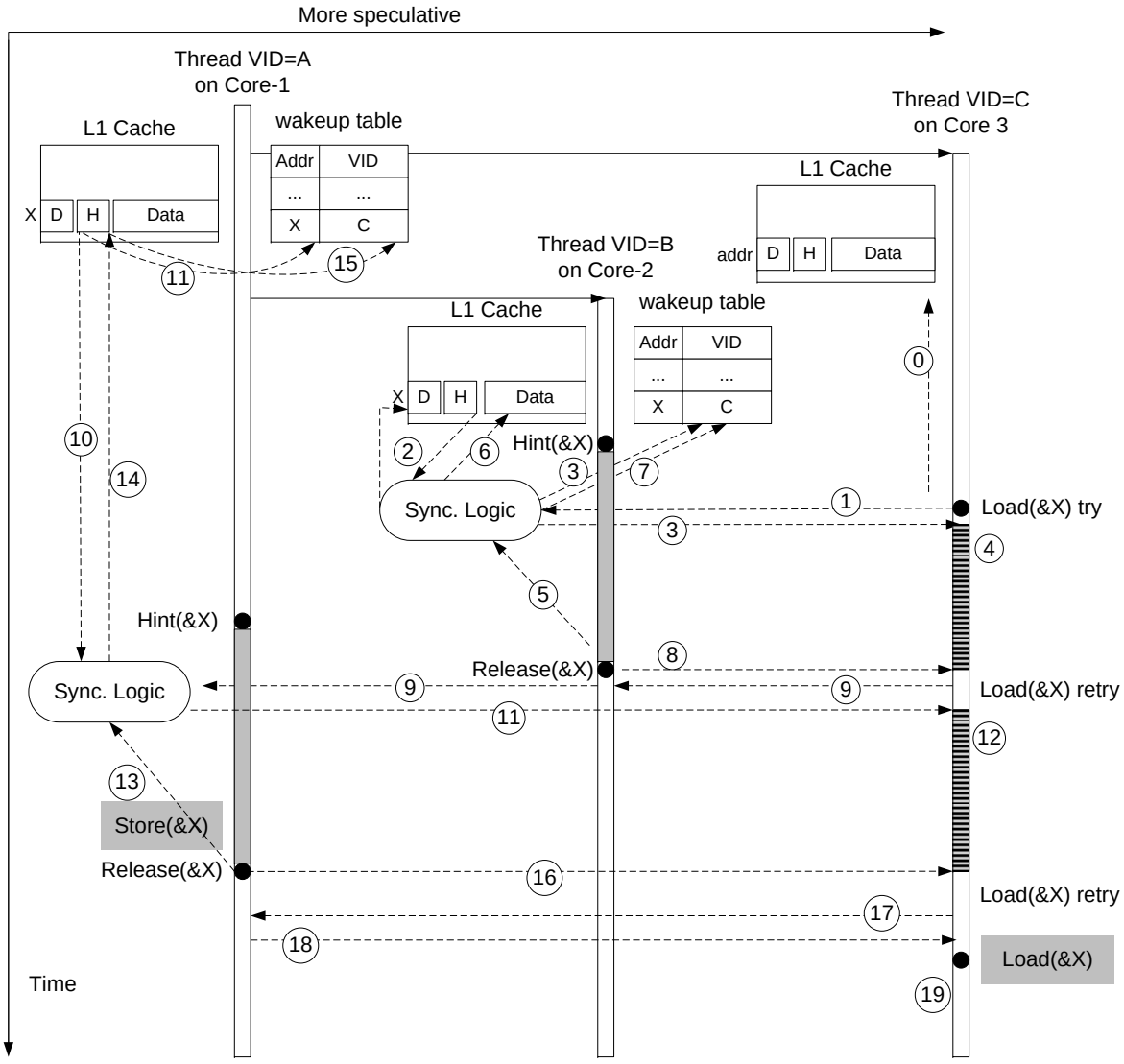


Figure 3.5: An Example of *HiRe* Run-Time Actions

C is suspended again (12).

When a *release*(&X) instruction executes in thread A later (13), it clears H-bit, and wakes up thread C (14,15,16). On resume execution, thread C re-executes load(&X) instruction and checks again. Since this time not only H-bit but also D-bit are set on core 1, which means core 1 has the latest update to location X in the system at this moment, the search is then stopped, and the cache value on core 1 is consumed by the load (18). Thread C can then move forward (19).

Note, in the above example, that the affecting scope of hint-release window on thread A is in fact enlarged via overlapping with the window of thread B. If not lingering due to the hint in thread B, the load in thread C would have missed the *hint* on thread A because that *hint* executes later than the load on thread C. In that case, a dependence violation would happen on execution of store(&X) on thread A.

Chapter 4

Methodology

4.1 Compiler Infrastructure and Code Generation

We implement our compiler in a copy of the POSH [25] compiler we obtained from the original authors, and we have ported it to GCC 4.3. We use the compiler to generate a few different binaries for each application evaluated, shown in Table 4.1. The *Base* case is a sequential binary that we normalize all of our plots against. *TLS* shows the result of the POSH compiler with all of its optimizations enabled. The major criteria for POSH to evaluate a task includes its size, static and dynamic hoist distance, violation rates, prefetching effect, and costs to parallelize it. Only high quality tasks will survive to the final output. During profiling, we use a $p_{sq} = 0.55$ and a $t_{szMin} = 30$.

Table 4.1: Compile Settings for Each Application Evaluated.

Name	Description
Base	-O2
TLS	Base + POSH optimizations
TLS+PRV	All POSH tasks plus *PRV* tasks that survive optimization.
SWsync	TLS + Signal&Wait Sync
HiRe	TLS + *HiRe* support

For PRV evaluation, *TLS+PRV* includes tasks automatically selected using all POSH's optimizations plus the support of *PRV* detection and parallelization. Even though a *PRV* may be detected and parallelized, POSH leverages a profiling stage to weed out ineffective tasks.

For HiRe evaluation, *HiRe* adds Hint-Release synchronization support to the POSH on the SLPs selected by the profiler and compiler. *SWsync* adds Signal&Wait synchronization to SLPs with occurring frequency greater than 50% in the profiling.

In PRV evaluation, PRV's effect on task selection is considered. Thus, the final tasks selected in *TLS+PRV* favor tasks containing more PRVs. To compare fairly, we force the *TLS* baseline to selecting the same tasks. In HiRe evaluation, however, feeding HiRe's effect back to task selection is too complicated (as will be discussed in Section 5.2.6), so we simply use the *TLS* task selector and use the same tasks as in the *HiRe* case. Thus, the final tasks selected in HiRe evaluation favor *TLS* case. For this reason, we see different base *TLS* performance in PRV evaluation and HiRe evaluation because they have different tasks.

4.2 Simulated Architecture

We target our compiler to the SESC simulator [36], a cycle accurate execution driven simulator. The simulator models out-of-order superscalar processors and memory systems with cycle-accurate precision. It also contains a TLS architecture model as shown in Table 4.2. Each processor is a 3-issue core and has a private L1 cache that buffers the speculative data. The L1 caches are connected through a crossbar to an on-chip shared L2 cache. All communication between cores occurs through the TLS coherence protocol which implements lazy task commit similar to [37]. Our baseline TLS architecture supports silent stores [22].

Table 4.2: Architectural Details. (Cycle counts are in processor cycles.)

Frequency	4 GHz	ROB	132
Fetch width	8	I-window	68
Issue width	3	LD/ST queue	48/42
Retire width	3	Mem/Int/Fp unit	1/2/1
Branch predictor:		Spawn Overhead	12 cycles
Mispred. Penalty	14 cycles	Squash Overhead	20 cycles
BTB	2K, 2-way		
Private L1 Cache:		Shared L2 Cache:	
Size, assoc, line	32KB, 4, 64B	Size, assoc, line	2MB, 8, 64B
Latency	3 cycles	Latency	10 cycles
		Memory:	
Lat. to remote L1	at least 8 cycles	Latency	500 cycles
		Bandwidth	10GB/s
***HiRe* parameters:**		***HiRe* parameters:**	
Wakeup Table (per core)	4 entries	Prefech Task Table	4 entries
Stalled load timeout	1600 cycles		
Hardware Dependence Predictors:			
MBVS: Moshovos *et al.* style predictor with unlimited size MDPT. [27](twolf,vpr)			
HybridMBVS: MBVS predictor that only synchronizes on memory locations likely to have a violation.(bzip2,gap,equake)			
Stall&Release: Cintra&Torrellas style predictor, with dynamically tuned release delay.(gzip,mcf,mesa)			

4.3 Benchmark

We evaluate our proposal on a set of SPEC CPU 2000 applications that work in the POSH compiler and the SESC simulator as provided by their original implementers. To accurately compare the performance of the different binaries, simply timing a fixed number of instructions is incorrect. Instead, simulation markers are inserted in the code of each binary, and simulations are run for a given number of markers. After skipping the initialization (typically 1-6 billion instructions), a certain number of markers are executed, so that the baseline binary graduates from 500 million to 1 billion instructions. We use *train-* input set for the profiling, and *ref-* input set for the simulation.

4.4 Implementation of Related Work to Compare Against

In the evaluation of PRV, we just compare our results to the basic TLS scheme. In the evaluation of HiRe, besides base TLS scheme, we also compare against both software and hardware schemes.

To compare HiRe with Zhai's proposal[53], we implement their scheme in our framework. *SWsync* binary in Table 4.1 is generated by adding Signal&Wait synchronization to SLPs that occur with a frequency greater than 50% during the profiling. Although they find 25% or 5% is the best threshold, our compiler selects both subroutine continuations and loops for TLS tasks. We explored the design space for *SWsync* with thresholds of 5%, 25%, 50%, 75%, and 90%, and found it gets the best performance on a threshold of 50%. Also, we assume special high speed signal&wait hardware to give it an additional advantage.

To compare HiRe to hardware predictors, we implement three hardware dependence predictors, shown at the bottom of Table 4.2. We do not have space to fully evaluate and characterize these designs. Therefore, when we compare against the hardware based predictors (labeled *HWpred*), we give each application the predictor that works the best for it. While this is unfair to us, we believe it may represent a highly tuned hybrid predictor that a clever chip designer may be able to develop. The applications that use a predictor are shown in parentheses above.

Chapter 5

Evaluation

5.1 Evaluation of *PRV*

5.1.1 Performance and Speculation Efficiency

Figure 5.1 shows the speedup on our evaluated applications over *Base*. The geometric mean of speedup for *TLS+PRV* is 1.31 for the CINT applications and 1.57 for the CFP. Compared to *TLS*, CINT is better on our proposed system, on average, by 5.84% and, CFP is better by 15.82%. These speedups were attained mostly using a fully-automatic compiler approach and are significant.

Figure 5.2 shows the fraction of wasted work for *TLS* and *TLS+PRV*. This is calculated as the number of squashed instructions (due to dependence violations) divided by the total number of committed instructions. Since *TLS+PRV* removes some cross-thread dependences by handling PRVs, it should have fewer dependence violations and, thus, have a lower waste rate than *TLS*. For most applications and for the average behavior for both CINT and CFP, we see such results in Fig. 5.2.

Table 5.1 provides some characterizations of the reduction variables that are included in the final speculative tasks selected by our compiler infrastructure. For each application, the reduction variables are classified as RV or PRV. Note that most reduction variables detected are *PRV* – except 2 significant *RVs* in *parser* (the 4 *RVs* in *mcf* don't speedup performance at all). That is why we did not show the speedup of *TLS+RV* in Fig. 5.1 – because it is almost the same as *TLS*, except that *parser* has the same speedup as that of *TLS+PRV*.

The last column of the Table 5.1, together with Fig. 5.1 and Fig. 5.2, explains how these reductions affect performance. Most of the gains of our approach come from *PRV* tasks selected in *twolf*, *vpr*, *art*, and *mesa*. Not surprisingly, these applications contain tasks that had no loop-carried dependences other than the *PRVs*. Once the *PRVs* were optimized, the tasks were parallelizable and provided significant performance gains. The loop from *twolf* described in

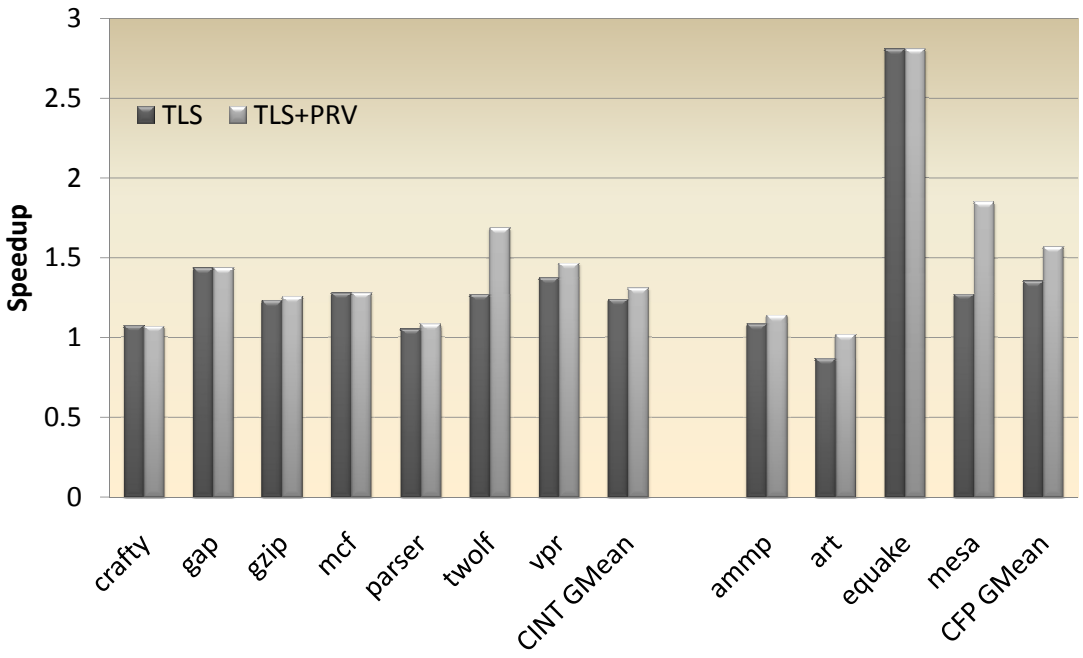


Figure 5.1: *PRV* Speedup Normalized to Base.

Section 2.1 is such a case. Impressively, identifying two *PRVs* was enough to provide the large performance gains in *twolf*. Note that the waste rates for *vpr* and *mesa* are considerably lower with *PRV* tasks. That's because the *PRVs* introduce major dependences in these tasks. Once these dependences are handled, the task selector prefers these tasks which are quite different from those of *TLS*. For *twolf*, more waste occurs on the *TLS+PRV* tasks than on the *TLS* tasks. Without PRV support, *TLS* selects a set of tasks which speculate less aggressively. For *TLS+PRV*, the additional speculation opportunities brought by PRVs brings the benefit of higher performance at the cost of higher waste.

In *parser*, *gzip*, and *mcf*, although *PRV/RV* are detected and handled, they don't boost performance much because either they are located in secondary or small tasks (*gzip*) or there still exist nontrivial cross-thread dependences other than *PRVs* (*mcf*). However, we can still notice the reduction in wasted work in Fig. 5.2 after handling the PRVs.

In *crafty*, *gap*, *ammp*, and *equake*, although many PRV/RV are detected (see Table 1), none of them survive profiling; thus, there is nearly no performance gain. However, the performance of *crafty* and *ammp* is slightly affected due to the selection of a different set of tasks. This is because the early phase of compilation handles all *PRVs* – this affects the behaviour of tasks in which *PRVs* are located and leads to a different profiling result.

The third column of Table 5.1 classifies the reductions from another aspect. *Local* means

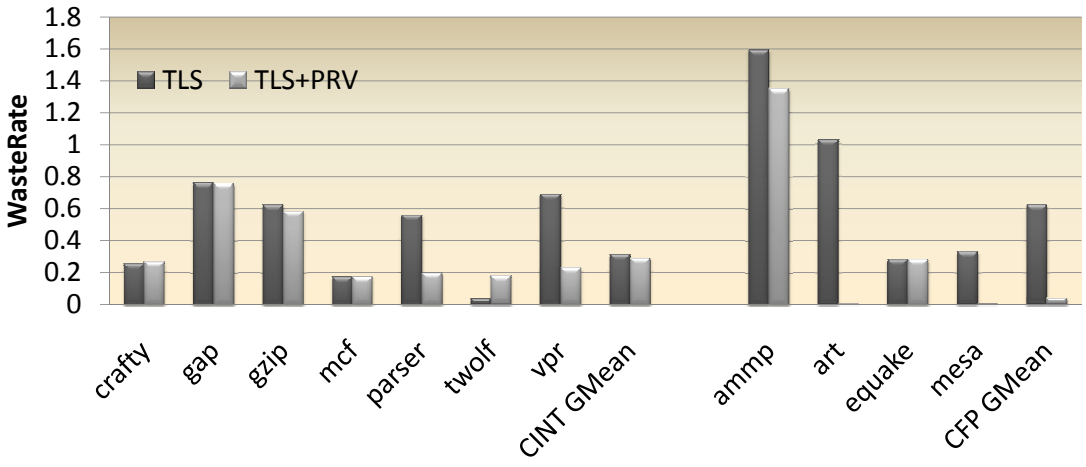


Figure 5.2: Fraction of Wasted Work of *PRV*. (The 2nd bars of *art* and *mesa* are too small to see.)

Table 5.1: *PRV* Characterizations.

Benchmarks	RV	PRV	Local	Ptr	Global	Analysis
crafty	0	0				No PRV survives, but it affects task-selection a little bit and thus slows down slightly.
gap	0	0				No PRV survives. TLS and TLS+PRV select same tasks.
gzip	0	1			1	It is located in 1 task, and speedup a little bit.
mcf	4	0	4			They are located in 2 tasks, but don't speedup the performance due to other dependences.
parser	2	0	2			They are located in 2 tasks, from which the performance gain mostly comes
twolf	0	3		2	1	They are located in 2 tasks, from which the performance gain mostly comes. TLS doesn't select these 2 tasks due to dependences on these PRVs.
vpr	0	1	1			It is located in 1 task, from which the performance gain mostly comes
ammp	0	0				No PRV survives, but it affects task-selection a little bit and thus speeds up slightly.
art	0	3	3			They are located in 3 tasks, from which the performance gain mostly comes
equake	0	0				No PRV survives. TLS and TLS+PRV select same tasks.
mesa	0	3	3			Located in 1 task, from which the performance gain mostly comes. Task selection affected

the RV/PRV is a locally declared variable whose address is not taken (thus has no alias); *Ptr* means it is a pointer variable; and *Global* means it is a global variable. From the table, note that (1) the latter 2 cases have may-aliases and need the full architectural and compiler support we proposed; (2) the *Local* PRVs are not traditional reductions and need our compiler schemes to handle them; (3) only *parser* contains significant RVe that can be handled by traditional techniques, but its performance gain is limited. Such a distribution of reductions shows the importance of handling *PRVs* (not just *RVs*) when parallelizing sequential codes.

5.1.2 Discussion

While many PRVs were identified by the compiler, there were many fewer tasks containing PRVs that were ultimately profitable. By examining the code that excluded these PRVs, we identified some key reasons. (1) Many PRV tasks in *gap* and *gzip* have small loop sizes and thus

do not overcome the initial cost of speculation. For this reason, the compiler eliminates them during profiling. We believe that loop unrolling, if applied judiciously, can help generate some good PRV tasks by increasing the task size. (2) Some PRVs appear only on a branch which is seldom taken. Supporting these PRVs may add synchronization when not needed. Finding ways to reduce this cost and generate better PRV code helps some cases found in *gzip*. (3) Some PRVs occur in tasks with other frequently occurring dependences, preventing the task from being selected. Incorporating additional techniques that target non-PRV dependences could increase the value of our mechanism.

Furthermore, some legitimate *PRVs* are missed by our compiler pass. Note, our algorithm requires that the update occur on a single variable. Consequently, we miss the classic array-reduction `A[i]+=...` if `i` is the induction variable for the innermost loop in the surrounding loop nest. Some preliminary analysis suggests that many such *PRVs* exist in this form even in SPEC applications and many of them are located in potentially good tasks. We believe that extending our techniques to support such *PRVs* is possible and would increase the performance of our techniques.

5.2 Evaluation of *HiRe*

We compare four sync schemes. The *SWsync* and *HWpred* bars are the sync schemes we compare against, as described in the previous section. The *HiRe* bar shows our *HiRe* scheme. The *HiRe+HWpred* bar shows a combination result of *HiRe* scheme and hardware dependence predictor. The *TLS* bar, if we don't normalize sync schemes to it, shows the basic *TLS* scheme without any synchronization support.

5.2.1 Speculation Efficiency and Performance

The most direct impact of *HiRe* is the reduction of dependence violations. We counted the number of violations for each application and normalized it to that of basic *TLS* scheme. Figure 5.3 shows the normalized violation rate and compares 4 sync schemes.

We find that every technique decreased the violation rates from *TLS*, and *HiRe* beats *HW-pred* and *SWsync* by a considerable margin. *mcf* and *vpr* drops dramatically under *HiRe* scheme; and *twolf* also has an outstanding decrease. Overall, *HiRe* has, by geometric mean, only 23% of the violations that TLS does. Whereas, *HWpred* has a rate of 53% and *SWsync* has a rate of 93%. Interestingly, Figure 5.3 shows that we can get the best violation rate if *HiRe* is combined with *HWpred* scheme, bringing it down to only 15%. This suggests that the two schemes handle some complementary dependence patterns, as expected.

To further understand the importance of reducing violation rate, we measured the Instruction Waste Rate. It is defined as the number of squashed instructions divided by the number

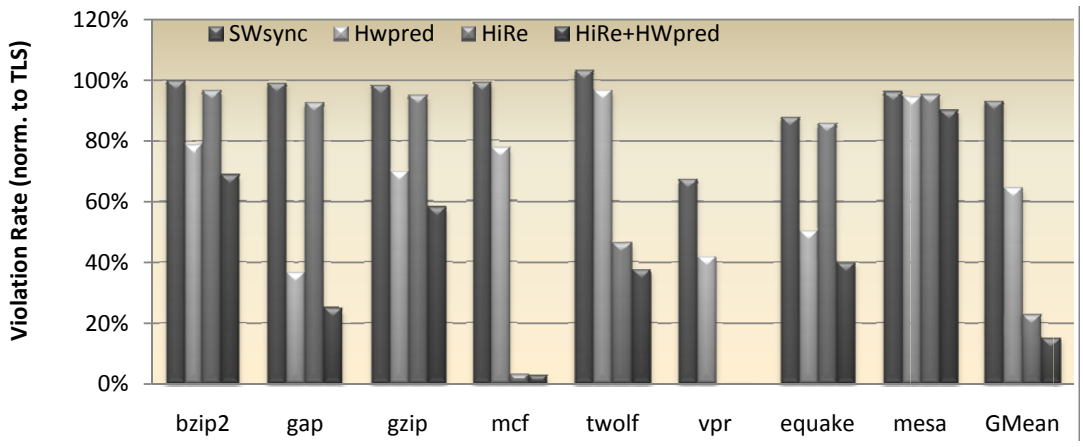


Figure 5.3: Violation Rate Comparison.

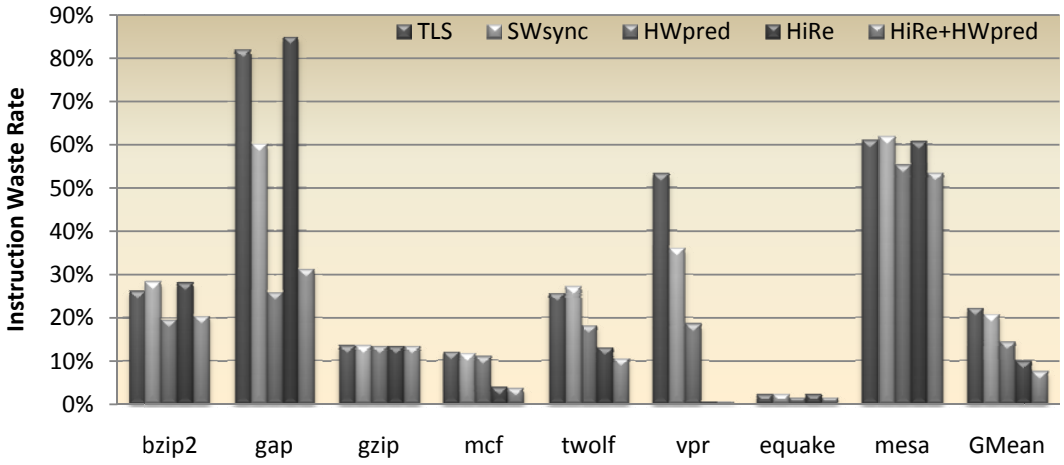


Figure 5.4: Wasted Instruction Rate

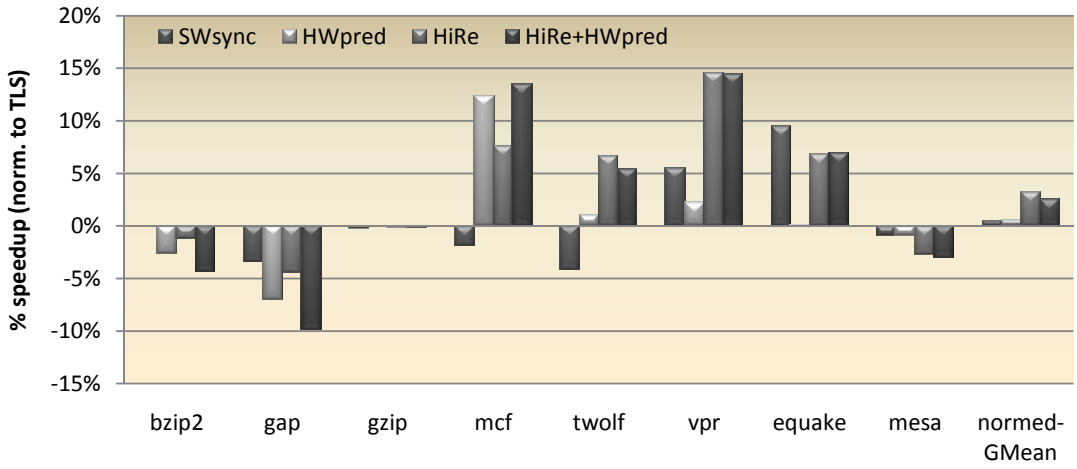


Figure 5.5: Performance Comparison Normalized to *TLS*

Table 5.2: *TLS* Speedup Compared to Serial Execution.

Benchmarks	bzip2	gap	gzip	mcf	twolf	vpr	equake	mesa	G-mean
TLS speedup	1.1187	1.181	1.0296	1.3225	1.5567	0.994	3.3677	1.333	1.3712

of useful instructions. This is an indicator of the efficiency of speculation. Figure 5.4 compares the waste rate across the same experiments. Overall, *HiRe* has an average waste rate of only 10%, while *TLS* has a waste rate of 22%, *SWsync* has a waste rate of 21%, and *HWpred* has a waste rate of 14%. Again, if we can get the lowest waste rate under the combined scheme of *HiRe* and *HWpred*.

We also measured the overall performance of these systems. Since the task selection policy is tuned for a TLS system without support for dependence prediction, we do not expect large speedups. Rather, we expect only modest speedups. *HiRe* is the best performing synchronization technique overall. Figure 5.5 shows the percentage of performance improvement for each system normalized to that of basic TLS. Note, bars above the line mean speedup and bars below mean slowdown from *TLS*. Table 5.2 gives speedup of base TLS from serial execution. Overall, *HiRe* achieves an average performance improvement of 3.2% over *TLS*. The largest single gains are in *vpr*, *mcf*, *equake*, and *twolf*. Also, the best gain of *HiRe* is on par with the best gains of *HWpred*. Interestingly, *HiRe* does not have the negative effects the other schemes have on application performance. This is important because a good dependence predictor should avoid slowdowns if it does not provide speedup. The worst degradation for an application on *HiRe* is -4.4% on *gap*.

To explain how our scheme decreases violations without having performance suffer much,

Table 5.3: Efficiency of Release Operation

Benchmarks	addedRelease	addedWakeup	RelWakeup
bzip2	23.89%	2.36%	64.49%
gap	0.00%	0.00%	0.00%
gzip	0.00%	0.00%	0.00%
mcf	87.30%	1.54%	37.94%
twolf	0.07%	2.36%	0.02%
vpr	47.27%	43.38%	53.83%
equake	0.00%	0.00%	0.00%
mesa	0.00%	0.00%	0.00%

Table 5.4: Task Statistics

Benchmarks	nAllTask	nSelTask	nPrefTask
bzip2	35	12	0
gap	24	13	3
gzip	37	17	0
mcf	6	5	3
twolf	44	22	0
vpr	14	5	0
equake	8	7	4
mesa	23	19	1

we collected statistics for release operations, shown in Table 5.3. The second column shows, for each application, how frequent a *Hint* is released from a side-exit *release* operation rather than the one following the store instruction. Since some of these releases could happen before the load occurs and thus the load need not stall at all, we show, in the third column, what percentage of such releases are going to really wake up a stalled load. And the last column shows the percentage of all the stalled loads that are released in this way. Because *HiRe* selects infrequent SLPs too, it is very important to provide a release on side-exits so that the load can proceed when the store doesn't happen.

5.2.2 *HiRe* Characterizations

In this section, we will characterize several aspects of the *HiRe* system. Table 5.4 shows the breakdown of selected TLS tasks. For each application, the second column shows the number of tasks chosen by the first phase of profiling; the third column is the number of tasks remaining after the second phase; the fourth column shows the number of tasks selected for prefetching. When these prefetching tasks execute, they employ the optimization described in Section 3.3.2.

Table 5.5 characterizes the dependences selected for *HiRe* and *SWsync*. For each application the second column *nAllDep* shows the total number of dependences detected by the profiler. *nSW* is the number of dependences selected and handled by *SWsync* scheme at the occuring frequency threshold of 50%. *nHIRE* is the number of dependencies pre-selected by *HiRe* scheme. Even though many dependences are pre-selected, we only handle a few of them in the end according to the SLP selection heuristics described in Table 3.1. *nStatHints* is the number of *Hint* instructions that are finally placed in the code – generally they had a sufficient hoist distance. *nDyncHints* is the dynamic number of *Hint* instructions executed and successfully committed at run time.

To evaluate the efficiency of *HiRe* scheduling algorithm, for the dependences synchronized using *HiRe*, we classify how often each dependence occurs in Region A, B, or C (defined in

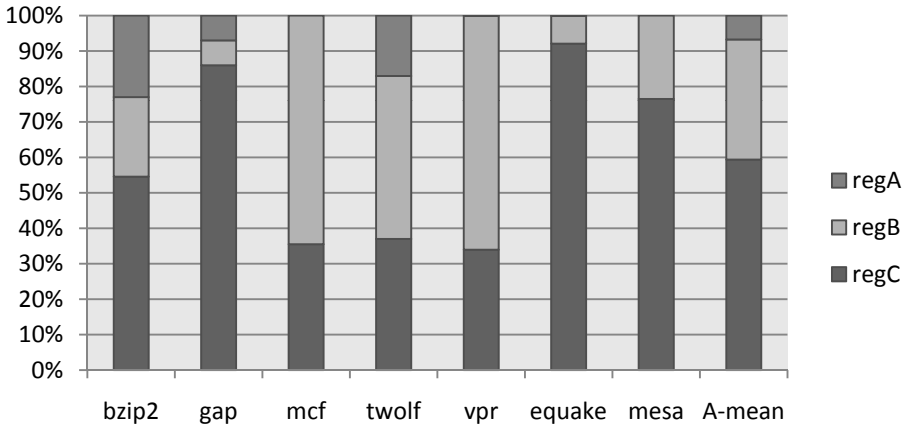


Figure 5.6: *HiRe* Regions Breakdown

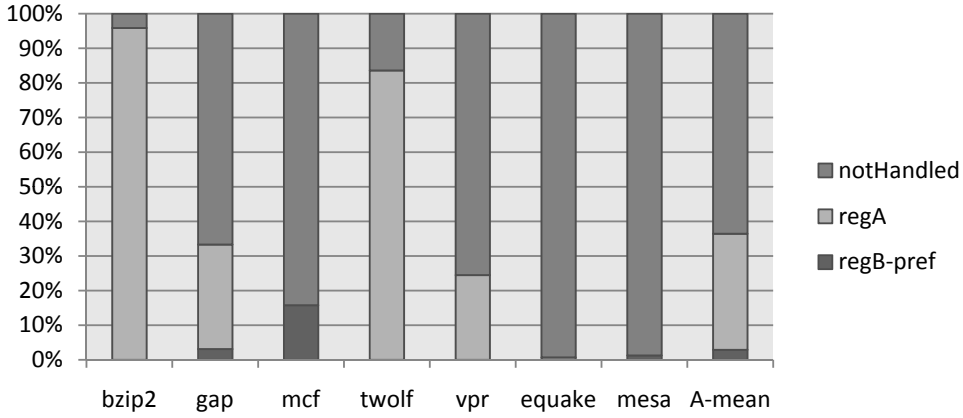


Figure 5.7: Violations Breakdown

Table 5.5: Dependence Statistics

APP	nAllDep	nSW	nHIRE	nStatHints	nDyncHints
bzip2	85	2	12	6	1641167
gap	236	6	134	8	2497839
gzip	102	4	12	0	0
mcf	27	3	5	4	16444042
twolf	124	6	56	6	4493541
vprman	43	2	24	8	11080543
equake	30	2	6	6	6616380
mesa	85	25	33	2	1530322

Section 3.1). Figure 5.6 shows the fraction in each region summed up over all dependences finally selected for *HiRe*. We removed *gzip* since there is no SLP selected for it. A large majority are captured in Region B and Region C. This means that most stores we marked using *HiRe* are successfully synchronized with their loads. Only *bzip2*, *gap*, and *twolf* had an appreciable fraction of dependences that occurred in Region A. For these applications, our compiler failed to take away some SLPs that cannot be handled well in the system because the compiler is using unified thresholds to all applications. Tuning the thresholds individually for these applications can fix this problem.

We also characterized the dependences that did violate into three categories, as shown in Figure 5.7. *NotHandled* are dependences for which we made no attempt; *regA* are dependences that occurred in region A and cannot be handled by *HiRe*; *regB-pref* are dependences that occurred in Region B and thus could be handled by *HiRe* but we chose not to synchronize them for their prefetching benefits. *bzip2*, *gap*, *twolf*, and *vpr* have a significant fraction of dependences in Region A. There are hints not hoisted far enough for these cases. *gap*, *mcf*, and *mesa* all have a significant number of prefetching based violations; this is expected since they all had prefetching tasks. *gap*, *gzip*, *mcf*, *twolf*, *vpr*, *equake*, and *mesa* had the majority of their dependence violations fall into the *NotHandled* category. Even so, the violation rate for *HiRe* is only 23% of that for *TLS*. Some of these remaining may need to be handled using a dynamic technique, or improvements are needed in the overall process of selecting SLPs and hoisting hints.

TLS adds some overhead to the execution of the program, as reported previously in [37] as f_{bloat}. Figure 5.8 shows the f_{bloat} of *TLS*, *SWsync*, and *HiRe* binaries respectively. The bloat

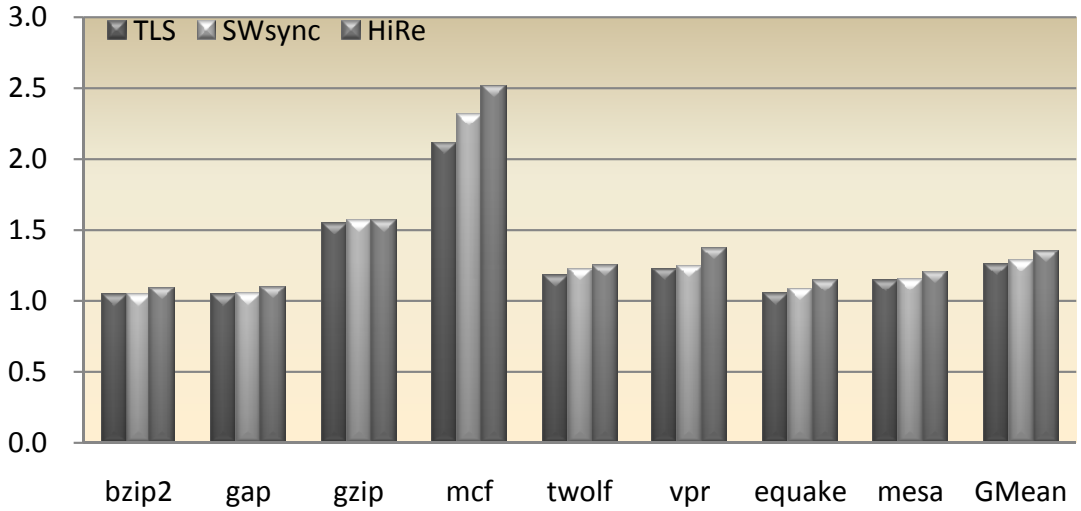


Figure 5.8: Fraction of Work Added due to *TLS* and *HiRe* Instructions.

factor for *TLS* in our experiments is 1.25, on average. This means that *TLS* performs 25% more instructions to complete the same amount of work. For *HiRe*, the bloat factor increases to 1.35, on average. This extra bloat is caused by *hint* and *release* instructions and the extra instructions added for the slice that precomputes the hint address. This increase in bloat is enough to reduce the performance gains of *HiRe* in our experiments.

5.2.3 *HiRe* Window Size and Synchronization Time

We show detailed timing statistics for SLPs in this section. Before looking at the figures, we need to define a few terminologies.

A sync window, or a HiRe window (*HireWin*), is defined as the region in between a hint instruction and its corresponding release instruction. It can be classified into 2 types: A Store-Released HiRe Window (*St-Win*) is a *HireWin* that really contains a store instruction updating the hinted memory location; A Release-Released HiRe Window (*Rel-Win*) is the case that the hinted memory location is not updated by any store instruction and is released by a release instruction; We call the time, in number of cycles, from a hint instruction to its release instruction the HiRe window size. We measure the HiRe window size with 3 average parameters as defined in line 1-3 in Figure 5.9(a). *avgHireWin* is the overall average window size of *Rel-Win* and *St-Win*.

A synchronization involves a store and a load. The HiRe window size parameters are used to evaluate the store side. At the load side, we measure the average waiting time of each sync load if it falls into region B (which is same region as the HiRe window). We define Sync Time

as the time from a load finding a hint and being suspended till it is resumed by a release instruction. Store-Released Sync Time (*StSyn*) is the Sync Time of the case that there is a store instruction updating the hinted memory location in its HiRe window. Release-Released Sync Time (*RelSyn*) is the Sync Time of the case that there is no such store instruction and a release instruction resumes the suspended load. *avgHireSyn* is the overall average Sync Time for both cases. Line 4-6 in Figure 5.9(a) shows their formulas.

Other subfigures in Figure 5.9 shows these 6 parameters for each application. For each HiRe window identified by an ID number on X-axis, the Y-axis shows its 6 parameters in simulated cycles.

The first thing we learn from Figure 5.9 is that the Sync Time varies greatly from window to window even in the same application. This shows how hard it will be to use a time-out method to resume suspended loads in order to keep them from over-synchronizing – it is impossible to find a fixed unified time-out time. And it is also hard to find an easy rule or a common pattern for this variation so as to train hardware to adjust the time-out time effectively. However, *HiRe* solves the over-sync problem easily by introducing the *release* instruction. This is the key reason why it doesn't hurt the performance while reducing the violations greatly via synchronization.

In *bzip2*, *mcf*, *vpr*, and *twolf*, release instructions play important roles. For some HiRe windows in these applications, for example *HireWin* 278 of *vpr* in Figure 5.9(f), the average values of window size (*avgHireWin*) and Sync Time (*avghireSyn*) are mostly dominated by the release bars (*RelWin* and *RelSyn*), but not by the store bars. In *HireWin-278* of *vpr*, the release instructions release the synchronized loads after 102 cycles of them being suspended since there is no corresponding store instruction that will be executed. Without such releases, the majority of the suspended loads would have to either wait for a time-out (1600 cycles in our experiments) or until the loading thread becomes a safe thread, which could be hundred of cycles. In these windows, release instructions are the key to avoid over synchronization. This result is consistent with Table 5.3, where we see the percentage of active release instructions is large.

On the other hand, however, some applications, such as *equake*, *mesa*, and *gap*, have almost no synchronization occuring via a release instruction because they do not suffer from the over synchronization problem. That also explains why *SWsync* outperforms *HiRe* in these cases.

There are also some windows, in which the *release* plays a small role. For example, in *HireWin* 456 of *vpr* in Figure 5.9(f), the store-based releas is the dominaing behavior.

Note that for some windows, the Sync Time is even greater than the window size. For example, *HireWin* 162 of *mcf* in Figure 5.9(d). This is because for some windows the hint and release instructions can execute much more frequently than the load. Although the window size is greater than the Sync Time when the sync happens, for most cases when there is no sync the window sizes are small. Thus, averagely we see a small window size but a big Sync Time.

Also, for some windows, such as *HireWin* 50 of *bzip2* in Figure 5.9(b), the *release* takes

RelWin = SUM(Rel-Win Size) / (# Rel-Win)

StWin = SUM(St-Win Size) / (# St-Win)

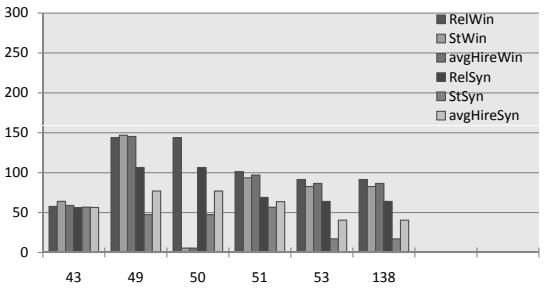
$$\text{avgHireWin} = \frac{\text{SUM(Rel-Win Size)} + \text{SUM(St-Win Size)}}{(\#\ \text{Rel-Win}) + (\#\ \text{St-Win})}$$

RelSyn = SUM(Rel-Syn time) / (# Rel-Syn)

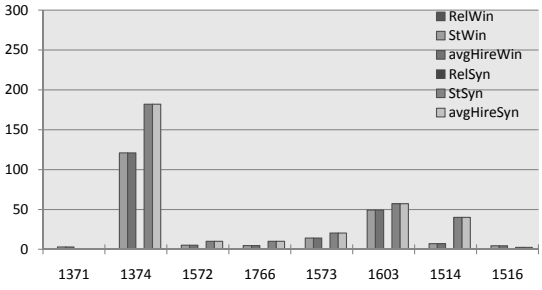
StSyn = SUM(St-Syn time) / (# St-Syn)

$$\text{avgHireSyn} = \frac{\text{SUM(Rel-Syn Time)} + \text{SUM(St-Syn Time)}}{(\#\ \text{Rel-Syn}) + (\#\ \text{St-Syn})}$$

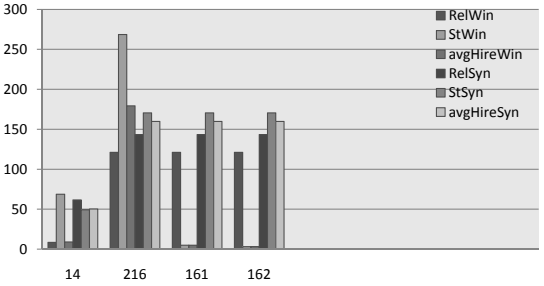
(a) Definitions



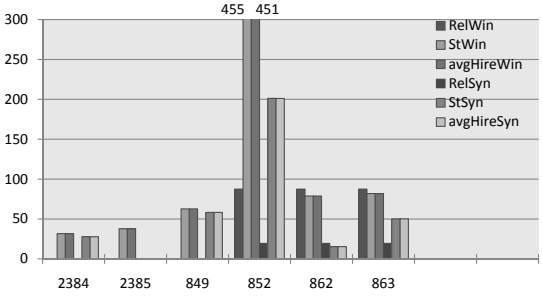
(b) bzip2



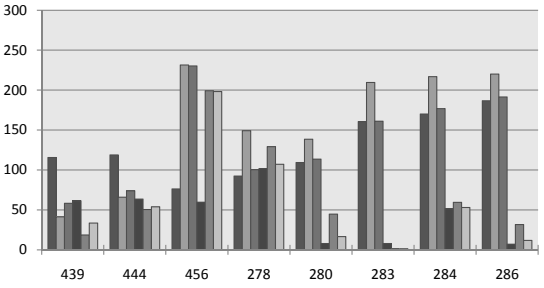
(c) gap



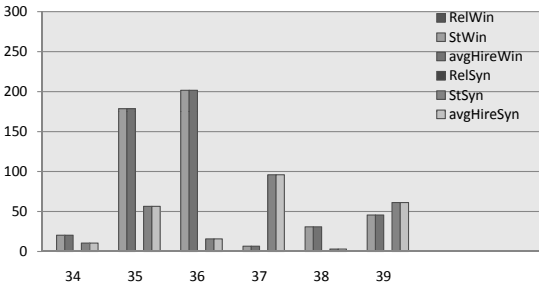
(d) mcf



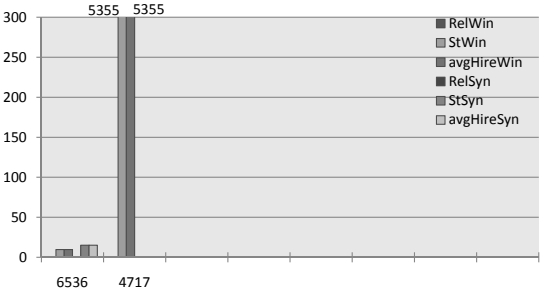
(e) twolf



(f) vpr



(g) equake



(h) mesa

Figure 5.9: Synchronization Timing of *HiRe* Windows.

much longer time to release the *hint* than the *store* does. This is because the *hint* and *store* are located in a loop task, and a side-exit *release* is schedule on the loop exit-edge; and, the *store* is not always updating the hinted location. Mostly the *store* keeps updating the location that is hinted and releases the *load* in a short time (thus a small StWin size). But, occasionally, if the *store* is not updating that location, the *load* will have to wait until the loop exits and the *release* is reached. In that case, we see a long release time released by *release*, and thus a big RelWin size. This also explains why the StWin dominates the average window size in this case.

5.2.4 Effect of Prefetching

As pointed out in Section 3.3.2, prefetching is important for performance of TLS system with a sync scheme. To evaluate this, we disabled the pretetching support in the simulator for the 4 applications with prefetching tasks (see Table 5.4). Compared to *HiRe* with prefetching support, the performance drops 3.1% for *mcf*, 1% for gap, 0.5% for *mesa*, and 0.3% for *equake* respectively. The effect of prefetching is not as big as we expected. After looking into the simulation details, we find that many L2 misses are located before the sync load rather than after it; thus their prefetching is not always prevented by the suspended load. Although our prefetching scheme is still helpful for the sync loads themselves and the loads after them, such benefit is reduced when there are more L2 misses located before the sync load in a thread. However, the prefetching scheme is generally beneficial to get higher performance. And, its role is important for cases which have high dependence coverage, since more loads will be stuck by more suspended sync loads.

5.2.5 SLP Selection

Even though many dependences occur in the code, we only handle a few of them in the end, mostly due to their cost. (Section 5.2.6). Table 5.6 characterizes the SLPs selected for *HiRe* step by step. For each application, the second column, *Detected*, shows the total number of dependence detected by the profiler.

The SLP selector can be broken into 3 steps according to Table 3.1. In order to get the statistics after each step, we run dedicated experiments and show them in Table 5.6 with 3 frames. The frame of *Can be handled* shows the statistics of SLP that survived reason 1 in Table 3.1. *Pre-selected* are the SLPs survived reason 1 through 5. *Post-selected* are the SLPs survived all 6 reasons and finally are placed in the code – generally they had a sufficient hoist distance for the hint. In each frame, The column of nW is the static number of writes in the code, nR is the number of reads, and $nSLP$ is the number of store-load pairs. Note that due to multi-load writes and multi-store reads, the number of SLP could be greater than the number of writes or reads. Finally, *dyncHints* is the dynamic number of Hint instructions executed and

Table 5.6: Dependence Statistics after Each Step of SLP Selection

	Detected	Can be handled				Pre-selected				Post-selected			
APP	nDep	nW	nR	nSLP	dyncHints	nW	nR	nSLP	dyncHints	nW	nR	nSLP	dyncHints
bzip2	85	17	7	22	2450441	11	6	12	2039135	6	3	7	1641167
gap	236	38	39	87	39947920	38	34	72	39947920	8	8	16	2497839
gzip	102	18	21	30	6123205	8	9	10	33544	0	0	0	0
mcf	27	6	8	10	21293295	4	4	5	16444042	4	4	5	16444042
twolf	124	32	35	58	22549068	28	34	51	9312486	6	8	15	4493541
vpr	43	20	24	36	20212353	14	19	22	17953881	8	9	11	11080543
equake	30	6	4	6	6616380	3	3	3	5711310	3	3	3	5711310
mesa	85	14	26	30	1578846	13	26	29	1530752	2	5	6	1530322

successfully committed at run time.

To evaluate the efficiency of our SLP selector, we show region- and violation- breakdown after each step of SLP selection in Figure 5.10 and Figure 5.11 respectively. *_all*, *_sel*, and *_prof* bars are accordant with *"Can be handled"*, *"Pre-selected"*, and *"Post-selected"* in Table 5.6. In Figure 5.11, we normalized the number of violations to *HiRe_all* and give its rates compared to *TLS* in Figure 5.12.

Overall we see good trends step after step: (1) percentage of region B is increasing. Compiler removes anticipated unsuccessful (region A) and useless (region C) HiRe code to reduce cost. This is important for performance because most of *HiRe* code is located in hot sections of the application. (2) number of violations are decreasing or don't increase while getting higher performance. (3) The percentage of violations caused by not handled dependences in Figure 5.11 is increasing. This is due to taking away unsuccessful *HiRe* code to control cost. The compiler finally choose not to handle *gzip* at all because handling it made the violation rate and performance worse.

5.2.6 Limitations

The current implementation verifies and evaluates the basic idea of *HiRe*, and thus, has many limitations In this section we discuss these limitations. We believe improvements can provide more benefit.

Dependence Coverage

We have to point out that the coverage of *HiRe*-handled SLPs is not as good as we hoped. This can be seen in Table 5.5 – *nSLP* of *"Can be handled"* versus *nDep* of *"Detected"*. Dynamically,

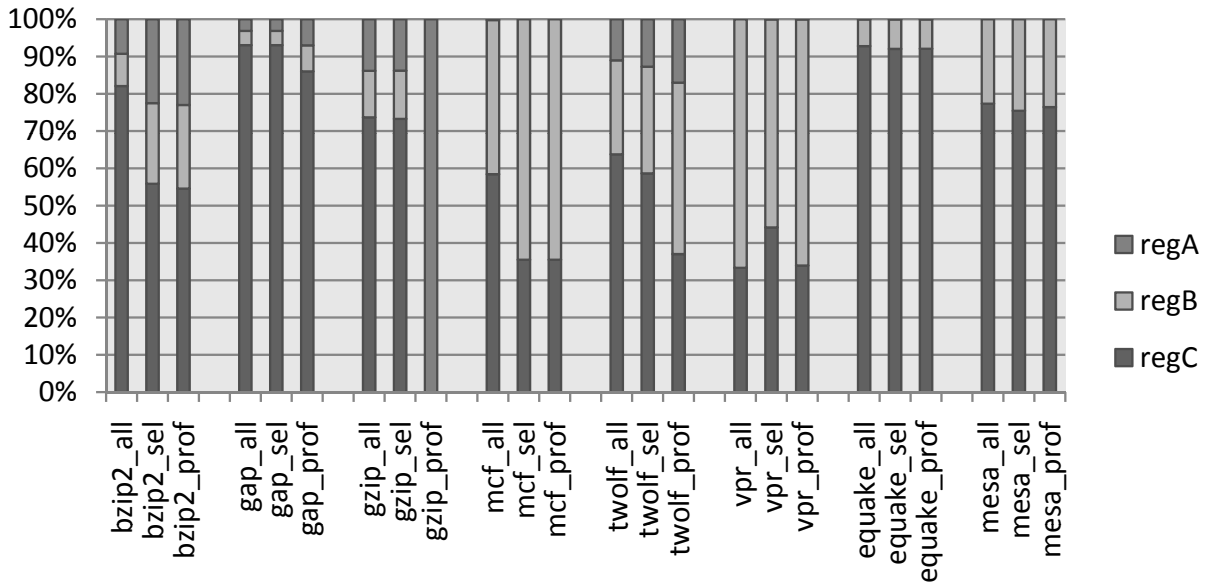


Figure 5.10: Region Breakdown after Each Step of SLP Selection

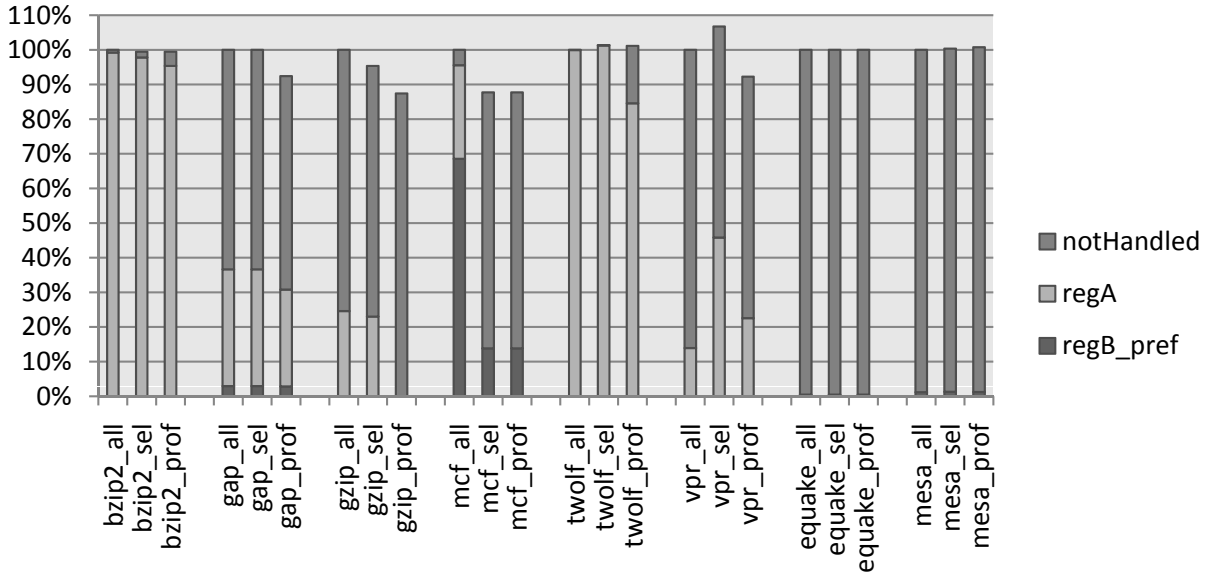


Figure 5.11: Violation Breakdown after Each Step of SLP Selection

APP	bzip2	gap	gzip	mcf	twolf	vpr	equake	mesa	GMean
HiRe_all	0.9726	1.0041	1.0886	0.0323	0.4596	0.0008	0.6907	0.9465	0.2313

Figure 5.12: *HiRe_all* Violation Rates Compared to *TLS*

in Figure 5.11, the not handled dependences dominate the violations of some applications from the very beginning, such as *equake_all* and *mesa_all* bars.

First of all, our implementation is in front-middle end of GCC. Back-end optimizations could push register variables back to the memory after our code transformation. These memory locations cannot be seen by our pass of optimization and thus dependences carried by them cannot be handled by *HiRe*. This is especially true for *equake* and *mesa*. This is the dominating reason of the low coverage now.

Another limitation due to our implementation is we hoist hints with addresses of scalars or pointer-referenced locations only. We have not handled hoisting hints with addresses of members of arrays or structures and thus gave up opportunities in there.

Function Boundary Limitation

Some applications in Figure 5.11 have a large section of regA, such as *bzip2* and *twolf*. This means some hints are not hoisted far enough to be seen by the load. We looked at the source code and found there are 2 major reasons for this. (1) The address of the store is generated too late in source code and there is no way to hoist the hint farther. Fortunately, such a case is not the dominating reason. (2) The communicating memory location is a global scalar, whose address is not changable, or is referenced by a global pointer whose value is very likely not changed throughout the hoisting region. For these memory locations, the hint can be hoisted as far as to the beginning of the function. However, this is still not far enough to be seen by some load due to nested function calls in the threads. Although our compiler applied function-inlining, it is still not good enough for some SLPs. This function boundary limitation is a difficult obstacle for *HiRe*, as well as other software sync schemes. Combining with hardware schemes could help this issue. For example *HiRe+HWpred* as we show in the evaluation section.

Cost

We already discussed *fBloat* in the evaluation and agreed that such cost hurts performance. Currently we use built-in functions to implement functionalities of *hint* and *release*. Through experiments we find that the major cost is not from inserting the precomputation slices, but from such implementation of *hint* and *release*. It hurts performance in two ways: first, a single operation needs many instructions to maintain the function frame and stack. Since most *HiRe* code is located in hot sections of the application, such cost is nontrivial. Second, the *hint* and *release* instructions now become branch instructions jumping to these built-in functions. They truncate the fetch width, and reduce ILP. Furthermore, these side-effects serve to postpone the store instruction and can introduce new dependence violations.

Task Selection

Currently, tasks were selected without considering the benefit of dependence prediction. The compiler prefers to choose tasks that contain fewer violating dependences. Such a task selection strategy limits our opportunity to get higher performance. Since the HiRe scheme can handle many dependences, it enables the selection of more tasks. Hence, we should take advantage of the dependence handling technique to help the task selection to achieve higher performance.

According to Figure 5.4, task selection is a hard job for *gap* because it still has a very high instruction waste rate even though the current task selection strategy tried to minimize it. We thus choose it to see if we can get higher performance via different task selection. We tried manually selecting different tasks assuming the support for *HiRe*, and managed to speed it up to 1.35x with *HiRe* support. Originally it was sped up to 1.18x by *TLS*.

Also, we tried to develop a compiler heuristic which can let the task selector take into account *HiRe*'s benefit when choosing tasks. We do so by ignoring some violating SLPs during profiling, with the assumption that the *HiRe* can handle them perfectly. However, we have not yet found a heuristic that maximizes the performance with a limited number of profiling phases. The difficulties include: (1) The profiler heuristic that estimates which SLPs could be handled by *HiRe* at run-time is not accurate enough and is misleading to some extent. This is because the *HiRe* scheme is very sensitive to timing. An SLP that caused a violation at the time of profiling may not violate at run time in simulation; at the same time, a SLP that is not anticipated to violate and thus is not handled by the compiler could violate at run time. (2) Task-selection and SLP-selection interact with each other and it's hard to get a convergent result quickly. On the one hand, the change of selected tasks will change timing greatly, and will make issue(1) much worse. On the other hand, an updated SLP selection could change the benefit of a task, and thus would change the result of task selection.

Theoretically, we can decouple the task selection and SLP selection by running a first phase of profiling to select tasks and then run a second phase of profiling to select SLPs, and repeat such steps until we get a converged result. However, such a method is time-consuming and unacceptable in reality. We tried a few compromises under this idea and got a sub-optimal result with 4 phases of profiling. With this method we successfully sped up *twolf* by 28%, *vpr* by 35%, and *mesa* by 10% compared to basic *TLS* scheme.

Task selection is out of the scope of this study. So, we didn't fully explore and evaluate this benefit. With this preliminary data, we just show the potential of taking advantage of *HiRe* during task-selection.

Chapter 6

Detailed Comparison to Related Work

6.1 *PRV* Related Work

Reductions are a type of recurrence that are easy to parallelize on multiprocessors. They have been well researched in the context of optimizing compilers. The work in this area focused on a variety of issues (loosely categorized as follows) including detection of reductions [1, 4, 30, 15], parallelization/scheduling strategies [12, 31, 43, 42, 35, 49, 50], speculative approaches [34, 8, 7, 54], and architectural support [13]. The earlier techniques were often limited to loops in which the reduction variables and operators were fully analyzable by the compiler (even if the loop bounds were unknown). However, given the frequency of array based reductions and the limitations of dependence analysis on indirect array references, many loops could be optimized with this technique.

More recent work has identified the importance of employing speculation to extract more parallelism from hard-to-analyze reductions. Rauchwerger and Padua proposed the LRPD test [34] as a way of overcoming the limitations of static analysis. Instead of requiring a complete static analysis, some disambiguation tests were delayed until runtime. Their scheme works by first creating private versions of each array participating in the reduction computation, and calculating read and write sets for each array at runtime. Their technique is able to detect if the operations were in fact reductions during the actual execution, even though static analysis could not verify it either due to control or aliasing. Refinements of the LRPD algorithm aim to increase coverage and reduce overhead of this approach [8]. Many of the arrays classified by the LRPD scheme would be classified as *PRVs* by our scheme. However, these mechanisms require full control-flow analysis of the code being parallelized and identification of all loads and stores that *may* participate in the reduction. Non-analyzable control flow (e.g. into libraries)

with unknown reads/updates cannot be treated with this mechanism. For nested loops which dominate execution time, this limitation is reasonable, however, for many C programs, these restrictions are prohibitive. Instead, we need an approach that can cope with potentially unknown reads or writes to the reduction variable at runtime. Our proposed scheme can tolerate such reads or writes to the reduction variable by monitoring all accesses to the variable and taking corrective actions when such an update occurs. The proposed mechanism does not require the insertion of dependence tracking and tests to validate the reduction and can handle unknown paths and references never analyzed by the compiler.

Hardware support for reductions has been proposed in the past that extends beyond support for efficient synchronization primitives. Garzaran *et al.* proposed PCLR [13], hardware support for reductions that accelerates the merging phase of the reduction after the parallel region is complete. Their approach allows reductions to complete efficiently and lazily by combining the partial results of a reduction operation in hardware at the directory controller rather than in software. Our hardware support also performs merging, but that is not its primary responsibility. Our hardware support is focused on monitoring accesses to the reduction variable and taking corrective actions when needed. Our approach is largely orthogonal to PCLR and could be integrated with aspects of PCLR to support efficient merging on highly parallel codes. Past work on support for efficient synchronization primitives is also orthogonal to our scheme and could be leveraged to reduce sync time.

Our definition of PRV shares many similarities with that described by Zhang *et al* in UPAR[55]. However, UPAR only evaluates certain kinds of PRV. Our approach broadens the definition; it tolerates explicit accesses to the PRV, not just aliases. In addition, we search for PRVs in pointer-intensive integer codes, and our approach is better suited to find a wide variety of PRVs in this domain.

Compilers and systems for Thread-Level Speculation have identified the need to effectively handle reduction variables. Zhai *et al.* [54] show the benefit of reductions for SPECint applications and shows modest gains. However, this mechanism employed the strict definition of reduction variable, not the more flexible definition of PRV proposed in this paper. Also, Prabhu *et al.* [32] identifies reductions as an important transformation to unlock the potential of key loops in *vpr*, *mcf*, and *twolf*. Since they focused on manual techniques they transformed some reductions manually that would be classified as *PRV* in our approach. Finally, work in Thread-Level Speculation targeting efficient synchronization of cross-thread dependences [52, 53] is also relevant, since they are an alternative, but less direct, mechanism for supporting reduction variables.

6.2 *HiRe* Related Work

There are two closely related prior works that study memory dependence synchronization for TLS [27, 53]. In this thesis, we call Zhai's software-based scheme *Binding Sync* or *SWsync*, and Moshovos's hardware predictor *HW Predictor* or *DDP*.

In Table 6.1, the first column shows the schemes that are compared against *HiRe*. The second column indicates the method of dependence pair selection in order to normalize the comparison. *HiRe* and Zhai's proposal [53] cannot identify new dependences at runtime, whereas DDP [27] can identify new load-store pairs. Based on previous analysis, we anticipate that this leads to few discrepancies in practice[1]. Therefore, for the purpose of this discussion, we assume a load-store pair that meets the criteria for synchronization.

The next two columns consider the case of load occurring in advance of a write from its store set. The third column examines the case in which no dependence will actually occur. The first two approaches will stall since the load occurs frequently in a violation. The Binding Sync likely stalls more since it requires the insertion of signal&wait in the code. However, *HiRe* will not stall since no dependence exists, however, it does pay some overhead in terms of the *hint* instruction and its slice. This is an important contribution because even load-store pairs that frequently cause violations are likely to execute without causing one. The fourth column considers the case in which the store, if it occurs, will write to the same address. This is the case all synchronization mechanisms are designed to handle, and all will force a sync in this case. However, if the store does not occur, the software scheme must still conservatively sync. Furthermore, the DDP does not know when to release the load and must either timeout or wait to become safe. *HiRe* improves on these cases by rightly stalling the load but releasing it as soon as the store executes or as soon as it reaches the release.

Finally, the last column considers the case that the load occurs after the store. Ideally, this case should add no overhead to execution. However, Binding Sync must still incur some overhead to receive the signal check if a dependence as occurred and, in some cases, retrieve a communicated value. This extra work is inserted to optimize the case that a dependence occurs, but it penalizes execution in the case that the load would have been ordered properly. DDP adds no time overhead in this case as long the store has updated the DDP structures before the load executes. *HiRe* adds no additional cost in this case as well.

Overall, *HiRe* offers a low cost approach for synchronizing data dependences in TLS. It can be cheaper than prior schemes by restricting synchronization only to the cases that require a stall and by releasing loads that were prematurely stalled in safe and timely manner.

[1] If profiling does not uncover problematic dependences, then the training input can be expanded to include these behaviors over time, and we believe does not represent a fundamental limitation except in isolated cases.

Table 6.1: *HiRe* Compared to Other Synchronization Mechanisms.

	Dependence Detection	**No Dependence**	**Dependence Can Exist**	**Dependence with LD After ST**
Binding Sync. [Zhai et al.]	Profiling w/ Static Selection	Sync	Sync	Dequeue (+ check ovhd)
HW Predictor [Moshovols et al.]	Dynamic Training	Sync w/ Timeout	Sync w/ Timeout	None (likely case)
HiRe	Profiling w/ Static Selection	None	Sync on Hint w/ Release	None

Chapter 7

Conclusions and Future Directions

This thesis proposes novel techniques for efficient cross-thread data dependence handling for TLS systems. First, we consider a kind of shared variable that may behave like a reduction at runtime, called partial reduction variables (*PRV*). Given the frequency of *PRV*, it is important to consider hardware and software mechanisms to exploit them. We propose a compiler-architectural integrated mechanism to automatically parallelize *PRVs* on TLS systems. Second, we propose the *HiRe* technique that is able to synchronize both frequently and infrequently occurring dependences in irregular TLS tasking patterns. To solve the over-sync problem, our mechanism provides a way to wakeup the load and let it resume execution as soon as the store *will or will not* occur.

We implement our proposals on one of the state-of-the-art TLS systems, and evaluate them on a set of SPEC 2000 applications. We find that supporting *PRVs* provides up to 46% and on average 10.7% performance gain over basic TLS. Also, a TLS system supporting *HiRe* suffers from only 22% of the violations that occur in our base TLS system, and it cuts the waste rate of TLS in half. Furthermore, it outperforms the prior approaches by 3%.

For *PRVs* in highly irregular and integer codes, there are still many important optimization opportunities remaining. Depending on the reference or control structure that leads to detection of a *PRV* instead of *RV*, *PRVs* can be further classified into a variety of types. Parallelization strategies could be tailored for each kind of *PRV* to enable the most parallelism. Furthermore, *PRVs* can also be important in a variety of other systems. For example, dynamic optimization environments are often limited in the kinds of parallelization transformations that are possible due to the limitations of dynamic analysis – both in terms of the scope of code analyzed and the quality of analysis. The flexibility of our proposed hardware mechanism may be valuable when speculatively parallelizing loops in such environments.

For *HiRe*, we believe there are many ways this work can be extended to cover more dependences. Due to the limitations of the compiler, many hints cannot be hoisted far enough in

advance of the store to be useful. Also, there are times that Release could be scheduled earlier but was prevented due to function call boundaries. Carefully integrating the *HiRe* technique with a hardware dependence predictor may significantly boost coverage. Another interesting possibility is a hardware predictor that generates hints and releases rather than trying to synchronize loads and stores directly. We believe that such refactorings could allow even higher coverage than achieved in this work. If data dependence prediction can achieve the same or better accuracies as branch prediction, automatic parallelization techniques like TLS may become ever more attractive.

Task selection, although out of the scope of this research, can be greatly enhanced by techniques presented in this study. In our current infrastructure, as well as most other current TLS systems, only a few TLS tasks are beneficial and can be selected because tasks carrying many data dependences are very likely to be weeded out due to their high violation rates. Performance of these TLS systems are quite limited because of the limited number of tasks. Since many dependences can be handled with the techniques presented in this study, squash rates of tasks carrying these dependences can drop greatly; and as a result more good TLS tasks can survive the weeding out passes in the compiler and will very likely bring higher performance. An interesting TLS framework is: (1) Using value prediction technique to handle dependences with easy-to-predict values, such as silent stores, induction variables, and function returns. (2) Using *PRV* schemes to handle PRVs. (3) Using *HiRe* schemes to predict and selectively synchronize all other dependences. (4) Finally, let task selection take advantage of these dependence handling techniques and weed out frequent violating tasks that carry many hard-to-handle dependences. These techniques altogether open a door to high-performance speculation-efficient TLS systems.

REFERENCES

[1] Zahira Ammarguellat and W. L. Harrison III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. *SIGPLAN Not.*, 25(6):283–295, 1990.

[2] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *Proceedings of 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, August 2002.

[3] B. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 11(7):422–426, July 1970.

[4] David Callahan. Recognizing and Parallelizing Bounded Recurrences. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 169–185. Springer-Verlag, 1992.

[5] P. S. Chen, M. Y. Hung, Y. S. Hwang, R. D. Ju, and J. K. Lee. Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis. In *Proceedings of the 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 25–36, June 2003.

[6] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*, pages 43–54, 2002.

[7] F. Dang, M. Jesus Garzaran, M. Prvulovic, Ye Zhang, A. Jula, Hao Yu, N. Amato, L. Rauchwerger, and J. Torrellas. Smartapps, an Application Centric Approach to High Performance Computing: Compiler-Assisted Software and Hardware Support for Reduction Operations. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 172–181, 2002.

[8] Francis Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. Technical report, Texas A&M University, 2001.

[9] Diego Novillo. Tree SSA — A New Optimization Infrastructure for GCC. In *Proceedings of the First GCC Developers Summit*, May 2003.

[10] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs. In *Proceedings of SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, June 2004.

[11] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 1987.

[12] Allan L. Fisher and Anwar M. Ghuloum. Parallelizing Complex Scans and Reductions. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 135–146, Orlando, Florida, United States, 1994. ACM.

[13] Maria Jesus Garzaran, Milos Prvulovic, Ye Zhangy, Josep Torrellas, Alin Jula, Hao Yu, and Lawrence Rauchwerger. Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, page 243. IEEE Computer Society, 2001.

[14] GNU Compiler Collection. URL, 2009. http://gcc.gnu.org/.

[15] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-Driven SSA Form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, 1995.

[16] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 8th* , October 1998.

[17] Lance Hammond, Mark Willey, and Kunle Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, San Jose, California, United States, 1998. ACM.

[18] Liang Han, Wei Liu, and James M. Tuck. Speculative parallelization of partial reduction variables. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 141–150, New York, NY, USA, 2010. ACM.

[19] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[20] T.A. Johnson, R. Eigenmann, and T.N. Vijaykumar. Min-cut Program Decomposition for Thread-Level Speculation. In *Proceedings of SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[21] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers*, pages 866–880, September 1999.

[22] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 182–191, New York, NY, USA, 2000. ACM.

[23] Kevin M. Lepak and Mikko H. Lipasti. Temporally silent stores. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 30–41, San Jose, California, 2002. ACM.

[24] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *Proceedings of the 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, March 2006.

[25] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS Compiler that Exploits Program Structure. In *Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, New York, New York, USA, 2006. ACM.

[26] Pedro Marcuello, Jordi Tubella, and Antonio Gonzlez. Value prediction for speculative multithreaded architectures. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 230–236, Haifa, Israel, 1999. IEEE Computer Society.

[27] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th annual international symposium on Computer architecture - ISCA '97*, pages 181–193, Denver, Colorado, United States, 1997.

[28] Andreas Moshovos and Gurindar S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *MICRO'97*, pages 235–245, 1997.

[29] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative Thread-Level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, page 303. IEEE Computer Society, 1999.

[30] Shlomit S. Pinter and Ron Y. Pinter. Program Optimization and Parallelization Using Idioms. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 79–92, Orlando, Florida, United States, 1991. ACM.

[31] Bill Pottenger and Rudolf Eigenmann. Parallelization in the Presence of Generalized Induction and Reduction Variables. *In ACM Int. Conf. on Supercomputing (ICS95)*, 1995.

[32] Manohar K. Prabhu and Kunle Olukotun. Exposing Speculative Thread Parallelism in SPEC2000. In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 142–152, Chicago, IL, USA, 2005. ACM.

[33] Carlos Garca Quiones, Carlos Madriles, Jess Snchez, Pedro Marcuello, Antonio Gonzlez, and Dean M. Tullsen. Mitosis Compiler: an Infrastructure for Speculative Threading Based on Pre-Computation Slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 269–279, Chicago, IL, USA, 2005. ACM.

[34] L. Rauchwerger and D.A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *Parallel and Distributed Systems, IEEE Transactions on*, 10(2):160–180, 1999.

[35] Xavier Redon and Paul Feautrier. Scheduling Reductions. In *Proceedings of the 8th International Conference on Supercomputing*, pages 117–125, Manchester, England, 1994. ACM.

[36] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[37] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 179–188, Cambridge, Massachusetts, 2005. ACM.

[38] Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, and Josep Torrellas. Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation. In *Proceedings of the 19th International Conference on Supercomputing*, June 2005.

[39] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[40] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[41] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede Approach to Thread-Level Speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, 2005.

[42] Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. Detection and Global Optimization of Reduction Operations for Distributed Parallel Machines. In *Proceedings of the 10th International Conference on Supercomputing*, pages 18–25, Philadelphia, Pennsylvania, United States, 1996. ACM.

[43] Yong Meng Teo, Wei-Ngan Chin, and Soon Huat Tan. Deriving Efficient Parallel Programs for Complex Recurrences. In *Proceedings of the second International Symposium on Parallel Symbolic Computation*, pages 101–110, Maui, Hawaii, United States, 1997. ACM.

[44] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. Hot Chips, August 1999.

[45] J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew. The Superthreaded Processor Architecture. *IEEE Trans. on Computers*, 48(9):881–902, September 1999.

[46] J. Y. Tsai, Z. Jiang, and P. C. Yew. Compiler Techniques for the Superthreaded Architecture. In *International Journal of Parallel Programming*, pages 27(1):1–19, 1999.

[47] T. Vijaykumar and G. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 81–92, November 1998.

[48] Christoph von Praun, Luis Ceze, and Calin Cacaval. Implicit Parallelism with Ordered Transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–89, San Jose, California, USA, 2007. ACM.

[49] Jan-Jan Wu. An Interleaving Transformation for Parallelizing Reductions for Distributed-Memory Parallel Machines. *J. Supercomput.*, 15(3):321–339, 2000.

[50] Hao Yu and Lawrence Rauchwerger. Adaptive Reduction Parallelization Techniques. In *Proceedings of the 14th International Conference on Supercomputing*, pages 66–77, Santa Fe, New Mexico, United States, 2000. ACM.

[51] A. Zhai, C. Colohan, J. Steffan, and T. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *ASPLOS X Proceedings*, San Jose, CA, October 2002.

[52] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler Optimization of Scalar Value Communication between Speculative Threads. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–183, San Jose, California, 2002. ACM.

[53] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of Memory-Resident value communication between speculative threads. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 39, Palo Alto, California, 2004. IEEE Computer Society.

[54] Antonia Zhai, Shengyue Wang, Pen-Chung Yew, and Guojin He. Compiler Optimizations for Parallelizing General-Purpose Applications under Thread-Level Speculation. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 271–272, Salt Lake City, UT, USA, 2008. ACM.

[55] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. A Unified Approach to Speculative Parallelization of Loops in DSM Multiprocessors. Technical report, 1998.