A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code

DONGKEUN KIM and DONALD YEUNG
Electrical and Computer Engineering Department
Institute for Advanced Computer Studies
University of Maryland at College Park

Pre-execution is a promising latency tolerance technique that uses one or more helper threads running in spare hardware contexts ahead of the main computation to trigger long-latency memory operations early, hence absorbing their latency on behalf of the main computation. This article investigates several source-to-source C compilers for extracting pre-execution thread code automatically, thus relieving the programmer or hardware from this onerous task. We present an aggressive profile-driven compiler that employs three powerful algorithms for code extraction. First, *program slicing* removes non-critical code for computing cache-missing memory references. Second, *prefetch conversion* replaces blocking memory references with non-blocking prefetch instructions to minimize pre-execution thread stalls. Finally, *speculative loop parallelization* generates thread-level parallelism to tolerate the latency of blocking loads. In addition, we present four "reduced" compilers that employ less aggressive algorithms to simplify compiler implementation. Our reduced compilers rely on back-end code optimizations rather than program slicing to remove non-critical code, and use compile-time heuristics rather than profiling to approximate runtime information (*e.g.*, cache-miss and loop-trip counts).

We prototype our algorithms on the Stanford University Intermediate Format (SUIF) framework and a publicly available program slicer, called *Unravel* [Lyle and Wallace 1997]. Using our prototype, we undertake a performance evaluation of our compilers on a detailed architectural simulator of an 8-way out-of-order SMT processor with 4 hardware contexts, and 13 applications selected from the SPEC and Olden benchmark suites. Our most aggressive compiler improves the performance of 10 out of 13 applications, reducing execution time by 20.9%. Across all 13 applications, our aggressive compiler achieves a harmonic average speedup of 17.6%. For our reduced compilers, eliminating program slicing and relying on back-end optimizations degrades performance minimally, suggesting that effective pre-execution compilers can be built without program slicing. Furthermore, without cache-miss profiles, we still achieve good speedup, 15.5%, but without loop-trip count profiles, we achieve a speedup of only 7.7%. Finally, our results show compiler-based pre-execution can benefit multiprogrammed workloads. Simultaneously executing applications achieve higher throughput with pre-execution compared to no pre-execution. Due to contention for hardware contexts, however, time-slicing outperforms simultaneous execution in some cases where individual applications make heavy use of pre-execution threads.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**]: Performance Anal-

This research was supported in part by NSF Computer Systems Architecture grant CCR-0093110, and in part by NSF CAREER Award CCR-0000988.

Author's address: Dongkeun Kim and Donald Yeung, University of Maryland, Department of Electrical and Computer Engineering, College Park, MD 20742. E-mail: dongkeun@eng.umd.edu, yeung@eng.umd.edu

ysis and Design Aids; C.0 [**General**]: Modeling of computer architecture; System Architectures; C.4 [**Performance of Systems**]: Design Studies; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Algorithms, Design, Experimentation, Performance, Measurement

Additional Key Words and Phrases: Data Prefetching, Multithreading, Program Slicing, Speculative Loop Parallelization, Prefetch Conversion, Memory-Level Parallelism, Pre-Execution

1. INTRODUCTION

Processor performance continues to be limited by long-latency memory operations. In the past, researchers have studied prefetching [Chen and Baer 1995; Mowry 1998] to tolerate memory latency, but these techniques are ineffective for irregular memory access patterns common in non-scientific applications. Recently, a more general latency tolerance technique has been proposed, called *pre-execution* [Annavaram et al. 2001; Collins et al. 2001; Collins et al. 2001; Kim and Yeung 2002; Liao et al. 2002; Luk 2001; Moshovos et al. 2001; Roth and Sohi 2001; 2002; Sundaramoorthy et al. 2000; Zilles and Sohi 2001]. Pre-execution uses idle execution resources, for example spare hardware contexts in a simultaneous multithreading (SMT) processor [Tullsen et al. 1996], to run one or more helper threads in front of the main computation. Such *pre-execution threads* are purely speculative, and their instructions are never committed into the main computation. Instead, the pre-execution threads run code designed to trigger cache misses. As long as the pre-execution threads execute far enough in front of the main thread, they effectively hide the latency of the cache misses so that the main thread experiences significantly fewer memory stalls.

A critical component of pre-execution is the construction of the pre-execution thread code. Since this task is labor-intensive and prone to human error, it is highly inconvenient for programmers to carry out. Hence, for pre-execution to become a widely accepted latency tolerance technique, the construction of pre-execution code must be automated.

The design space for automating pre-execution is quite large because pre-execution code can be extracted at any point in time, *e.g.* compile, link, load, or runtime. In Figure 1, we show four possible approaches. Figure 1a illustrates *compiler-based extraction.* In this approach, a source-to-source compiler extracts the pre-execution code at compile time via static analysis of program source code. The compiler emits source-level pre-execution code. Alternatively, Figure 1b illustrates *linker-based extraction.* Rather than analyze source code, this approach extracts pre-execution code from program binaries at link or load time using a binary analysis tool, producing binary-level pre-execution code. Similar to linker-based extraction, *dynamic optimizer-based extraction*[1] in Figure 1c also analyzes and extracts binary-level code, but does so at runtime using dynamic optimization techniques. Finally, Figure 1c illustrates *hardware-based extraction.* In this approach, trace-processing hardware inside the processor extracts the pre-execution code from in-

[1]To our knowledge, dynamic optimizer-based extraction has not yet been studied, but we believe it is a viable approach. For completeness, we include it in our design space.

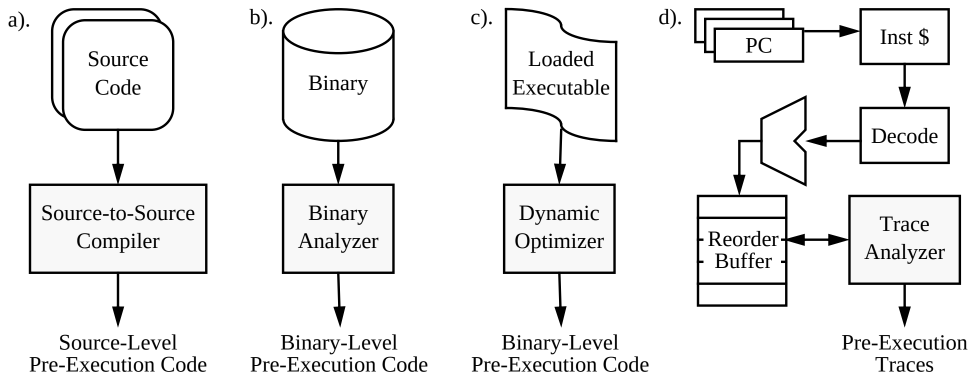


Fig. 1. Four approaches for automatically extracting pre-execution code: a). compiler-based extraction, b). linker-based extraction, c). dynamic optimizer-based extraction, and d). hardware-based extraction. The techniques differ in when the code extraction is performed.

struction traces as they execute at runtime. The extracted code, consisting of trace fragments, are cached and used for pre-execution during the same program run.

Each approach in Figure 1 exhibits very different characteristics due to the fact that code extraction is performed using different analysis techniques. This leads to several tradeoffs:

. **Information for code extraction**. Hardware- and dynamic optimizer-based extraction can exploit runtime information to construct pre-execution traces. By performing code extraction earlier, compiler- and linker-based extraction do not have direct access to runtime information, and must rely on off-line profiling if runtime information is desired, which is cumbersome and potentially inaccurate. On the other hand, compiler- and linker-based extraction can exploit information about high-level program structure such as loops or procedures. At runtime, program structure information is typically not available.

. **Portability of extracted code**. The earlier code extraction is performed, the more portable the generated pre-execution code is. Compiler-based extraction generates source-level pre-execution code that can be compiled onto multiple target ISAs. Linker- and dynamic optimizer-based extraction relies on binary analysis, so a different extractor is necessary for every target ISA. Worse yet, hardware-based extraction requires a different hardware trace analyzer for every processor implementation.[2]

. **Language dependence of extracted code**. While source-level pre-execution code is portable, it is also language specific. Compiler-based extraction analyzes application source code. Thus, a different compiler front-end is required for each supported language. On the other hand, linker-, dynamic optimizer-, and hardware-

[2]Note, the *performance* of compiler-generated pre-execution code may not port completely across different platforms without re-extraction for each platform. Nevertheless, the compiler approach permits the same pre-execution code to at least run on multiple platforms. For this reason, we claim the compiler approach affords portability.

based extraction are language independent since they analyze binaries or traces.

. **Transparency to the user**. The later code extraction is performed, the more transparent the technique is from the user's standpoint. Since runtime extraction of pre-execution code happens in a dynamic optimizer or hardware, it is transparent to the user. Linker-based extraction is less transparent than the runtime approaches because it requires an off-line binary analysis step. However, it is more transparent than compiler-based extraction because it does not require source code, making it a viable approach even when source code is not available.

. **Hardware complexity of the approach**. Compiler-, linker-, and dynamic optimizer-based extraction are software techniques, so they reduce complexity since the hardware is not involved in constructing pre-execution code. In contrast, hardware-based extraction requires adding trace-processing hardware to each processor, increasing hardware complexity.

As demonstrated by these tradeoffs, each approach in Figure 1 offers different (and often complementary) advantages. To evaluate the effectiveness of pre-execution thoroughly, we believe all approaches should be pursued. In this article, we undertake a major investigation of the compiler-based approach for automating pre-execution. We present the design, implementation, and evaluation of several source-to-source C compilers for automatically extracting pre-execution code. Our goal is to develop effective compiler algorithms, understand their performance for different workloads, and quantify the advantages of the compiler-based approach enumerated above.

We believe research of this nature is important for the following reasons. First, given the advantages of compiler-based extraction mentioned earlier (*i.e.*, exploitation of program information, portability, and reduced hardware complexity), we believe compilers will play an important role in enabling pre-execution for future high-performance processors. Being one of the first comprehensive studies on the compiler-based approach, this article lays the initial groundwork for such pre-execution techniques. And second, despite the importance of automating pre-execution, relatively little work has been devoted to software techniques for extracting pre-execution code automatically [Kim and Yeung 2002; Liao et al. 2002; Roth and Sohi 2002]. Most previous work has either assumed manual construction of pre-execution code, or has focused on hardware techniques. By studying compiler-based pre-execution, this article helps bridge the gap between our understanding of pre-execution hardware and how to generate code for it automatically using software tools.

This article represents an extension of our original work on compiler-based pre-execution [Kim and Yeung 2002] which, to our knowledge, is the first work to automate pre-execution using a compiler. In this article, we make the following contributions:

(1) We propose several compiler algorithms for automatically extracting pre-execution code, broken into two categories: aggressive and reduced. Our aggressive algorithms exploit profile information to identify problematic memory references and estimate work inside loops. In addition, they include three powerful performance algorithms: program slicing, prefetch conversion, and speculative

loop parallelization. These performance algorithms enhance the ability of the pre-execution threads to get ahead of the main thread, thus triggering cache misses sufficiently early to tolerate their latency.

(2) In addition to the aggressive algorithms, we propose several "reduced" compiler algorithms for constructing pre-execution code that are less aggressive, and thus enable us to analyze sensitivity to the type of algorithms. Instead of program slicing, which requires sophisticated analysis, we rely on back-end code optimizations performed during C compilation to remove the code that is unnecessary to execute cache-missing memory references, thus simplifying compiler implementation. Also, we develop simple compile-time heuristics to approximate runtime information, thus eliminating the profiling step and streamlining the compiler.

(3) We present 5 prototype compilers, each constructed using a mix of our aggressive and reduced algorithms. Our prototype compilers are built from three toolsets: Unravel [Lyle and Wallace 1997], a commercially available program slicer, SimpleScalar [Burger and Austin 1997], and the Stanford University Intermediate Format (SUIF) framework. Using our prototype compilers, we conduct a detailed experimental evaluation of our compiler algorithms using 13 benchmarks from the SPEC CPU2000 [SPEC 2000] and Olden [Rogers et al. 1995] suites on an architectural simulator of an SMT processor. Our evaluation quantifies the performance of our aggressive compiler, measures the performance impact of our reduced compilers, and studies the contributions to overall performance of individual algorithms.

(4) In addition to evaluating pre-execution for individual benchmarks, we also apply our techniques in the context of multiprogramming. We introduce a new metric, Threading Duty Factor (TDF), to measure the portion of program execution where pre-execution threads are active. This conveys the demand an application places on the idle hardware contexts to run pre-execution threads. Using 10 multiprogrammed workloads consisting of applications with different TDF values, we study the profitability of combining pre-execution with simultaneous execution of multiple programs.

The remainder of this article is organized as follows. First, Section 2 presents an overview of compiler-based pre-execution, briefly describing all our compiler algorithms. Next, Sections 3 and 4 discuss the most aggressive algorithms in detail: program slicing, slicing-based prefetch conversion, and speculative loop parallelization. Then, Section 5 describes a prototype compiler that implements these algorithms, and Section 6 evaluates the prototype compiler's performance. After discussing our most aggressive compiler, Section 7 presents and evaluates our reduced compilers. This is followed by an evaluation of pre-execution in the context of multiprogramming in Section 8. Finally, Section 9 discusses related work, and Section 10 concludes the article.

2. COMPILER ALGORITHMS FOR PRE-EXECUTION

Our compiler algorithms address two performance considerations–*cache-miss coverage* and *pre-execution effectiveness*–and one correctness consideration–*side effects.*

This section provides an overview of these algorithms. We first introduce the performance algorithms, presenting the algorithms for cache-miss coverage in Section 2.1 and the algorithms for pre-execution effectiveness in Section 2.2. Then, in Section 2.3, we discuss how our compilers ensure correctness of the pre-execution code they generate.

2.1 Cache-Miss Coverage

For pre-execution, high cache-miss coverage results from two factors. First, pre-execution code generated by the compiler should execute those static loads that suffer a large number of cache misses; hence, our compilers must identify the most problematic static loads in the application source code. Second, pre-execution code should also compute the address streams of all identified problematic loads accurately.

2.1.1 *Identifying Problematic Loads.* Our work explores two approaches for identifying problematic static loads. One approach uses *summary cache-miss profiles* [Abraham et al. 1993] to directly characterize memory behavior. To acquire summary cache-miss profiles, we execute each program in a separate profiling run prior to compiler analysis. During the profiling run, the number of cache misses is accumulated for each static load in the application, thus summarizing cache behavior on a per-load basis. Later, the cache-miss summaries are used by our compiler to identify the program's most problematic static loads. Our most aggressive prototype compiler, described in Section 5, uses this profile-based approach to identify problematic loads. In contrast, the other approach relies solely on compile-time analysis to identify problematic loads without cache-miss profiles. We develop static analysis combined with simple heuristics to predict cache behavior. While predicting cache behavior exactly at compile time is intractable due to the dynamic nature of memory hierarchies, we find our compiler-based approach can effectively identify problematic loads in many cases. In Section 7, we will further discuss our compiler-based approach for identifying problematic loads.

2.1.2 *Generating Accurate Pre-Execution Code.* In addition to identifying problematic load instructions, our compilers must also generate pre-execution code that will compute the address streams of problematic loads accurately. Similar to most existing pre-execution techniques, our compilers generate separate code for pre-execution threads, and they do so via *code cloning.* (This is in contrast to having pre-execution threads execute the same code executed by the computation thread, as is done in [Luk 2001]). Cloning produces accurate pre-execution code trivially by copying the main computation thread code. Furthermore, cloning decouples pre-execution and computation thread code, preventing transformations on pre-execution code from affecting the main thread. Most of the performance optimizations we will overview in Section 2.2 would not be possible if our compilers did not perform cloning. Cloning, however, suffers from an increased instruction working set size. Fortunately, we have not observed significant degradations in I-cache performance due to cloning for the benchmarks we study.

Our compilers perform cloning at the granularity of a *pre-execution region*, a code fragment encompassing the problematic load instruction or group of load instructions that defines the scope for pre-execution. In our current compilers, a

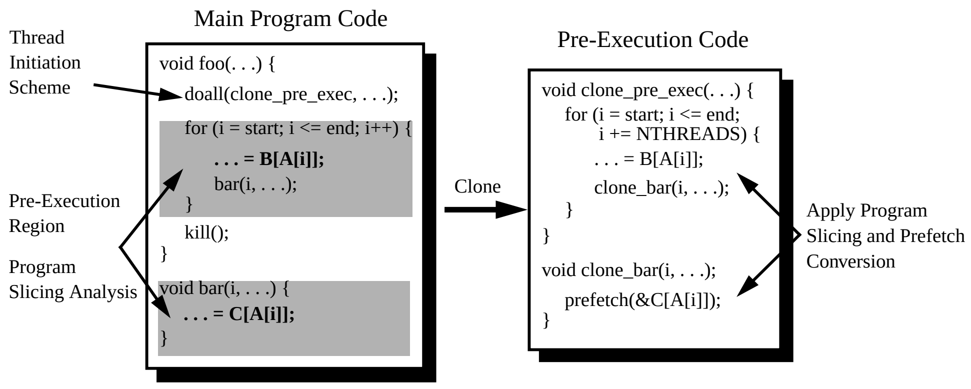


Fig. 2. Code analyses and transformations performed by our compilers. Profiling or static analysis identifies cache-missing memory references (bold-faced code). Pre-execution region cloning guarantees accurate execution of problematic loads. Program slicing or back-end code optimizations during C compilation remove non-critical code and enable prefetch conversion. Speculative loop parallelization creates thread-level parallelism for tolerating the latency of blocking load instructions.

pre-execution region is always a loop containing the problematic load(s). A simple code example appears in Figure 2, showing a pre-execution region and the code cloning step performed upon it (this example also illustrates other code transformations discussed later). In Figure 2, the reference `B[A[i]]` in the main program is a frequent cache-missing memory reference (identified via profiling or compiler analysis as described earlier), and the shaded loop defines the pre-execution region for this reference. Notice a pre-execution region spans multiple procedures whenever loops call procedures containing problematic loads. In Figure 2, the memory reference `C[A[i]]` in the `bar` procedure also misses the cache frequently, and is included in the pre-execution region since `bar` is called from the same loop. Our compilers generate pre-execution code by cloning the shaded code, as shown in the right half of Figure 2.

2.2 Pre-Execution Effectiveness

Besides cache-miss coverage, another important performance consideration is pre-execution effectiveness. For pre-execution to be effective, pre-execution threads must trigger cache misses sufficiently early so that their latency can be tolerated. Our compilers employ three types of optimizations on pre-execution code to enhance pre-execution effectiveness. Two types of optimizations increase the speed of a single pre-execution thread relative to the main thread by removing unnecessary code and blocking associated with problematic loads. A third type of optimization speeds up pre-execution progress by overlapping the latency of blocking loads using multiple pre-execution threads.

2.2.1 *Removing Unnecessary Code.* Pre-execution threads need only execute the critical computations leading up to problematic load instructions; all other computations can be removed, allowing pre-execution threads to run more rapidly. We investigate two approaches for streamlining pre-execution code. The first approach removes unnecessary code explicitly at the source-code level, using a technique called *program slicing* [Binkley and Gallagher 1996; Weiser 1984]. The second approach relies on back-end code optimizations applied during compilation of the pre-execution code into machine code to perform code removal.

Program slicing is a software evaluation technique with numerous applications in debugging and testing of high-integrity software, as well as automatic parallelization. The goal of program slicing is to extract a code fragment, or *a program slice*, from a program based on a *slice criterion.* The slice criterion identifies an intermediate result in the original program, and the program slice is the subset of source code lines from the original program responsible for computing the slice criterion. We use program slicing to streamline pre-execution code. By specifying the memory address of problematic load instructions as slice criteria, our slicing analysis identifies the critical code necessary to execute problematic loads. All program statements excluded from such memory-driven slices do not affect problematic load execution, and can be removed from the pre-execution code during cloning, as illustrated in Figure 2. We employ program slicing in our most aggressive compiler by integrating into the compiler an existing program slicer, called *Unravel* [Lyle and Wallace 1997]. Section 3 will describe Unravel, and how we adapt it to extract program slices for pre-execution.

As we will see in Section 3, program slicing is complex, and slicing tools such as Unravel require significant effort to integrate into compilers. Hence, we would like program slicing to provide a benefit that justifies its implementation effort. However, we observe that program slicing can be redundant. Since the pre-execution code our compiler generates is C code, it must be translated into machine code by a C compiler. Most C compilers perform back-end code optimizations, such as *dead code elimination*, that already provide a code removal benefit. In many cases, Unravel removes code that dead code elimination would have removed anyways. To understand the extent to which program slicing is redundant, we investigate a simpler approach that does not perform program slicing, but instead relies on conventional code optimizations to remove unnecessary code as a consequence of C code compilation. Section 7 will discuss and evaluate this simpler approach.

2.2.2 *Removing Blocking Loads.* In addition to removing unnecessary code, another way to speed up pre-execution threads is to reduce blocking using prefetch instructions. Since prefetch instructions are non-blocking, they allow the pre-execution thread to trigger cache misses and continue executing, saving the pre-execution thread from having to wait for the data to be fetched. However, prefetch instructions are only effective if the prefetched data is not needed by the pre-execution thread shortly after the prefetch; otherwise, the pre-execution thread will still block whether or not a prefetch is inserted.

Our compiler performs *prefetch conversion* on pre-execution code, as illustrated in Figure 2, to replace as many blocking loads as possible with prefetch instructions. Prefetch conversion is considered for all problematic load instructions, but is applied only in those cases where it is profitable. This article studies two algorithms for identifying candidates for prefetch conversion, one coupled with program slicing (described in Section 3) and the other that can be performed in the absence of program slicing (described in Section 7).

2.2.3 *Tolerating Memory Latency.* Removing unnecessary code and blocking loads both attempt to increase the speed of a single pre-execution thread relative to the main computation thread by optimizing pre-execution code. In many cases, however, these optimizations alone do not provide the speed advantage necessary for effective pre-execution. The problem we have observed is blocking loads that cannot be converted into prefetches. If blocking loads remain in the pre-execution code after optimizations have been applied, the pre-execution thread will stall, preventing it from getting sufficiently far ahead of the main thread.

Pre-execution code with blocking loads can be handled using multiple pre-execution threads, allowing individual threads to block independently and overlap their long-latency memory operations. We extract thread-level parallelism by parallelizing loops, and initiating multiple pre-execution threads to execute separate loop iterations simultaneously. Currently, our compilers recognize two forms of loop-level parallelism: *doall* and *doacross*. We apply loop parallelization transformations similar to those employed in conventional parallelizing compilers [Cytron 1986; Padua et al. 1980] to exploit these forms of parallelism. Figure 2 illustrates how our compilers insert parallelization directives to initiate multiple pre-execution threads.

A key difference between our compilers and previous parallelizing compilers is we can apply loop parallelization much more aggressively. Because pre-execution threads run speculatively (see Section 2.3), they in turn permit our compilers to parallelize loops speculatively (*i.e.*, even when our compilers cannot guarantee the legality or safety of transformations under all circumstances). In Section 4, we will discuss our speculative parallelization techniques in greater detail.

2.3 Correctness

The algorithms introduced in Sections 2.1 and 2.2 are designed to generate pre-execution code for high performance. In addition to performance, another important design consideration is correctness. Specifically, our algorithms must not compromise the correctness of the main computation in the process of optimizing pre-execution code. To preserve the integrity of the main computation, we rely on both the architecture to provide a speculative pre-execution model as well as the compiler to remove side effects from pre-execution code.

2.3.1 *Speculative Pre-Execution Model.* As in previous pre-execution techniques [Collins et al. 2001; Collins et al. 2001; Kim and Yeung 2002; Liao et al. 2002; Luk 2001; Roth and Sohi 2001; 2002; Zilles and Sohi 2001], we use a Simultaneous Multithreading (SMT) processor to run pre-execution threads alongside the main thread. We assume pre-execution threads run speculatively, using techniques previously proposed to support speculation. In particular, our SMT processor provides the following three speculative pre-execution model features:

(1) Results computed by pre-execution threads are never integrated into the main thread.

(2) Exceptions signaled in pre-execution contexts terminate the faulting pre-execution thread but do not disrupt main thread execution.

(3) `kill` instructions executed by the main computation thread halt active pre-execution threads.

The first two features isolate pre-execution threads from the main computation thread, preventing incorrect results or exceptions generated by pre-execution code from disrupting the main computation. The last feature allows the main computation to reclaim execution resources from runaway pre-execution threads. As shown in Figure 2, our compilers insert a `kill` directive that halts pre-execution threads still active after the main computation thread leaves a pre-execution region.

2.3.2 *Removing Side Effects.* The architectural support described in Section 2.3.1 provides a degree of isolation between pre-execution threads and the main computation thread; however, under this architecture model, pre-execution threads can still impact the main computation due to side effects through memory. Because pre-execution threads share memory with the main thread, we must guarantee pre-execution threads never write to main thread data structures. Previous pre-execution techniques have proposed hardware support to protect the main computation from stores executed by pre-execution threads [Luk 2001]. In contrast, we rely on the compiler to provide memory isolation. Our compilers perform *store removal* to eliminate memory side effects from pre-execution code.

Aside from memory side effects, there are no other correctness issues that our compilers need to consider. All other correctness assurances are provided by the hardware, as described in Section 2.3.1. This enables our compilers to be extremely aggressive. For example, many of the compiler optimizations for pre-execution effectiveness described in Section 2.2 are not legal or safe under all circumstances. Nevertheless, our compilers can apply them aggressively due to the speculation hardware support, permitting our compilers to make performance tradeoffs freely without worrying about their impact on correctness.

3. PROGRAM SLICING

Having presented an overview of our compiler algorithms in Section 2, we now describe them in greater detail. We begin by exploring the key performance algorithms used in our most aggressive compiler: program slicing, slicing-based prefetch conversion, and speculative loop parallelization. This section discusses program slicing and slicing-based prefetch conversion, while Section 4 will discuss speculative loop parallelization. Then, following the implementation and evaluation of our aggressive compiler (Sections 5 and 6), Section 7 will explore the remaining compiler algorithms which are used in our reduced compilers.

3.1 Unravel

To perform program slicing, our aggressive compiler uses Unravel, a publicly available program slicer for ANSI C from the National Institute of Standards and Technology (NIST).[3] Unravel is a software evaluation tool designed to assist programmers in debugging and program understanding tasks. It consists of an *analyzer*, which parses all .c and .h source files in the application and generates a program dependence graph (PDG) [Ferrante et al. 1987], and a *slicer*, which traverses the PDG iteratively, performing data and control flow analyses to extract the program

[3]Source code for Unravel can be downloaded from
http://www.itl.nist.gov/div897/sqg/unravel/unravel.html.

slice.

3.1.1 *Basic Analysis.* Unravel's slicer performs the basic program slicing algorithm [Lyle et al. 1995] presented below in Equations 1 and 2.

$$S_{<m,v>} = \begin{cases} S_{<n,v>} & \text{if } v \notin defs(n) \\ Sdef_{<n,v>} & \text{otherwise} \end{cases} \tag{1}$$

$$Sdef_{<n,v>} = \{n\} \bigcup \left(\bigcup_{x \in refs(n)} S_{<n,x>} \right) \bigcup \left(\bigcup_{y \in refs(k)} \bigcup_{k \in control(n)} S_{<k,y>} \right) \tag{2}$$

In Equation 1, $S_{<m,v>}$ denotes the program slice for the slice criterion $<m, v>$, or variable v at statement m. The algorithm considers all statements n which are predecessors of m. If n does not assign v, we omit n from the slice, and we recursively evaluate $S_{<n,v>}$, the program slice for variable v at statement n. Otherwise, if n assigns v, we follow Equation 2. In this case, we add n to the slice, and we recursively evaluate the program slice for all referenced variables x used to compute v at statement n (the first two terms in Equation 2). This captures those statements that affect the dataflow to statement n. In addition, we also recursively evaluate the program slice for all referenced variables y at all statements k which control the execution of n, denoted by the $control(n)$ function (the last term in Equation 2). This captures those statements that affect the control flow to statement n.

3.1.2 *Advanced Analysis.* In addition to the basic slicing algorithm presented in Equations 1 and 2, Unravel's slicer also performs several advanced analyses intended to provide more exact dependence information, thus detecting fewer false dependences and improving the quality of program slices. Specifically, Unravel's advanced analyses address the following language features found in C code:

. **Arrays and Structures**. Unravel performs index analysis on array references, and resolves different structure fields. Hence, an assignment or reference to an array element or structure field does not access the entire array or structure, but only the individual element.

. **Pointers**. Unravel performs pointer analysis for statically allocated objects. For every assignment and reference through a pointer to a static object, Unravel keeps track of the set of objects that can possibly be reached. This analysis takes into consideration accesses through multiple levels of indirection, treating the objects at each indirection level separately. Unravel uses this information to prune away those objects that cannot be reached at each pointer access, thus disambiguating accesses to separate objects.

. **Procedures**. Unravel constructs program slices across procedure boundaries. To enable inter-procedure slices, Unravel performs inter-procedure analysis, matching actual parameters with formal parameters and handling return values at call sites to track data dependences across procedure calls.

Although the analyses performed by Unravel are quite sophisticated, one noteworthy limitation is that Unravel ignores indirect procedure calls. Program slicing terminates at the boundary of any procedure called indirectly, which occurs with some frequency in our benchmarks.

3.2 Slicing for Pre-Execution

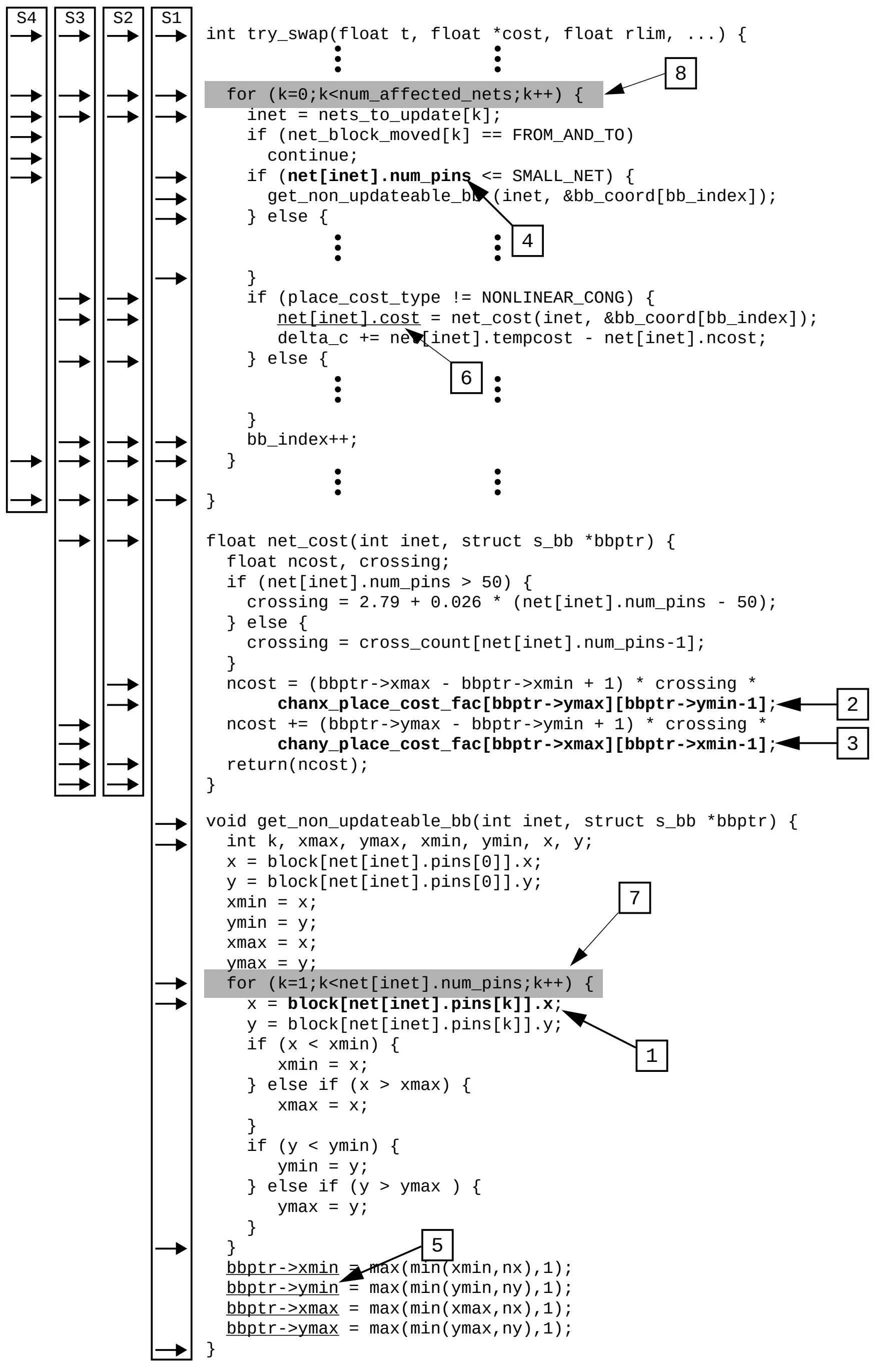
We use Unravel to compute program slices for memory references that suffer frequent cache misses by specifying each memory reference to Unravel as a separate slice criterion. We modified Unravel to address five issues related to our memory-driven program slices: slice criterion specification, store removal, slice termination, slice merging, and code pinning. This section describes our modifications using the code example in Figure 3 from VPR, a SPEC CINT2000 benchmark.

Slice Criterion Specification. As described in Section 2.1.1, we rely on either cache-miss profiles or static analysis to identify problematic loads. Our aggressive compiler employs profiling, in which case, the problematic loads are identified by the profiling tool as load PCs. We translate each of these identified load PCs into a source code line number and variable name using debugging information. In Figure 3, four frequent cache-missing memory references in the VPR application appear in bold-face, labeled “1”-“4.” These memory references occur across three different procedures, `try_swap`, `net_cost,` and `get_non_updateable_bb`. Each memory reference is used as the slice criterion during a single slicing run, described below.

Store Removal. As discussed in Section 2.3, pre-execution threads should never modify memory state visible to the main thread to ensure correct main thread execution. Our SUIF passes, described in Section 5, remove all stores to statically allocated global variables, and stores to heap variables through pointers when generating pre-execution code. Such store removal enables more aggressive program slicing. In addition to removing code off the critical path of cache-missing memory references, our program slicer can also remove code associated with stores that will eventually be eliminated by SUIF. Hence, before running the slicer, we delete all DEFs to global and heap variables in the PDG produced by Unravel. When we run the slicer, all code associated with the removed DEFs will themselves be sliced away. In Figure 3, the underlined references labeled “5” and “6” represent stores to heap and global variables, respectively. Our slicer removes the DEFs associated with these references.

While store removal is necessary for main thread correctness, it can disrupt pre-execution code correctness. For example, the computations at “5” in Figure 3 are necessary to execute the cache-missing memory references at “2” and “3.” By removing the stores at “5,” the cache misses will not be correctly pre-executed each time `net_cost` is entered following a call to `get_non_updateable_bb`. Fortunately, we find memory references “2” and “3” are exceptional cases, and that dataflow through global or heap variables within a pre-execution region rarely lead to cache-missing memory references. (See Section 6.2.3 for results that support this observation, and for further discussion on why this observation is true). In exceptional cases like those in VPR, the speculative nature of pre-execution threads ensures that incorrect pre-execution code never compromises main thread integrity.

Slice Termination. After modifying the PDG to reflect store removal, we run the slicer once for every criterion corresponding to a problematic load instruction.



```
int try_swap(float t, float *cost, float rlim, ...) {
            ⋮           ⋮
  for (k=0;k<num_affected_nets;k++) {
    inet = nets_to_update[k];
    if (net_block_moved[k] == FROM_AND_TO)
      continue;
    if (net[inet].num_pins <= SMALL_NET) {
      get_non_updateable_bb(inet, &bb_coord[bb_index]);
    } else {
            ⋮           ⋮
    }
    if (place_cost_type != NONLINEAR_CONG) {
       net[inet].cost = net_cost(inet, &bb_coord[bb_index]);
       delta_c += net[inet].tempcost - net[inet].ncost;
    } else {
            ⋮           ⋮
    }
    bb_index++;
  }
            ⋮           ⋮
}

float net_cost(int inet, struct s_bb *bbptr) {
  float ncost, crossing;
  if (net[inet].num_pins > 50) {
    crossing = 2.79 + 0.026 * (net[inet].num_pins - 50);
  } else {
    crossing = cross_count[net[inet].num_pins-1];
  }
  ncost = (bbptr->xmax - bbptr->xmin + 1) * crossing *
       chanx_place_cost_fac[bbptr->ymax][bbptr->ymin-1];
  ncost += (bbptr->ymax - bbptr->ymin + 1) * crossing *
       chany_place_cost_fac[bbptr->xmax][bbptr->xmin-1];
  return(ncost);
}

void get_non_updateable_bb(int inet, struct s_bb *bbptr) {
  int k, xmax, ymax, xmin, ymin, x, y;
  x = block[net[inet].pins[0]].x;
  y = block[net[inet].pins[0]].y;
  xmin = x;
  ymin = y;
  xmax = x;
  ymax = y;
  for (k=1;k<net[inet].num_pins;k++) {
    x = block[net[inet].pins[k]].x;
    y = block[net[inet].pins[k]].y;
    if (x < xmin) {
       xmin = x;
    } else if (x > xmax) {
       xmax = x;
    }
    if (y < ymin) {
       ymin = y;
    } else if (y > ymax ) {
       ymax = y;
    }
  }
  bbptr->xmin = max(min(xmin,nx),1);
  bbptr->ymin = max(min(ymin,ny),1);
  bbptr->xmax = max(min(xmax,nx),1);
  bbptr->ymax = max(min(ymax,ny),1);
}
```

Fig. 3. VPR code example. Labels "1"-"4" indicate cache-missing memory references selected for slicing. Labels "5" and "6" indicate memory references requiring store removal. Labels "7" and "8" indicate loops that bound the scope of slicing. Labels "S1," "S2," "S3," and "S4" show the slice result for the selected memory references.

For each slicer run, Unravel computes a program slice across the entire program. Such slices are too large; in fact, we are interested in slicing only the code that will eventually form the pre-execution region for the problematic load. Unfortunately, Unravel does not know the extent of pre-execution regions–these are determined in a separate compiler pass. However, as described in Section 2.1.2, a pre-execution region is defined by a loop containing one or more problematic load instructions. As we will see later in Section 4.2, our pre-execution region selection algorithm chooses either the inner-most loop or the next-outer loop encompassing a problematic load to serve as its pre-execution region. Hence, we modified Unravel to terminate slicing once two nested looping statements above the slice criterion have been encountered (if two nested looping statements cannot be found, we terminate slicing after one looping statement).

Figure 3 illustrates slice termination for the VPR benchmark. Memory reference "1" is contained inside the loop labeled "7." The next outer loop, labeled "8," is where slicing terminates for this memory reference. Memory references "2," "3," and "4" are contained inside the loop labeled "8." The next outer loop, which is not shown in Figure 3, is where slicing terminates for these three memory references.

As illustrated in Figure 3, our slice termination policy permits slices to span multiple procedures (there is no limit on call depth). From our experience, inter-procedure analysis is important because loops can be nested across procedure boundaries in some cases, particularly in non-numeric applications like VPR. When slicing across procedures, however, multiple paths can occur if a procedure is called from multiple sites. Our slicer pursues all call paths and searches for the two nested looping statements along every path, possibly identifying multiple loops where slicing terminates for a single problematic load instruction. Smaller slices could be constructed if the slicer only considers the most frequently executed paths (this applies to paths within procedures as well as across procedures); however, this would require path profiles which are not currently supported in our compiler.

Slice Merging. After slicing analysis completes, we have a program slice for each sliced memory reference. Figure 3 illustrates the slices computed for the four cache-missing memory references in VPR by placing an arrow to the left of each source code line contained in the slice. The slices for memory references "1"-"4" are specified by the columns of arrows labeled "S1," "S2," "S3," and "S4," respectively. (Note, slices S2, S3, and S4 should continue up to the next outer loop). Unravel stores each program slice as a bitmask with one bit per line of source code in the program.

Since invoking pre-execution threads for each individual slice may incur significant overhead, we merge multiple slices and invoke pre-execution threads once to cover all the problematic loads within each merged slice together. Slice merging occurs at the granularity of pre-execution regions. Once the pre-execution regions have been selected (see Section 4.2.1 for our selection algorithm), we "OR" together the bitmasks of all slices whose problematic loads reside in the same pre-execution region. We also clear any bits that lie outside of the selected pre-execution region. For example, if the loop labeled "8" in Figure 3 were selected as a pre-execution region, we would merge the bitmasks from slices S1 – S4 since memory references "1" – "4" are included within loop "8." This merged slice would contain 28 out of

```
a) void get_non_updateable_bb(int inet, struct s_bb *bbptr) {
     int k, x;
                                                         1
     for (k=1;k<net[inet].num_pins;k++) {
       x = block[net[inet].pins[k]].x;
       asm(" " : : "r" (block[net[inet].pins[k]].x));
     }
   }

b) void get_non_updateable_bb(int inet, struct s_bb *bbptr) {
     int k, x;

     for (k=1;k<net[inet].num_pins;k++) {
       prefetch(&block[net[inet].pins[k]].x);
     }
   }                                             2
```

Fig. 4. Code generated by our aggressive compiler for the `get_non_updateable_bb` function from Figure 3. a) Pre-execution code after program slicing with `asm` macro added for code pinning (label "1"). b) Pre-execution code after program slicing and prefetch conversion (label "2"). Bold-face code denotes cache-missing memory references.

the original 57 lines of code in Figure 3.

Code Pinning. Since our compilers are source-to-source compilers, the generated pre-execution code is C code, and must eventually be translated into machine code by a C compiler. Our system uses the *gcc* compiler for this purpose. Unfortunately, pre-execution code by its very nature is dead code since store removal eliminates all side effects, and is thus likely to be removed during C code compilation (we compile pre-execution code with the "-O2" flag which activates dead code elimination in *gcc*). For example, after store removal and program slicing, the `get_non_updateable_bb` function in Figure 3 is reduced to a single loop that "touches" the elements in the `block` array. Since this code performs no useful computation, *gcc* removes it.

To prevent pre-execution code removal during C compilation, we insert an `asm` macro that artificially consumes the loaded value from each problematic load instruction, thus "pinning" the load and all associated pre-execution code. Figure 4a illustrates how our compilers perform code pinning. In Figure 4a, we show the pre-execution code for the `get_non_updateable_bb` function from Figure 3 after program slicing. An `asm` macro containing a null instruction, labeled "1," has been added to consume the data from the `block` array memory reference (*i.e.*, the slice criterion used by the program slicer). Since *gcc* does not remove `asm` code, the `asm` code in turn prevents the removal of the pre-execution code due to the data dependence between the null instruction and the `block` array memory reference.

3.3 Prefetch Conversion

Program slicing removes non-critical computations from pre-execution code, resulting in more efficient pre-execution threads. Another way to speed up pre-execution threads is to reduce blocking by using prefetch instructions. However, as described in Section 2.2.2, such a *prefetch conversion* optimization is profitable only when the prefetched data is not needed by the pre-execution thread shortly after the prefetch.

Since program slicing performs dependence analysis to identify unnecessary code, it already computes the information necessary for prefetch conversion. As a result, prefetch conversion can be performed trivially when coupled with a program slicer (*e.g.*, Unravel) in the following manner. We consider each problematic load instruction in all pre-execution regions after program slicing has been performed. If the data accessed by the load instruction is not needed by the slice code (*i.e.*, the program statements dependent upon the load have been removed by the program slicer), we convert the blocking load instruction into a non-blocking prefetch. Applying this simple algorithm to the VPR code example in Figure 3, we see that memory references "1," "2," and "3" can be converted into prefetches. Figure 4b illustrates the final sliced pre-execution code for the `get_non_updateable_bb` function after converting the blocking memory reference (label "1" in Figure 3) into a non-blocking prefetch (label "2" in Figure 4b). Notice, our compiler assumes the target architecture supports a prefetch instruction (see Section 5.2), which is in-lined into the pre-execution code using the `prefetch` macro in Figure 4b.

4. PRE-EXECUTION INITIATION AND SPECULATIVE LOOP PARALLELIZATION

In addition to the program slicing and prefetch conversion optimizations presented in Section 3, our aggressive compiler also employs multiple pre-execution threads to tolerate the latency of blocking memory instructions, as described in Section 2.2.3. In fact, our reduced compilers, which we will present later in Section 7, use these same latency tolerance techniques as well. This section describes how we initiate pre-execution threads, including the schemes for creating multiple pre-execution threads through speculative loop parallelization. First, Section 4.1 presents the thread initiation schemes used by our compilers. Then, Section 4.2 discusses the algorithms for assigning thread initiation schemes to pre-execution regions. Finally, Section 4.3 describes how our compilers generate code for each initiation scheme assignment. (Note, while all our compilers perform pre-execution initiation similarly, there are some slight differences. In cases where differences exist, this section presents the approach taken by our most aggressive compiler, leaving a discussion of the reduced compiler approaches to Section 7).

4.1 Thread Initiation Schemes

Our compilers employ three schemes for initiating pre-execution threads: SERIAL, DOALL, and DOACROSS. Figure 5 illustrates these schemes.

Serial. This scheme initiates a single pre-execution thread for each pre-execution region. As shown in Figure 5a, the main thread (solid line) forks a single pre-execution thread (dotted line) prior to entering a pre-execution region. The pre-execution thread then executes the code for the entire pre-execution region sequentially.

For the SERIAL scheme to be successful, the lone pre-execution thread must get ahead of the main thread to trigger cache misses sufficiently early to hide their latency. Program slicing and prefetch conversion provide the pre-execution thread with a speed advantage over the main thread. In many cases, unfortunately, these optimizations alone may not be sufficient. As explained in Section 2.2.3, the problem is blocking loads that program slicing and prefetch conversion are unable to

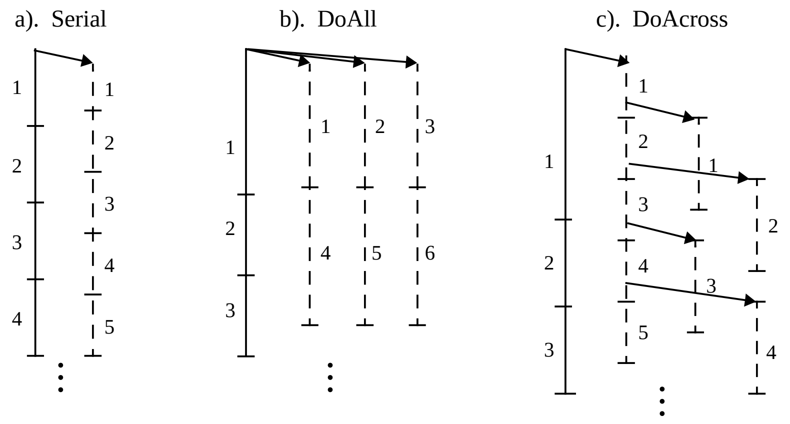


Fig. 5. Three pre-execution thread initiation schemes: a) SERIAL, b) DOALL, and c) DOACROSS. Solid lines denote the main thread, dotted lines denote pre-execution threads, arrows denote thread spawning, and numeric labels denote loop iteration counts.

remove. For example, memory reference "4" in Figure 3 is a problematic memory reference that cannot be converted into a prefetch because the value it loads is needed by the pre-execution code. Such blocking loads will cause the pre-execution thread to stall, preventing it from getting ahead of the main thread.

DoAll. Pre-execution code with blocking loads can be handled using multiple pre-execution threads, allowing individual threads to block independently and tolerate the long-latency memory operations by overlapping memory stalls. Our compilers extract thread-level parallelism for latency tolerance purposes through loop parallelization. Conventional loop parallelization requires the compiler to analyze dependences exactly, which is nearly impossible for the loops we would like to pre-execute due to complex control flow and pointers. Fortunately, our compilers do not need to guarantee correctness thanks to the speculative pre-execution model described in Section 2.3.1, permitting us to parallelize loops speculatively. Our compilers perform loop induction variable analysis during parallelization, but we do not analyze dependences in the loop body and assume (optimistically) that no loop-carried dependences exist except through induction variables.

Our compilers recognize two types of loop induction variables, giving rise to two speculative loop parallelization schemes. The first parallelization scheme, DOALL, speculatively parallelizes affine loops (*i.e.*, loops whose induction variables are updated arithmetically). When our compilers encounter an affine loop, they assume the loop is fully parallel, and generate code to pre-execute the loop iterations independently. As shown in Figure 5b, the main thread forks multiple pre-execution threads prior to entering a pre-execution region, with loop iterations distributed to threads in round robin sequence (denoted by the loop iteration labels). In this scheme, each thread keeps a private copy of the loop induction variable and updates it locally every iteration.

DoAcross. The second parallelization scheme, DOACROSS, speculatively paral-

a). Given: Global loop nest graph, G_L
          Loop iteration count profiles
    Compute: Pre-Execution Region Set, P

```
 1: P = $\Phi$;
 2: for each loop L in $G_L$ from inner-most to outer-most {
 3:   if (level(L) == INNER_MOST) {
 4:     if (iteration_count(L) $\geqslant$ 25)
 5:       P = P $\cup$ {L};
 6:   } else {
 7:     if {P $\cap$ nested_loops(L) == $\Phi$)
 8:       P = P $\cup$ {L};
 9:   }
10: }

11: for each inner-most loop L in $G_L$ {
12:   if (P $\cap$ outer_loops(L) == $\Phi$)
13:     P = P $\cup$ {L};
14: }
```

b). Given: Pre-Execution Region Set, P
    Compute: Serial Loop Set, SE
             DoAll Loop Set, DA
             DoAcross Loop Set, DX
             Procedure Set, F

```
 1: SE = DA = DX = F = $\Phi$;
 2: for each loop L in P {
 3:   if (num_blocking_load(L) == 0)
 4:     SE = SE $\cup$ {L};
 5:   else {
 6:     if (induction(L) == AFFINE)
 7:       DA = DA $\cup$ {L};
 8:     else
 9:       DX = DX $\cup$ {L};
10:   }
11:   F = F $\cup$ called_procedures(L);
12: }
```

Fig. 6. Algorithm for selecting thread initiation schemes. a) Computation of the set of pre-execution regions, P. b) Selection of the thread initiation scheme for each pre-execution region. Φ denotes the empty set.

lelizes pointer-chasing loops (*i.e.*, loops whose induction variables are updated through a pointer dereference). Pointer-chasing loops are serial if for no other reason due to the serial update of loop induction variables. However, they can be speculatively parallelized by overlapping induction variable updates with loop body computations. As shown in Figure 5c, our compiler creates a single thread, called the *backbone thread*, to execute the induction variable update code serially. The backbone thread then forks additional threads at each iteration, called *rib threads*, to execute the loop bodies. Even though induction variable updates are serialized, separate loop bodies execute in parallel. Note in DoAcross, inter-thread communication is required every loop iteration to pass the induction variable value.

4.2 Scheme Selection Algorithm

Figure 6 presents our algorithm to determine the thread initiation schemes for pre-execution. The algorithm operates in two steps. First, we compute the set of pre-execution regions, P, from which we will initiate pre-execution threads. Then, for each pre-execution region in P, we select one of the thread initiation schemes from Section 4.1 that will provide the highest performance possible. The following two sections discuss our algorithms in greater detail.

4.2.1 *Selecting Pre-Execution Regions.* The first step in our scheme selection algorithm is to select the pre-execution regions. As discussed in Section 2.1.2, a pre-execution region is a loop defining the scope of pre-execution for problematic load instructions. Our compilers identify the inner-most and next-outer loops containing one or more problematic load(s), and select one of these loops to serve as a pre-execution region based on two criteria. On the one hand, the likelihood of loop-carried dependences increases as pre-execution threads execute more distant code, reducing the effectiveness of speculative loop parallelization. This favors selecting inner-most loops. On the other hand, loops should contain enough work to amortize pre-execution startup costs. This favors selecting next-outer loops.

To identify the inner-most and next-outer loops, we construct a global loop nest graph, G_L. G_L is a DAG, with nodes representing loops and edges denoting loop nesting. The DAG specifies the nesting relationship between all loops in the entire program, taking into consideration nesting across procedure calls as well as within procedures (we use the program's procedure call graph to capture inter-procedure loop nesting, though we do not account for nesting across indirect calls). Once constructed, G_L is used to identify the inner-most and next-outer loops for all problematic load instructions. To decide which loops will serve as pre-execution regions, we estimate the amount of work performed inside inner-most loops and select the inner-most loop when sufficient work exists to amortize pre-execution startup costs. If the inner-most loop contains insufficient work, we instead select the next-outer loop. Our aggressive compiler uses loop-trip counts acquired via profiling to approximate the work in inner-most loops.[4]

Figure 6a presents our algorithm for computing the set of pre-execution regions, P, given the graph G_L and loop-trip count profiles. This algorithm visits all loops in G_L in inner-most to outer-most order (line 2), and considers 3 cases. First, whenever we visit an inner-most loop containing problematic loads (line 3), we add it to the set of pre-execution regions, P, if it iterates more than some minimum count (lines 4 and 5). We use a threshold of 25 iterations, which works well for most loops we've encountered. Such large inner-most loops contain sufficient work to amortize pre-execution startup costs, so they make good pre-execution region candidates. Second, whenever we visit a next-outer loop containing problematic loads (line 6), we add it to P as long as it does not contain any inner-most loops already selected as pre-execution regions (lines 7 and 8)–*i.e.*, we do not permit nesting of pre-execution regions. Such next-outer loops contain inner loops with insufficient work to amortize pre-execution startup costs, so pre-execution should occur from the next-outer level. Finally, after all loops have been visited, it is possible for some inner-most loops that iterate fewer than 25 times to be excluded from all pre-execution regions because a "sibling loop" in graph G_L was added to P, thus preventing their common next-outer loop from becoming a pre-execution region. We revisit all inner-most loops, and add to P those loops that have been excluded from all pre-execution regions (lines 11–13). This ensures all problematic loads get pre-executed, even if some loads reside in loops with insufficient work to amortize pre-execution startup costs.

4.2.2 *Selecting Thread Initiation Schemes.* After selecting the pre-execution regions, we choose a thread initiation scheme for each pre-execution region using the algorithm in Figure 6b. This algorithm chooses SERIAL for pre-execution regions where program slicing and prefetch conversion have removed all blocking loads (lines 3 and 4). SERIAL performs well in this case, and it has the lowest overhead of all schemes since only one pre-execution thread is initiated. If blocking loads remain after optimization, however, our algorithm speculatively parallelizes the loop at the

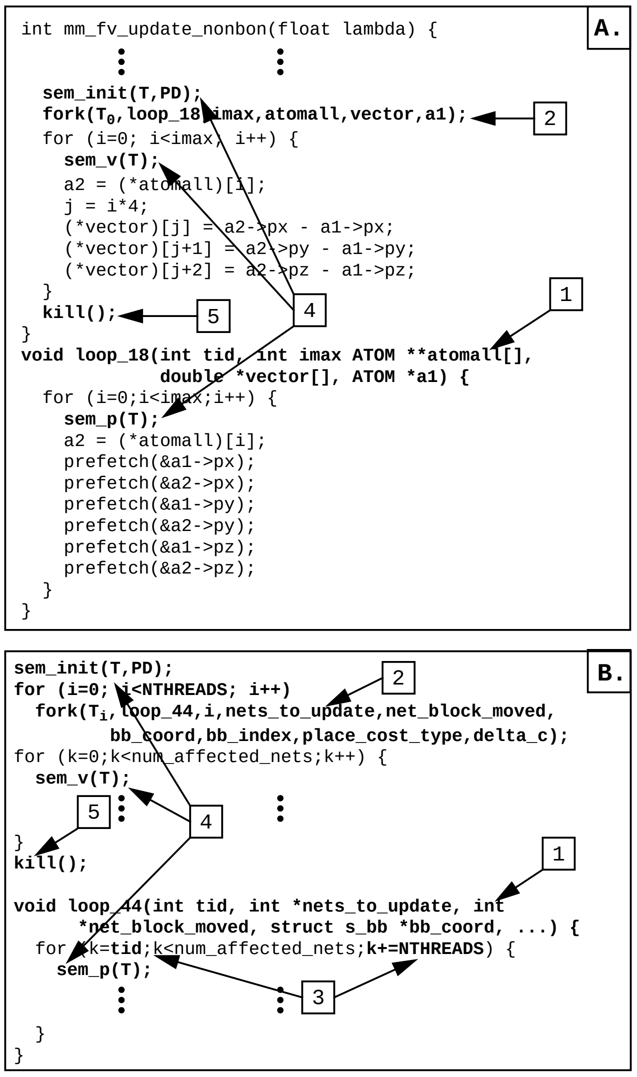
[4]In cases where the loop termination condition is known statically, the compiler can determine the loop-trip count at compile time. Unfortunately, none of the pre-execution regions in our benchmarks have static loop termination conditions, so we did not implement loop termination analysis in our aggressive compiler and instead rely on profiles.

top of the pre-execution region to tolerate the long-latency memory stalls. We use the DoAll (lines 6 and 7) and DoAcross (lines 8 and 9) parallelization schemes for pre-execution regions consisting of affine and pointer-chasing loops, respectively. As described in Section 4.1, DoAll performs well for affine loops because they permit local update of the loop induction variable, thus saving on communication and reducing overhead. DoAcross works best for pointer-chasing loops even though communication is required to pass the induction variable value because it exposes the parallelism between per-iteration loop bodies.

4.3 Generating Code

Once the thread initiation schemes have been selected, our compilers generate pre-execution code. Figures 7 and 8 illustrate the steps involved in generating code for each thread initiation scheme, showing output produced by our aggressive compiler. In Figure 7A, we apply the Serial scheme to a pre-execution region containing a single inner-most loop from the AMMP benchmark, and in Figure 7B, we apply the DoAll scheme to the outer loop of the VPR code example from Figure 3 (several lines in the VPR code example have been removed to conserve space). Code generation for these two schemes is similar and follows five steps, as indicated by the numeric labels in Figure 7. First, we clone the pre-execution region consisting of a loop header and loop body, and place the code in a single procedure (label "1"). Store removal, program slicing, and prefetch conversion are applied to the loop body of the cloned code. We also generate code to fork the pre-execution thread(s) and pass parameters for all local variables in the pre-execution region (label "2"). Then, for the DoAll scheme only, we adjust the loop induction variable update code to distribute iterations to threads in round-robin fashion (label "3"). Next, we insert a counting semaphore (label "4"). This semaphore, called "T," is initialized to the value "PD." Semaphore T blocks pre-execution threads that reach a *prefetch distance* number of iterations ahead of the main thread, preventing them from getting too far ahead. Finally, we add a `kill` directive to halt any active pre-execution threads after the main thread leaves the pre-execution region (label "5").

In Figure 8, we apply the DoAcross scheme to the outer loop of a pre-execution region consisting of two nested pointer-chasing loops from the TWOLF benchmark. Our implementation of DoAcross performs the serialized loop induction variable updates in a separate "backbone" thread which in turn initiates additional "rib" threads to perform the loop body computations in parallel, as described in Section 4.1. Generating code for the backbone and rib threads follows six steps. First, we clone the loop header and loop body, and place them in separate backbone and rib procedures, respectively (label "1"), applying the same store removal and code optimizations from Figures 7A and B on the loop body code. We also generate code to fork a single backbone thread (label "2") and multiple rib threads in round-robin order (label "3"), passing parameters as needed. As in Figures 7A and B, we must insert counting semaphores to synchronize the threads. We insert a single semaphore, called "T_0," to keep the backbone thread from getting more than "PD" iterations ahead of the main thread (label "4"), and multiple semaphores, one per rib thread, to synchronize each rib thread with the backbone thread during communication of the induction variable value (label "5"). Finally, we insert a `kill` directive (label "6").



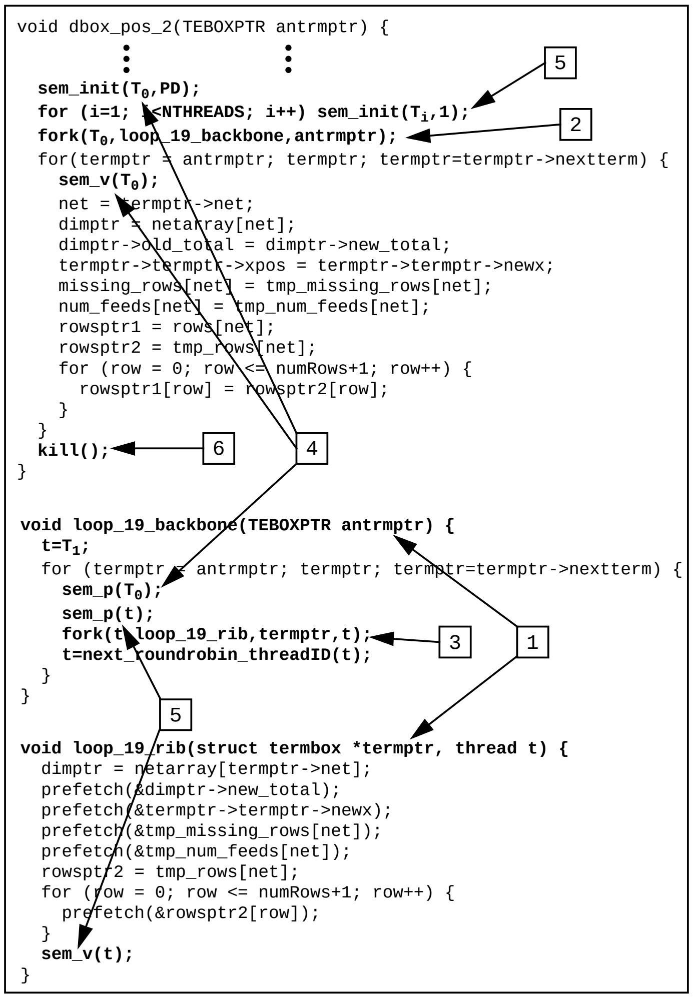
```
int mm_fv_update_nonbon(float lambda) {
        ⋮              ⋮
  sem_init(T,PD);
  fork(T0,loop_18,imax,atomall,vector,a1);
  for (i=0; i<imax; i++) {
    sem_v(T);
    a2 = (*atomall)[i];
    j = i*4;
    (*vector)[j] = a2->px - a1->px;
    (*vector)[j+1] = a2->py - a1->py;
    (*vector)[j+2] = a2->pz - a1->pz;
  }
  kill();
}
void loop_18(int tid, int imax ATOM **atomall[],
             double *vector[], ATOM *a1) {
  for (i=0;i<imax;i++) {
    sem_p(T);
    a2 = (*atomall)[i];
    prefetch(&a1->px);
    prefetch(&a2->px);
    prefetch(&a1->py);
    prefetch(&a2->py);
    prefetch(&a1->pz);
    prefetch(&a2->pz);
  }
}
```

```
sem_init(T,PD);
for (i=0; i<NTHREADS; i++)
  fork(Ti,loop_44,i,nets_to_update,net_block_moved,
         bb_coord,bb_index,place_cost_type,delta_c);
for (k=0;k<num_affected_nets;k++) {
  sem_v(T);
        ⋮              ⋮
}
kill();

void loop_44(int tid, int *nets_to_update, int
      *net_block_moved, struct s_bb *bb_coord, ...) {
  for (k=tid;k<num_affected_nets;k+=NTHREADS) {
    sem_p(T);
        ⋮              ⋮
  }
}
```

Fig. 7. Code generated by the aggressive compiler to implement the A. Serial scheme for the AMMP benchmark, and B. DoAll scheme for the VPR benchmark. Code for initiating pre-execution and synchronizing pre-execution threads appears in bold-face.

In addition to cloning the loop headers and loop bodies as shown in Figures 7 and 8, we must also clone any procedure(s) called from within loop bodies (not shown in the figures). Once cloned, these procedures should also undergo store removal and code optimizations since they are part of the pre-execution regions as well. For example, the `net_cost` and `get_non_updateable_bb` routines from Figure 3 should be cloned and optimized along with the loop from `try_swap`.



```
void dbox_pos_2(TEBOXPTR antrmptr) {
          :               :
  sem_init(T0,PD);
  for (i=1; i<NTHREADS; i++) sem_init(Ti,1);
  fork(T0,loop_19_backbone,antrmptr);
  for(termptr = antrmptr; termptr; termptr=termptr->nextterm) {
    sem_v(T0);
    net = termptr->net;
    dimptr = netarray[net];
    dimptr->old_total = dimptr->new_total;
    termptr->termptr->xpos = termptr->termptr->newx;
    missing_rows[net] = tmp_missing_rows[net];
    num_feeds[net] = tmp_num_feeds[net];
    rowsptr1 = rows[net];
    rowsptr2 = tmp_rows[net];
    for (row = 0; row <= numRows+1; row++) {
      rowsptr1[row] = rowsptr2[row];
    }
  }
  kill();
}

void loop_19_backbone(TEBOXPTR antrmptr) {
  t=T1;
  for (termptr = antrmptr; termptr; termptr=termptr->nextterm) {
    sem_p(T0);
    sem_p(t);
    fork(t,loop_19_rib,termptr,t);
    t=next_roundrobin_threadID(t);
  }
}

void loop_19_rib(struct termbox *termptr, thread t) {
  dimptr = netarray[termptr->net];
  prefetch(&dimptr->new_total);
  prefetch(&termptr->termptr->newx);
  prefetch(&tmp_missing_rows[net]);
  prefetch(&tmp_num_feeds[net]);
  rowsptr2 = tmp_rows[net];
  for (row = 0; row <= numRows+1; row++) {
    prefetch(&rowsptr2[row]);
  }
  sem_v(t);
}
```

Fig. 8. Code generated by the aggressive compiler to implement the DoAcross scheme for the TWOLF benchmark. Code for initiating pre-execution and synchronizing pre-execution threads appears in bold-face.

5. IMPLEMENTATION

This section discusses implementation issues. First, Section 5.1 describes a prototype compiler that implements our most aggressive compiler algorithms presented in Sections 2 through 4. Then, Sections 5.2 and 5.3 discuss the ISA and thread-level support, respectively, assumed by our aggressive compiler.

5.1 Aggressive Prototype Compiler

Figure 9 illustrates our most aggressive prototype compiler. This compiler employs cache-miss and loop-trip count profiles to drive optimizations, and program slicing to remove unnecessary code. We modified the cache simulator provided by the

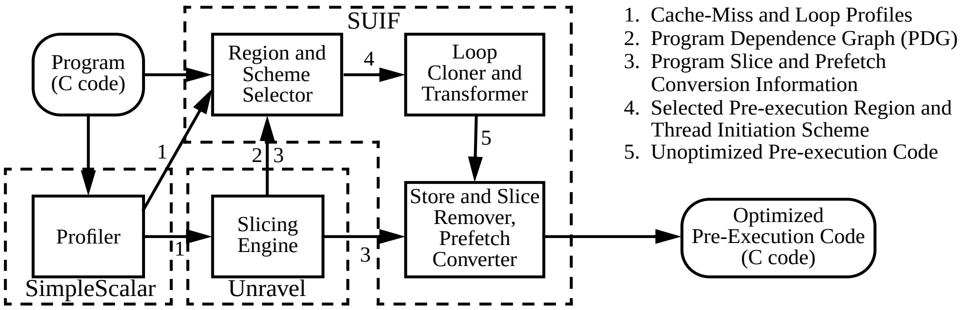

SUIF
Program (C code)
Region and Scheme Selector
Loop Cloner and Transformer
Profiler
Slicing Engine
Store and Slice Remover, Prefetch Converter
Optimized Pre-Execution Code (C code)
SimpleScalar
Unravel
1
2 3
3
4
5
1. Cache-Miss and Loop Profiles
2. Program Dependence Graph (PDG)
3. Program Slice and Prefetch Conversion Information
4. Selected Pre-execution Region and Thread Initiation Scheme
5. Unoptimized Pre-execution Code

Fig. 9. Major components in our most aggressive prototype compiler. This compiler uses Unravel for program slicing and profiling to drive optimizations. Arrows denote interactions between compiler modules.

SimpleScalar toolset [Burger and Austin 1997] to acquire the summary cache-miss profiles described in Section 2.1.1,[5] and the loop-trip count profiles described in Section 4.2.1. We also modified Unravel to implement the program slicing algorithms presented in Section 3.2 and the prefetch conversion analysis described in Section 3.3. Finally, we implemented the remaining algorithms in SUIF. We use SUIF to select the pre-execution regions and thread initiation schemes according to the algorithms in Section 4.2. When selecting pre-execution regions, we discard all regions contributing less than 3% of the application's total cache misses. Filtering out unimportant pre-execution regions helps minimize the runtime overhead incurred by pre-execution threads. We also use SUIF to perform all necessary code transformations. This includes generating code to initiate pre-execution threads as discussed in Section 4.3, as well as removing stores and sliced code, pinning code, and converting blocking loads to prefetches.

5.2 ISA Support

Our compilers assume an SMT processor with the following ISA support. First, we assume a `fork` instruction that specifies a hardware context ID and a PC. The fork initializes the program counter of the specified hardware context to the PC value, and activates the context. Second, we assume `suspend` and `resume` instructions. These instructions are used to "recycle" threads for low overhead thread initiation, which we describe in the next section. Both instructions specify a hardware context ID to suspend or resume. In addition, `suspend` causes a pipeline flush of all instructions belonging to the suspended context. While the processor state of a suspended context remains in the processor, the associated thread discontinues fetching and issuing instructions after the suspend and pipeline flush. Third, we assume a `kill` instruction that halts all currently active pre-execution threads. Only the main thread can execute `kill` instructions. Finally, we assume a `prefetch` instruction, which is a non-blocking load instruction.

[5]Although we use cache simulation to acquire the summary cache-miss profiles, these can also be acquired using profiling tools such as DCPI [Anderson et al. 1997] and Shade [Cmelik and Keppel 1993]. We did not explore these other approaches since profiling efficiency was not a concern in our work.

5.3 Thread-Level Support

Thread initiation can be expensive due to context initialization (our context initialization code contains 25 instructions). To minimize overhead, we "recycle" threads. We create a pre-execution thread for each idle hardware context once during program startup. Each pre-execution thread enters a dispatch loop and suspends itself. To perform a "fork," the forking context communicates a PC value through memory, and executes a `resume` instruction to unblock one of the suspended threads. The "forked" thread then jumps indirect through the PC argument. If the forked thread completes normally, it returns to the dispatch loop and suspends itself until the next fork, thus recycling the thread. If, however, the forked thread is halted by the main thread via a kill instruction, then it cannot simply be resumed. Instead, to prevent the thread from resuming its path of execution prior to the kill, we assume the `kill` instruction sets the thread's PC to point to the instruction immediately following the `suspend` instruction in the dispatch loop. Consequently, a fork performed on a killed thread resumes the thread as if it had returned to the dispatch loop normally.

Inter-thread communication occurs during thread initiation to pass arguments, and during synchronization. In both cases, we perform communication through memory. To pass arguments, we use a memory buffer and communicate values via loads and stores to the buffer. For synchronization, we implement the semaphore primitive from Section 4.3 in software. We allocate a global counter in memory, and during each iteration, the main thread performs a "V" by incrementing the counter. Since our parallelization schemes use the semaphore only for producer-consumer synchronization, we exploit this pattern by maintaining a private counter for each pre-execution thread. When the pre-execution thread performs a "P," it increments its private counter, and compares the count to the global counter. The pre-execution thread continues only if the difference between the counters does not exceed PD, the prefetch distance. Otherwise, the pre-execution thread busy waits. Since busy waiting consumes processor resources and degrades the performance of non-waiting threads, we insert a sequence of long-latency inter-dependent instructions into the busy-wait loop to throttle the fetch rate of busy-waiting threads.

6. AGGRESSIVE COMPILER EVALUATION

This section reports our experimental results for the aggressive prototype compiler described in Section 5.1. First, Section 6.1 describes the benchmarks and architectural simulator used to obtain the results. Next, Section 6.2 presents the performance achieved by our aggressive compiler. Then, Section 6.3 studies the contributions of individual compiler algorithms to overall performance gain. Finally, Section 6.4 examines the impact of architectural support for pre-execution threads.

6.1 Methodology

Table I lists the 13 benchmarks used in our study. These benchmarks have been chosen from the SPEC CINT2000 and CFP2000 suites [SPEC 2000], and the Olden suite [Rogers et al. 1995]. Unfortunately, there are 5 CINT2000, 10 CFP2000, and 8 Olden benchmarks that we could not study. One CINT2000 and all 10 remaining

Table I. Benchmark characteristics.

Suite	Name	Input	FastFwd	Sim	IPC
SPEC CINT2000	256.bzip2	reference	186,166,461	123,005,773	1.3849
	175.vpr	reference	364,172,593	130,044,367	1.4039
	300.twolf	reference	124,205,135	112,809,146	1.1595
	254.gap	reference	147,923,793	127,932,004	3.3558
	197.parser	reference	245,277,302	126,593,730	1.9844
	181.mcf	reference	12,149,459,578	137,280,363	0.7914
	164.gzip	reference	162,442,542	135,592,391	1.9840
SPEC CFP2000	183.equake	reference	2,570,651,646	21,850,552	0.7586
	188.ammp	reference	2,439,723,993	129,357,604	1.3600
	179.art	reference	12,899,865,395	113,811,999	1.0900
	177.mesa	reference	262,597,404	54,117,618	2.8605
Olden Benchmarks	mst	1024 nodes	183,274,940	24,361,256	0.1153
	em3d	20K nodes	53,331,921	108,341,604	0.5929

Table II. SMT simulator settings.

Processor Pipeline			
Issue width	8-way	# hardware contexts	4
RUU size	128 entries	Instruction fetch queue	32 entries
Load-store queue	64 entries	Functional units	8 Int, 4 FP units
Int add/ mult/ div	1/ 3/ 20 cycles	FP add/ mult/ div	2/ 4/ 12 cycles
Branch Predictor			
Gshare predictor	2K entries	Return of stack	8 entries
Branch target buffer	2K entries, 4-way set-associative		
Memory Hierarchy			
Level 1 cache	Split I & D, 32KB, 2-way set-associative, 32B block, 1 cycle latency		
Level 2 cache	Unified, 1MB, 4-way set-associative, 64B block, 10 cycle latency		
Main memory access time	122 cycles		

CFP2000 benchmarks are not written in C. Of the other CINT2000 benchmarks, two could not be processed by the original Unravel tool, one could not be processed by SUIF, and one performs system calls not supported by SimpleScalar. All 8 remaining Olden benchmarks perform recursive tree traversals which our compiler does not analyze. In Table I, the column labeled “Input” reports the inputs used to run each benchmark. The next two columns, labeled “FastFwd” and “Sim,” specify the number of skipped and simulated instructions, respectively, in our simulation regions. The column labeled “IPC” provides the instructions per cycle for each benchmark on our simulator, described below. Finally, when acquiring profiles, both profile and data collection runs use the same simulation regions; hence, our results do not account for discrepancies between profile and actual program inputs.

Using our aggressive prototype compiler described in Section 5.1, we process each benchmark in Table I to extract its pre-execution code. All profile runs and pre-execution code extraction steps are performed automatically by our compiler, without any manual intervention. Then, we run our benchmarks and their associated pre-execution threads on the SMT simulator from [Madon et al. 1999], which is derived from SimpleScalar’s out-of-order processor model [Burger and Austin 1997]. Our simulator uses the same functional unit, register renaming, branch predictor, and cache models provided by SimpleScalar. In addition, it has been augmented to model SMT’s multiple hardware contexts. The program counter, register map, and branch predictor tables have been replicated; all other structures are shared between contexts. Also, the issue logic selects instructions from one or more threads per cycle, using the ICOUNT fetch policy from [Tullsen et al. 1996]. Finally, the ISA support described in Section 5.2 has been provided. Table II reports the simulator

Table III. Characterization of pre-execution code generated by the aggressive compiler.

Benchmark	Load	Lines	Slice	Store	Pref	SE	DA	DX	Back	Span
256.bzip2	28	62	0	6	9	2	1	0	-	0
175.vpr	32	318	130	14	21	1	2	0	-	1
300.twolf	55	132	23	35	25	1	0	5	0.24	0
254.gap	8	14	8	1	4	1	0	0	-	0
197.parser	5	51	0	11	0	1	0	1	8.58	2
181.mcf	26	69	0	30	10	0	0	1	41.8	1
164.gzip	3	23	0	1	0	0	0	1	57.7	0
183.equake	67	21	0	6	24	0	1	0	-	0
188.ammp	41	67	19	24	35	3	0	0	-	0
179.art	13	40	11	7	12	5	2	0	-	0
177.mesa	1	11	0	2	1	1	0	0	-	0
mst	4	34	16	1	0	0	0	1	0.00	1
em3d	13	12	2	0	7	0	1	0	-	0
TOTAL:	296	854	209	138	148	15	7	9	-	5

settings used for our experiments.

It is important to note our compiler does not perform traditional software prefetching of simple array references, nor does our SMT simulator model hardware stride prefetching. For some benchmarks, our compiler may provide less benefit if traditional stride-based prefetching techniques are applied in concert with pre-execution.

6.2 Basic Evaluation

We present the basic performance evaluation of our aggressive compiler in three parts. First, we characterize the pre-execution code and threads generated by the compiler in Section 6.2.1. Then, we present the actual performance results in Section 6.2.2. Finally, in Section 6.2.3, we analyze the correctness of the generated pre-execution code.

6.2.1 *Characterization.* Before presenting the performance achieved by our aggressive compiler, we first characterize *what* the compiler did. For each benchmark, we report 10 measurements from the pre-execution code generated by the compiler, and 4 measurements from the pre-execution threads spawned at runtime by the pre-execution code. These measurements appear in Tables III and IV.

The 10 code measurements we report in Table III are: the number of problematic loads identified by summary cache-miss profiles ("Load"), the number of code lines in the pre-execution regions prior to program slicing ("Lines"), the number of code lines removed by Unravel ("Slice"), the number of code lines removed as a consequence of store removal ("Store"), the number of loads converted into prefetches ("Pref"), the number of pre-execution regions broken down into different thread initiation schemes ("SE" for Serial, "DA" for DoAll, and "DX" for DoAcross), the percent cache misses incurred in the backbone (as opposed to ribs) for DoAcross pre-execution regions ("Back"), and the number of pre-execution regions that span multiple procedures ("Span"). These measurements show 4 important characteristics of our pre-execution code. First, as indicated by the "Load" column, each benchmark contains a relatively small number of problematic loads. Since our compiler ignores all pre-execution regions contributing fewer than 3% of the total cache misses (see Section 5.1), most cache-missing loads are not considered for pre-execution, allowing our compiler to focus on the most important loads. Second, our compiler removes a significant number of lines of code within pre-execution regions–

Table IV. Characterization of pre-execution threads generated by the aggressive compiler.

Benchmark	Forks	Inst-Fork	Inst-Pre	Inst-Miss
256.bzip2	252003	488	238.6	20.7
175.vpr	275816	1103	228.7	15.0
300.twolf	591343	460	45.6	6.6
254.gap	32739	3908	940.9	110.5
197.parser	4093	27.6K	511.5	13.6
181.mcf	3997152	68.6K	25.7	4.1
164.gzip	1478363	513	53.9	30.3
183.equake	3	21.9M	4.123M	3.2
188.ammp	15951	8110	3.2K	9.5
179.art	612	243K	128.9K	8.3
177.mesa	23750	2218	33.6	26.4
mst	393471	47.6K	53.0	4.9
em3d	300	1.08M	324.2K	2.6
AVG:		16.03K	1038	10.9

40% across all benchmarks on average. Interestingly, a significant fraction of the removed code is due to store removal. Comparing the "Slice" and "Store" columns, we see that Unravel is responsible for 60% of the removed code while store removal accounts for the other 40%. Third, comparing the "Pref" and "Load" columns shows our compiler converts 50% of problematic loads into prefetches. This enables our compiler to employ the Serial scheme aggressively, which requires all problematic loads in the pre-execution region to be converted into prefetches. Out of 31 total pre-execution regions, our compiler selects the Serial scheme in 15 cases, using DoAll and DoAcross for the remaining 16 pre-execution regions. And finally, the "Span" column shows 5 pre-execution regions span multiple procedures, indicating that inter-procedure analysis is important for a few of our benchmarks.

The 4 thread measurements we report in Table IV are: the number of pre-execution threads forked ("Forks"), the number of instructions executed in the main thread between forks ("Inst-Fork"), the number of instructions executed by each pre-execution thread ("Inst-Pre"), and the number of pre-execution thread instructions executed between cache misses ("Inst-Miss"). These measurements show 2 important characteristics of our pre-execution threads. First, pre-execution threads are coarse-grained. As indicated by columns "Inst-Fork" and "Inst-Pre," the frequency of forks is low and the number of instructions executed by each pre-execution thread is high (usually in the thousands or higher). Even though we use a recycled threading model (see Section 5.3), thread spawning is still expensive in our system since thread dispatch and communication of arguments occurs in software. A coarse thread granularity minimizes the impact of these software startup costs. In comparison, previous pre-execution techniques that initiate pre-execution threads in hardware [Annavaram et al. 2001; Collins et al. 2001; Collins et al. 2001; Liao et al. 2002; Moshovos et al. 2001; Roth and Sohi 2001; 2002; Sundaramoorthy et al. 2000; Zilles and Sohi 2001] can tolerate finer-grained threads. Finally, pre-execution threads are efficient at triggering cache misses. Column "Inst-Miss" shows a cache miss is triggered every 10.9 pre-execution thread instructions on average. This measurement suggests Unravel is successful at streamlining pre-execution code.

6.2.2 *Performance Results.* Figure 10 presents the performance achieved by our aggressive compiler. We report execution time without pre-execution, labeled "Baseline," and with pre-execution, labeled "Pre-exec," broken down into three

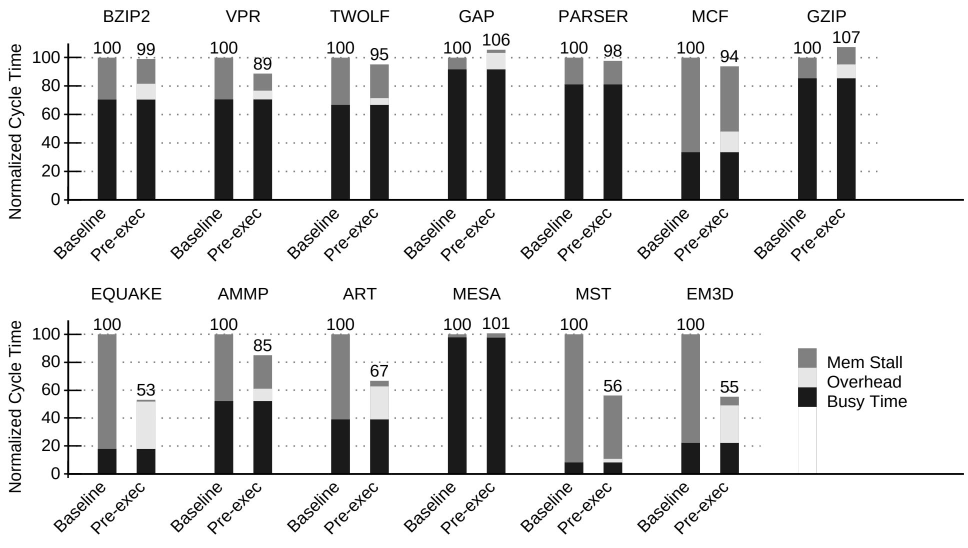


Fig. 10. Normalized execution time broken down into busy, overhead, and memory stall components. The “Baseline” and “Pre-exec” bars show performance for no pre-execution and with pre-execution, respectively.

components. “Busy” is the execution time without pre-execution assuming a perfect data memory system (*e.g.*, all D-cache accesses complete in 1 cycle whereas I-cache accesses work normally). “Overhead” is the incremental increase in execution time over “Busy” due to pre-execution, again on a perfect data memory system. “Mem Stall” is the incremental increase in execution time over “Busy” + “Overhead” assuming a real memory system. All bars are normalized against the “Baseline” bars.

Our aggressive compiler reduces execution time for 10 out of 13 applications. Execution time is reduced between 1% and 47%, providing a 20.9% reduction on average for all 10 applications. The remaining 3 applications experience a degradation in performance by 4.7% on average. Overall, our technique delivers a harmonic average speedup of 17.6% across all 13 applications. Notice with the exception of MESA, the last 6 applications (which are the SPEC CFP2000 and Olden benchmarks) achieve a larger speedup than the first 7 applications (which are the SPEC CINT2000 benchmarks). This is mainly due to the fact that the CFP2000 and Olden benchmarks exhibit greater memory stall than the CINT2000 benchmarks, thus providing more opportunity for performance improvement.

To provide more insight, Figure 11 reports cache-miss coverage. The “Baseline” bars in Figure 11 break down the L1 cache misses without pre-execution into misses incurred outside of pre-execution regions, labeled “Non-Region,” misses satisfied from the L2 cache, labeled “L2-Hit,” and misses to main memory, labeled “Mem.” The “Pre-exec” bars show the same three components, but in addition show those cache misses that are fully or partially covered by pre-execution, labeled “Full” and “Partial,” respectively.

Figure 11 shows our aggressive compiler effectively covers cache misses for VPR,

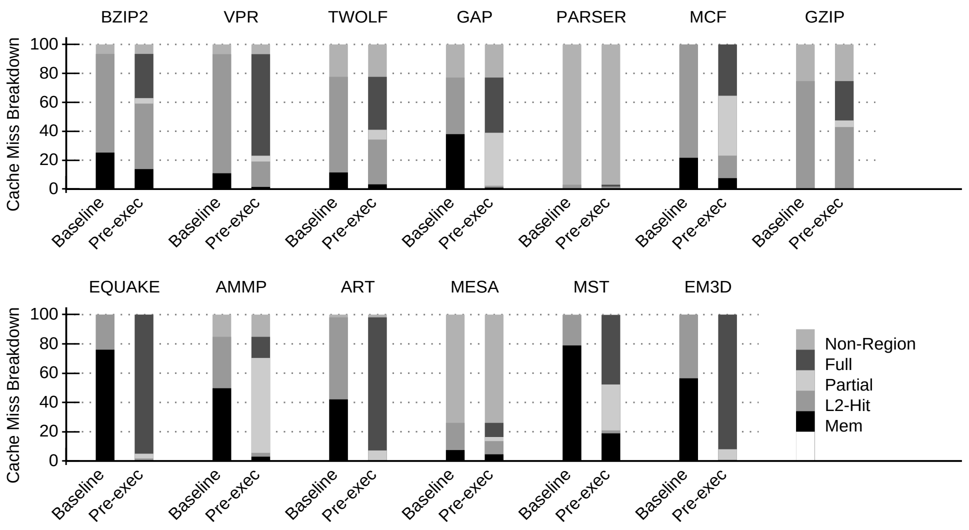


Fig. 11. Cache-miss coverage broken down into uncovered misses occurring outside of pre-execution regions, and misses fully or partially covered by pre-execution, labeled "Non Region," "Full," and "Partial," respectively. Remaining misses either hit in the L2, or are satisfied from main memory, labeled "L2-Hit" and "Mem," respectively.

GAP, MCF, EQUAKE, AMMP, ART, MST, and EM3D, converting 84.9% of the main thread's misses on average across the 13 benchmarks into fully or partially covered misses. For BZIP2, TWOLF, and GZIP, coverage is lower, 36.5%, and for PARSER and MESA, coverage is only 6.9%. Upon closer examination, we found three factors that contribute to reduced cache-miss coverage. First, as described in Section 5.1, our compiler ignores pre-execution regions contributing fewer than 3% of the total cache misses. This is responsible for the "Non-Region" components in Figure 11, which are particularly severe in PARSER and MESA. Second, some memory references are pre-executed late, issuing after the main thread has already suffered the cache miss. Pre-execution threads require a few loop iterations to get ahead of the main thread, so loops with small iteration counts are vulnerable to such late pre-execution. This factor accounts for some uncovered misses in BZIP2, VPR, and TWOLF. Finally, the third factor contributing to reduced cache-miss coverage is inaccurate pre-execution code generated by our compiler. This factor accounts for some uncovered misses in BZIP2, VPR, MCF, and GZIP, and is the focus of the next section.

6.2.3 *Pre-Execution Code Accuracy.* As discussed in Section 2.1.2, our compiler performs cloning within pre-execution regions to ensure pre-execution code faithfully mimics the main thread's memory reference stream. Unfortunately, two of the transformations we perform on pre-execution code, store removal and speculative loop parallelization, can potentially compromise the accuracy afforded by cloning. Store removal may remove code along the data or control flow path leading to problematic load instructions, as discussed in Section 3.2. Speculative loop parallelization may parallelize serial loops. As described in Section 4.1, we do not

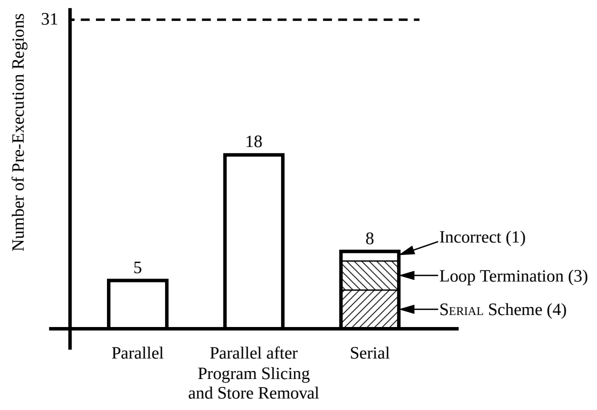


Fig. 12. Breakdown of pre-execution regions into 3 major categories: originally parallel, parallel after program slicing and store removal, and serial. The serial category is further broken down into regions using the SERIAL scheme, regions speculatively parallelized affecting control flow only, and regions that are incorrectly parallelized.

analyze dependences exactly to enable more aggressive parallelization. In particular, we analyze dependences for loop induction variables only; hence, we may mistakenly select a loop for parallelization that contains loop-carried dependences through other variables. Whenever dependences are violated as a consequence of either store removal or speculative loop parallelization, pre-execution threads can potentially generate incorrect memory addresses, leading to reduced cache-miss coverage.

To provide a deeper understanding, we carefully studied those pre-execution regions where store removal or speculative loop parallelization compromise correctness. Out of the 31 pre-execution regions selected by our aggressive compiler, we found 3 cases where store removal causes a problem. One of these cases occurs in VPR, and was discussed in detail in Section 3.2, and the other two cases occur in MCF and BZIP2. For the remaining 28 pre-execution regions, store removal does not affect the correctness of pre-execution code. Why is store removal harmless most of the time? We find control and data flow leading to problematic load instructions frequently involve loop induction variables. (For example, the memory addresses for references "1" and "4" in Figure 3 are computed via complex control and data flow rooted at the induction variables from loops "7" and "8"). Hence, preserving the integrity of induction variables is a necessary (though not sufficient) condition for high cache-miss coverage. Fortunately, the update of an induction variable rarely depends on a global or heap variable modified within the pre-execution region. Since store removal affects global and heap variables only, it rarely destroys data values that can reach induction variables.

Compared to store removal, speculative loop parallelization has an even smaller impact on the correctness of pre-execution code, causing a problem in only 1 pre-execution region from GZIP. Figure 12 sheds more light on this finding. The three bars in Figure 12 report the number of pre-execution regions that can be correctly parallelized using either the DOALL or DOACROSS schemes presented in Section 4.1.

Interestingly, Figure 12 shows only 5 out of the 31 pre-execution regions are parallelizable in their original form; however, after program slicing and store removal, an additional 18 pre-execution regions become parallelizable because code containing the loop-carried dependences are removed. In other words, even though most of the loops in our pre-execution regions contain loop-carried dependences, the "traversal kernel" required to trigger cache-missing loads contains significant parallelism. Of the 8 pre-execution regions that remain serial after program slicing and store removal, 4 employ the SERIAL thread initiation scheme, and hence are not parallelized, because our aggressive compiler is able to convert all their blocking loads into prefetches. And finally, 3 of the serial pre-execution regions, when parallelized, execute for an incorrect number of iterations due to compromised code controlling `continue` and `break` statements, but still execute the problematic load instructions properly.

6.3 Contributions of Algorithms

This section studies the *relative importance* of individual algorithms employed by our aggressive compiler to its overall performance gain. More specifically, Section 6.3.1 compares the relative importance of the algorithms for pre-execution effectiveness–program slicing, prefetch conversion, and speculative loop parallelization. Then, Section 6.3.2 studies the importance of selecting the best thread initiation schemes.

6.3.1 *Comparing Algorithms for Pre-Execution Effectiveness.* Figure 13 quantifies the relative importance of program slicing, prefetch conversion, and speculative loop parallelization in our aggressive compiler for the 10 applications from Figure 10 that our compiler speeds up. We apply each optimization incrementally to all the applications, and measure the change in performance provided by each optimization. In Figure 13, the "Parallel" bars report execution time when we apply speculative loop parallelization only, without program slicing and prefetch conversion. Next, in the "Slicing" bars, we report execution time when applying speculative loop parallelization together with program slicing, but still without prefetch conversion. Finally, the "Pre-exec" bars report execution time with all optimizations applied. Each bar is normalized against the execution time of the corresponding application without pre-execution, reported in the "Baseline" bars. Note, store removal is applied in all experiments except for "Baseline;" otherwise, pre-execution threads would crash the main thread. Also, the SERIAL scheme is never used in the "Parallel" and "Slicing" bars since pre-execution regions always contain blocking loads for these experiments (DOALL and DOACROSS schemes are selected instead).

The results in Figure 13 show speculative loop parallelization provides a performance gain for all the applications except for EM3D, demonstrating that it is an important optimization. In fact, for BZIP2, PARSER, and MCF, the entire speedup provided by our compiler is achieved when going from "Baseline" to "Parallel;" no additional performance gain is achieved by program slicing or prefetch conversion. Surprisingly, program slicing provides a measurable gain for only 1 application, VPR. In all other cases, going from "Parallel" to "Slicing" does not provide any measurable performance boost. This is due to redundancy between

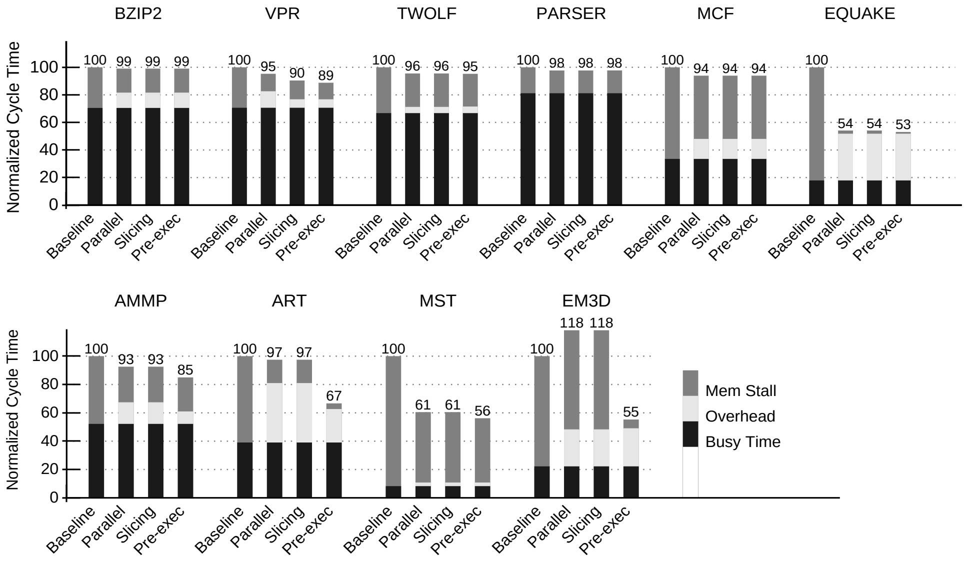


Fig. 13. Impact of individual algorithms on overall performance in the aggressive compiler. The “Parallel” bars show pre-execution with speculative loop parallelization only; the “Slicing” bars show pre-execution with program slicing and speculative loop parallelization, but without prefetch conversion; and the “Pre-exec” bars show pre-execution with all optimizations.

program slicing and back-end code optimizations performed during C compilation, as discussed in Section 2.2.1, and will be explored further in Section 7.

Finally, performance gain due to prefetch conversion, as measured by the difference between the “Slicing” and “Pre-exec” bars, occurs in 7 applications, with particularly large gains visible for AMMP, ART, MST, and EM3D. Prefetch conversion reduces the number of cache-missing loads that stall, allowing pre-execution threads to move more rapidly through the reorder buffer and trigger cache misses early. Even though our reorder buffer is large enough to accommodate multiple cache-missing loads simultaneously, it is *not* large enough to fully tolerate long-latency memory stalls. Hence, increasing the speed of pre-execution threads via prefetch conversion provides a performance boost. Note, since prefetch conversion is an important optimization, program slicing becomes important as well. Although Figure 13 suggests program slicing does not increase overall performance, it still makes an important contribution indirectly: program slicing enables prefetch conversion in our aggressive compiler which does provide significant performance gains.

6.3.2 *Comparing Pre-Execution Initiation Schemes.* This section evaluates the impact of selecting pre-execution initiation schemes on performance in our aggressive compiler. In Figure 14, we study four pre-execution regions from AMMP, EQUAKE, EM3D, and MST, each with different blocking load attributes and induction variable types. In AMMP, our aggressive compiler removes all blocking loads, while in EQUAKE, EM3D, and MST, blocking loads remain after program slicing. Furthermore, AMMP, EQUAKE, and EM3D have affine induction variables, while MST has a pointer-chasing induction variable. For each pre-execution

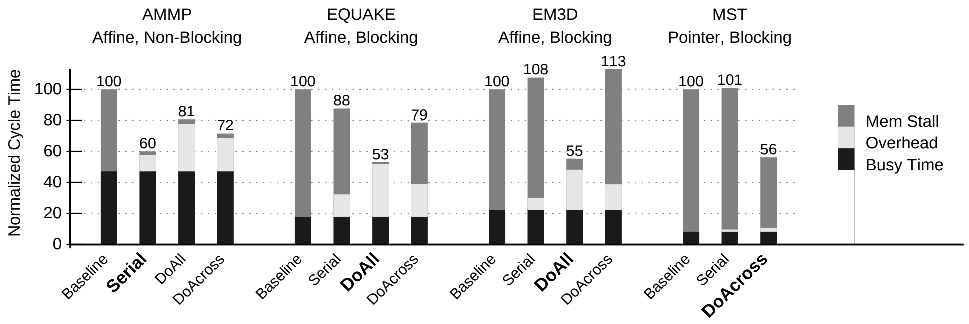


Fig. 14. Comparing pre-execution thread initiation schemes. "Baseline" bars report execution time for no pre-execution. The remaining bars report the execution time when the Serial, DoAll, and DoAcross schemes are applied, respectively. The schemes selected by our compiler appear in boldface.

region, we apply the Serial, DoAll, and DoAcross schemes, as labeled along the X-axis, and compare their performance. The schemes selected by our compiler appear in boldface.

Figure 14 shows three results. First, choosing the right thread initiation scheme has a first-order impact on pre-execution performance. In Figure 14, the best scheme for each pre-execution region outperforms the worst scheme by 25.9%, 39.8%, 51.3%, and 44.6% for AMMP, EQUAKE, EM3D, and MST, respectively.

Second, Figure 14 confirms the best scheme depends on pre-execution region type, as discussed in Sections 4.1 and 4.2. For AMMP, Serial is the best. Since there are no blocking loads, Serial has the same ability to get ahead of the main thread as the other schemes, but has lower overhead because it initiates fewer threads. For EQUAKE and EM3D, DoAll is the best. Serial is ineffective because the blocking loads prevent a lone pre-execution thread from getting ahead of the main thread. Furthermore, DoAll outperforms DoAcross since DoAll has no inter-thread communication once threads are initiated. In DoAcross, communication (and synchronization) must occur each loop iteration to pass the induction variable, limiting the speed of pre-execution threads. Finally, for MST, DoAcross is the best. Since the induction variable is pointer chasing, the DoAll scheme cannot be applied. And Serial is ineffective due to the blocking loads.

The third and final result is that our compiler picks the best scheme for all cases in Figure 14, thus validating the accuracy of the selection algorithm presented in Section 4.2.

6.4 Architectural Support for Synchronization

Thus far, we have evaluated our aggressive compiler assuming only conventional SMT hardware. This section studies the impact of special hardware support for pre-execution threads. In particular, we provide counting semaphores in hardware by implementing special registers for storing semaphore values (in our simulations, we assume 32 semaphore registers). Our counting semaphores are similar to the hardware locking mechanism proposed in [Tullsen et al. 1999]. Special `Sem_P` and `Sem_V` instructions are added to allow "P" and "V" operations on the semaphore

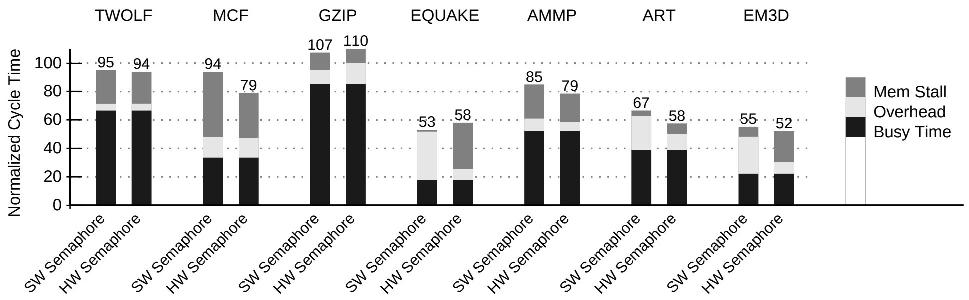


Fig. 15. Impact of architectural support for synchronization. "SW Semaphore" and "HW Semaphore" bars report execution time with software and hardware counting semaphores, respectively. All bars are normalized to the "Baseline" bars from Figure 10.

registers. A `Sem_P` instruction decrements a specified semaphore register if its value is greater than zero; otherwise, it blocks the thread until the value becomes non-zero. Similar to the `suspend` instruction described in Section 5.2, a blocking `Sem_P` causes a pipeline flush of all instructions belonging to the blocked context. A `Sem_V` instruction increments a specified semaphore register and resumes a blocked thread if one or more threads are waiting on the register value (waiting threads are resumed in FIFO order). To reduce the latency of semaphore operations, we allow both `Sem_P` and `Sem_V` to perform speculatively in the execute stage. This approach is similar to the speculative restart mechanism proposed in [Tullsen et al. 1999].

As described in Section 5.3, our compiler uses counting semaphores to synchronize pre-execution threads with the main thread, and in the DoAcross scheme, also to synchronize between pre-execution threads. Our baseline implementation of counting semaphores, described in Section 5.3, is inefficient because semaphore operations are implemented purely in software, and perhaps more importantly, use busy-waiting. Since semaphore operations occur frequently, our hardware counting semaphores have the potential to increase performance. Figure 15 compares pre-execution with software and hardware counting semaphores, labeled "SW Semaphore" and "HW Semaphore" respectively, for 7 applications (for the remaining 6 applications, there is no difference). All bars are normalized to the "Baseline" bars from Figure 10. Figure 15 shows hardware support improves performance by 3.7%. Across all 13 applications, we find hardware semaphores increase the overall speedup of pre-execution from 17.6% to 20.5%. These performance gains arise because hardware semaphores enable pre-execution threads to run farther ahead of the main thread. But in two cases, this degrades performance. In GZIP, pre-execution threads execute instructions along the wrong path due to incorrect loop parallelization, a situation that is aggravated by faster pre-execution threads. Also, in EQUAKE, pre-execution threads fetch data cache blocks faster with hardware semaphores, and incur more L1 cache misses due to cache thrashing.

7. REDUCED COMPILERS

Thus far, we have presented and evaluated our most aggressive compiler for generating pre-execution code. We now study several reduced algorithms designed to simplify compiler implementation with (hopefully) minimal impact on performance. Section 7.1 describes our reduced algorithms. Then, Section 7.2 presents several reduced prototype compilers that implement these algorithms. Finally, Section 7.3 evaluates the reduced compilers, and compares them against our aggressive compiler.

7.1 Reduced Compiler Algorithms

We propose to simplify compiler implementation by eliminating two major steps in the compilation process of our aggressive compiler, program slicing and profiling, and replacing them with simpler algorithms. We begin by discussing alternatives to program slicing in Section 7.1.1 and their impact on prefetch conversion in Section 7.1.2. Then, we discuss replacing cache-miss profiles and loop-trip count profiles with static analysis techniques in Sections 7.1.3 and 7.1.4, respectively.

7.1.1 *Eliminating Program Slicing.* Our program slicing analysis, described in Section 3.2, essentially consists of 3 steps: store removal, slicing, and code pinning. Of these, the slicing step can be redundant in some cases. Because store removal eliminates all side effects inside pre-execution regions, the code removed by program slicing is dead code and can potentially be removed during C compilation. Furthermore, while program slicing identifies exactly the critical computations leading up to problematic load instructions to ensure they are not removed, code pinning achieves a similar goal by introducing side effects for problematic loads and their associated code, thus pinning them down and preventing their removal. Due to this redundancy, we propose a simpler approach in which store removal and code pinning are performed alone, omitting the program slicing step. This approach permits us to eliminate Unravel and its integration with other compiler passes, thereby simplifying compiler implementation.

As an example, Figure 16 shows the pre-execution code generated by our simplified algorithm for the `get_non_updateable_bb` function from Figure 3. The code in Figure 16 is identical to the original code from Figure 3 except for the removal of side effects (underlined code labeled "5" in Figure 3) and the insertion of an `asm` macro (label "1" in Figure 16). In particular, the conditional tests in the loop body which are unnecessary for computing the problematic load remain in the pre-execution code since program slicing has been omitted. However, these extraneous statements do not have side effects due to store removal. Hence, even though the code in Figure 16 appears to be less efficient than its sliced counterpart from Figure 4a, we have observed that the unnecessary code is removed by `gcc`. As long as dead code elimination performed during C compilation removes the unnecessary code, the machine code generated from Figure 16 will be as efficient as the machine code generated from Figure 4a.

7.1.2 *Prefetch Conversion without Program Slicing.* Our baseline prefetch converter from Section 3.3 is coupled with program slicing, so eliminating Unravel also requires new algorithms to drive prefetch conversion. Without program slicing, prefetch conversion becomes harder for the following reason. The simplified algorithm presented in Section 7.1.1 does not remove non-critical code at the source level, as illustrated in Figure 16. Thus, given the pre-execution source code, the conversion of problematic load instructions into prefetches can potentially be limited whenever non-critical code consumes data accessed by the problematic loads. To

```
void get_non_updateable_bb(int inet,
                           struct s_bb *bbptr) {
  int k, xmax, ymax, xmin, ymin, x, y;

  x = block[net[inet].pins[0]].x;
  y = block[net[inet].pins[0]].y;
  xmin = x;  ymin = y;  xmax = x;  ymax = y;

  for (k=1;k<net[inet].num_pins;k++) {
    x = block[net[inet].pins[k]].x;
    asm(" " : : "r" (block[net[inet].pins[k]].x));
    y = block[net[inet].pins[k]].y;
    if (x < xmin) {
       xmin = x;
    } else if (x > xmax) {                       1
       xmax = x;
    }
    if (y < ymin) {
       ymin = y;
    } else if (y > ymax ) {
       ymax = y;
    }
  }
}
```

Fig. 16. Code generated by our compiler for the get_non_updateable_bb function from Figure 3 without program slicing with asm macro added for code pinning (label "1").

```
Given: Global loop nest graph, $G_L$
Compute:  Pre-Execution Region Set, P

 1: P = Φ;
 2: for each loop L in $G_L$
          from inner-most to outer-most {
 3:   if (level(L) == INNER_MOST)
 4:     continue;
 5:   else
 6:     if (P ∩ nested_loops(L) == Φ)
 7:        P = P ∪ {L};
 8: }

 9: for each inner-most loop L in $G_L$ {
10:   if (P ∩ outer_loops(L) == Φ)
11:     P = P ∪ {L};
12: }
```

Fig. 17. Algorithm for computing the set of pre-execution regions, P, without loop-trip count profiles. Φ denotes the empty set.

ensure all candidates for prefetch conversion are accurately identified in the absence of program slicing, it is necessary to disregard the dependences that problematic load instructions may have with non-critical code in anticipation of its removal during C compilation.

We compute the set of active variables necessary for pre-executing the problematic load instructions contained inside a pre-execution region, and convert any problematic load that does not assign one of these active variables into a prefetch. Since non-critical code that will be removed during C compilation do not contain such active variables, their presence in the pre-execution source code generated by our simplified algorithms will not limit opportunities for prefetch conversion. Equations 3 and 4 below compute the active variables set, $V_{<m,v>}$, for a memory variable, v, corresponding to a problematic load instruction at statement, m:

$$V_{<m,v>} = \begin{cases} V_{<n,v>} & \text{if } v \notin defs(n) \\ Vdef_{<n,v>} & \text{otherwise} \end{cases} \tag{3}$$

$$Vdef_{<n,v>} = \left(\bigcup_{x \in refs(n)} \{x\} \right) \bigcup \left(\bigcup_{x \in refs(n)} V_{<n,x>} \right) \bigcup \left(\bigcup_{y \in refs(k)} \bigcup_{k \in control(n)} V_{<k,y>} \right) \tag{4}$$

These equations are almost identical to Equations 1 and 2 from Section 2.2.1. In particular, the first two terms in Equation 4 account for dataflow within the critical code for computing variable v similar to the first two terms in Equation 2, while the last term in Equation 4 accounts for control flow similar to the last term in Equation 2. This similarity is not accidental; rather, it reflects the fact that prefetch conversion is fundamentally enabled by program slicing. Even if we do not use program slicers and rely on C compilers to remove unnecessary code, we must still perform slicing-like analysis to identify prefetch conversion candidates. Note, however, our standalone prefetch converter only performs basic data and control flow analysis similar to what was described in Section 2.2.1, and is thus far less complex than Unravel which performs several sophisticated analyses (see Section 3.1).

Equations 3 and 4 compute the active variables set for a single memory reference, v. We compute $V_{<m,v>}$ for all problematic memory references in the pre-execution region and take their union to form the set of all active variables. Using this merged active variables set, we identify candidates for prefetch conversion as mentioned above. If the data from a problematic load instruction does not assign a variable belonging to the merged active variables set, then the load instruction does not contribute to the critical data or control flow within the pre-execution region; hence, we convert the load instruction into a non-blocking prefetch. Otherwise, we do not apply prefetch conversion.

7.1.3 *Eliminating Cache-Miss Profiles.*

Our default approach for identifying problematic load instructions is to gather cache-miss profiles, thus directly measuring the cache-miss rate of the application's static loads. Cache-miss profiles provide an extremely accurate picture of memory behavior for a particular run of an application. On the other hand, cache-miss profiles are cumbersome to gather, requiring one or more profiling runs before compiler optimizations can occur. Hence, eliminating the need to gather profiles would simplify compiler implementation. Another drawback of cache-miss profiles is that they may not accurately reflect program behavior across a wide range of program inputs.

We propose replacing cache-miss profiles with static analysis techniques to identify problematic loads. Unfortunately, predicting memory behavior exactly using compile-time techniques is nearly impossible. Our approach uses heuristics to identify those static loads which are likely to cache-miss frequently. In particular, we have found the following heuristic to be extremely useful: load instructions whose load addresses are computed as a function of loop induction variables have a high likelihood to exhibit poor memory performance. To eliminate cache-miss profiles, our compiler performs loop-level analysis to identify all loads satisfying this heuristic (a fairly straight-forward task), and assumes all such loads are problematic loads. This simple approach successfully identifies 88% of the cache misses that occur in 10 of our most memory-intensive benchmarks.[6]

While our compile-time heuristic identifies most of the problematic loads correctly, it has a tendency to identify too many loads as problematic. This causes pre-execution to occur where it isn't needed, increasing runtime overhead. To

[6]We obtained this result via manual inspection of our benchmarks. Unfortunately, for 3 benchmarks, the majority of cache misses occur either in libraries for which we do not have source code or in too many loops, making it infeasible to analyze by hand. However, we expect our heuristic is effective for these 3 benchmarks as well.

minimize performance degradation due to overhead, we modify the pre-execution region selection algorithm presented in Figure 6a to omit the second pass through the inner-most loops (lines 11-13). This pass includes inner-most loops with fewer than 25 iterations in the set of pre-execution regions. Although it is difficult to pre-execute such short loops effectively, it is worthwhile trying as long as they contain problematic loads. Without cache-miss profiles, however, it is likely that such loops contain falsely identified problematic loads. Hence, we refrain from pre-executing any short loops in the absence of cache-miss profiles.

7.1.4 *Eliminating Loop Profiles.* In addition to cache-miss profiles, our algorithm for computing pre-execution regions, presented in Section 4.2.1, requires loop-trip count profiles as well to estimate the amount of work in inner-most loops. Since this approach also requires separate profiling runs, it shares the same drawbacks discussed in Section 7.1.3 for cache-miss profiles.

Figure 17 presents a reduced algorithm for computing the set of pre-execution regions, P, that avoids loop-trip count profiles. This algorithm is identical to the original algorithm from Figure 6a except it naively assumes all inner-most loops contain fewer than 25 iterations (this simple assumption is correct for 83% of the inner-most loops in our benchmarks). Hence, on the first pass of the algorithm, none of the inner-most loops are selected as pre-execution regions (lines 3 and 4); instead, the algorithm picks next-outer loops as pre-execution regions as long as nested pre-execution regions are not created (lines 5–7). Also, notice this algorithm performs a second pass through the inner-most loops to pick up any remaining loops that are not pre-executed (lines 9–11) as is done in Figure 6a, a step that was eliminated in the algorithm from Section 7.1.3 to reduce overhead. Because the algorithm in Figure 17 assumes all inner-most loops iterate fewer than 25 times, several important inner-most loops may not get pre-executed after the first pass of the algorithm. Although the second pass can add loops containing falsely identified problematic loads, we find it is worthwhile since it ensures all important inner-most loops are pre-executed.

7.2 Reduced Prototype Compilers

We created variations on our aggressive compiler by substituting the alternatives to program slicing and profiling presented in Section 7.1, thus forming several reduced compilers. Table V lists these compilers, lettered "A" through "E," along with the algorithms that distinguish them. The first (Compiler A) is the aggressive compiler which we have already studied–we include this for the sake of comparison. The remaining entries in Table V represent the reduced compilers. First, we eliminate program slicing and rely on the C compiler's dead code elimination pass to remove unnecessary code (Compiler B), though this compiler still uses Unravel to identify prefetch conversion candidates. Next, we use the standalone prefetch conversion algorithm described in Section 7.1.2 to extract prefetch conversion information (Compiler C), thus eliminating the need for Unravel completely. Then, we substitute summary cache-miss profiles with static analysis to identify problematic loads (Compiler D). Finally, we use the compile-time heuristics described in Section 7.1.4 to select pre-execution regions (Compiler E), thus removing loop-trip count profiles and eliminating the need for profiling completely.

Table V. Compilers used to generate pre-execution code for the experiments. The compilers differ in how they remove unnecessary code, select prefetch conversion candidates, identify problematic load instructions, and estimate inner-most loop work.

	Code Removal	Prefetch Conversion	Problematic Load Identification	Loop Work
A	Program Slicing	Slicing-Based	Profile	Profile
B	Dead Code Elimination	Slicing-Based	Profile	Profile
C	Dead Code Elimination	Standalone	Profile	Profile
D	Dead Code Elimination	Standalone	Static Analysis	Profile
E	Dead Code Elimination	Standalone	Static Analysis	Static Analysis

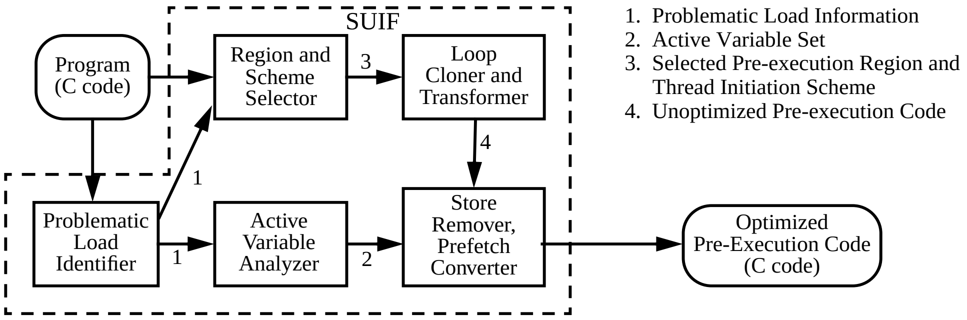

SUIF
Program (C code)
Region and Scheme Selector
Loop Cloner and Transformer
Problematic Load Identifier
Active Variable Analyzer
Store Remover, Prefetch Converter
Optimized Pre-Execution Code (C code)
1
1
2
3
4
1. Problematic Load Information
2. Active Variable Set
3. Selected Pre-execution Region and Thread Initiation Scheme
4. Unoptimized Pre-execution Code

Fig. 18. Major components in our least aggressive prototype compiler. This compiler uses dead code elimination to remove unnecessary code and static analysis to identify problematic loads and select pre-execution regions. Arrows denote interactions between compiler modules.

Figure 18 illustrates the major modules in our simplest reduced prototype compiler, Compiler E. This block diagram is similar to the corresponding block diagram from Figure 9 for the aggressive compiler, except many of the aggressive algorithms have been replaced by the reduced algorithms described above. In particular, this compiler does not incorporate Unravel nor the SimpleScalar simulators. Instead, it relies on C compilation to remove unnecessary code, and it implements the prefetch conversion, problematic load identification, and pre-execution region selection algorithms in SUIF. Note, while Figure 18 shows the implementation of Compiler E only, we also implemented the remaining reduced compilers from Table V by substituting the reduced algorithms incrementally.

7.3 Reduced Compilers Evaluation

In this section, we evaluate our reduced compilers. First, Sections 7.3.1 and 7.3.2 evaluate eliminating program slicing. Then, Sections 7.3.3 and 7.3.4 evaluate eliminating profiling. Results for all these evaluations appear in Figure 19, which shows the performance of each of the 5 compilers from Table V across the 13 benchmarks. Note, all the pre-execution code in these experiments use the baseline software counting semaphores, *not* the hardware semaphores presented in Section 6.4.

7.3.1 *Impact of C Compiler Code Removal.* Comparing Compiler A and Compiler B in Figure 19, we see eliminating Unravel and relying on back-end code optimizations during C compilation to remove unnecessary code impacts performance minimally, increasing the execution time of only 1 benchmark, VPR. The

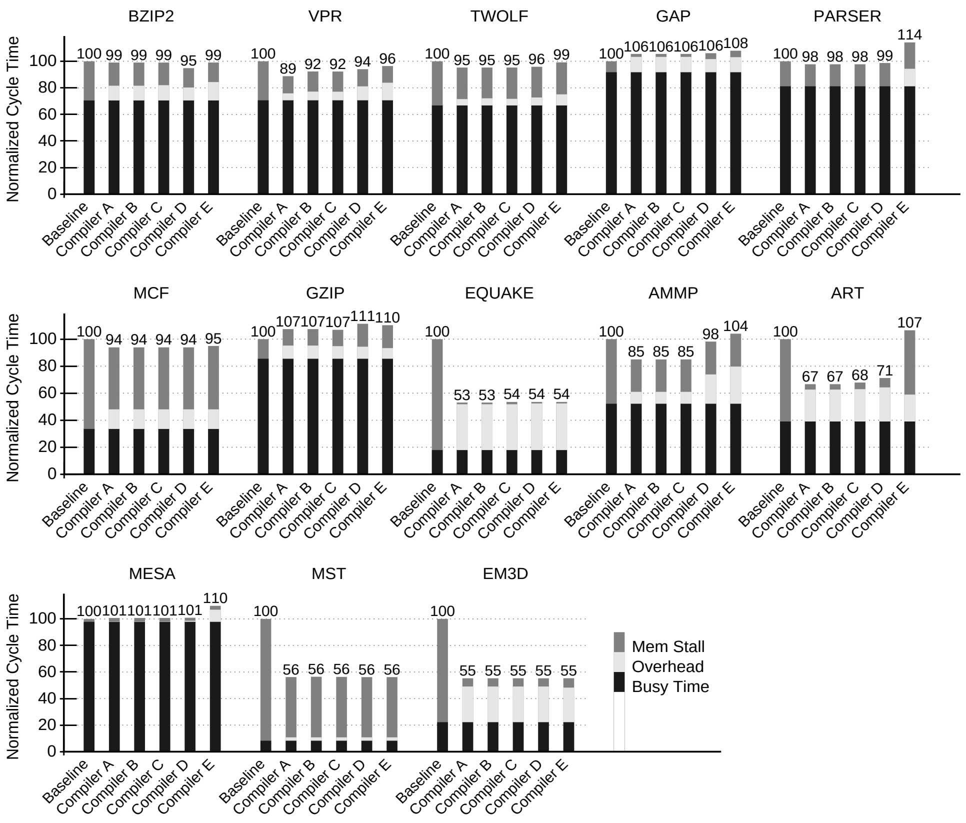


Fig. 19. Normalized execution time broken down into busy, overhead, and memory stall components. Groups of bars labeled "Compiler A" to "Compiler E" report performance for the 5 compilers from Table V.

remaining 12 benchmarks experience no performance degradation when using Compiler B despite the fact this compiler does not perform program slicing. On average, Compiler B achieves a speedup of 17.3%, almost matching the speedup of Compiler A (17.6%). This result suggests Unravel is redundant, removing code that would have been removed by the C compiler anyways. (The same reasoning explains the lack of performance gain when going from the "Parallel" to "Slicing" bars in Figure 13–most of the code removed by Unravel in the "Slicing" bars is removed during C compilation in the "Parallel" bars).

To provide more insight, we compare the number of dynamic instructions executed by each benchmark under Compilers A and B. Figure 20 reports normalized dynamic instruction count in the main thread and pre-execution threads using pre-execution code generated by Compiler A and Compiler B. Each bar has been broken down into compute and overhead components, with the latter accounting for parameter passing, thread initiation, and synchronization instructions. Only instructions executed within pre-execution regions are counted, and busy-wait instructions executed by pre-execution threads are omitted. Figure 20 shows Com-

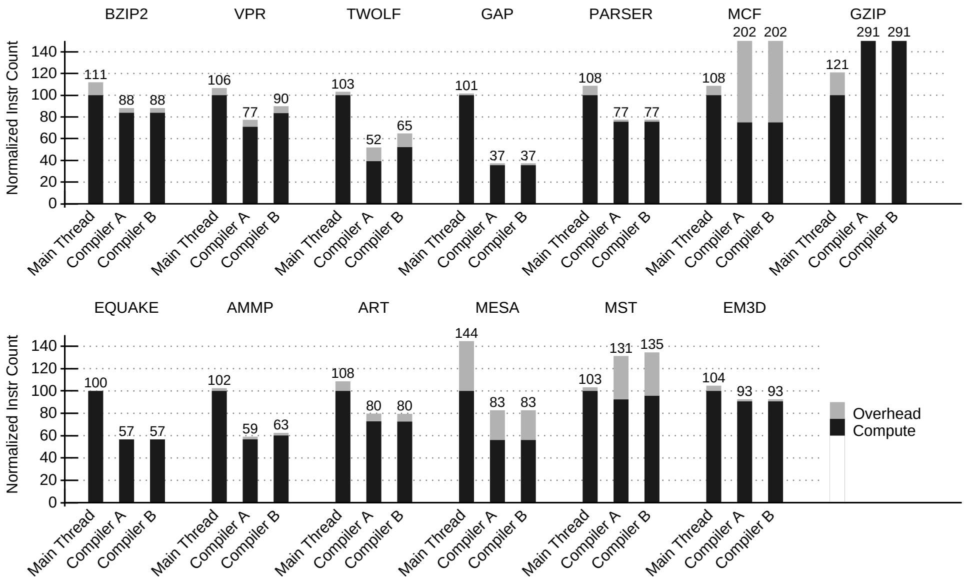


Fig. 20. Number of dynamic instructions removed by Compilers A and B. The "Main Thread" bars report instruction counts for the main thread. The "Compiler A" and "Compiler B" bars report instruction counts for pre-execution threads using pre-execution code generated by Compiler A and B, respectively. Each bar is broken down into compute and overhead components.

pilers A and B both effectively reduce computation performed in pre-execution threads. Comparing the compute components in the "Compiler A" and "Compiler B" bars against the compute components in the "Main Thread" bars, we see pre-execution threads under both compilers execute fewer instructions than the main thread across 12 applications. In one case (GZIP), pre-execution threads perform more computation than the main thread. This is due to incorrect pre-execution code generated by speculative loop parallelization (see Section 6.2.3). If we include overhead instructions, we see pre-execution threads under both compilers execute fewer instructions than the main thread across 10 applications. In MCF and MST, pre-execution threads execute more instructions than the main thread due to high overhead associated with passing arguments and forking threads.

Comparing the "Compiler A" bars against the "Compiler B" bars in Figure 20, we see pre-execution threads execute an equal number of instructions in 9 benchmarks, indicating Unravel is redundant in these cases. For these benchmarks, the code removed by program slicing in Compiler A is dead code because store removal eliminates their side effects. Although Compiler B does not remove this code at the source level, the C compiler removes it as a consequence of dead code elimination. For VPR, TWOLF, AMMP, and MST, however, Unravel is not completely redundant, and succeeds in further reducing the dynamic instruction count on top of the C compiler. After closer examination, we found two factors that permit Unravel to further streamline pre-execution code in these benchmarks. First, the dead code elimination pass in our C compiler, `gcc`, is not perfect. In some cases, it fails to remove dead code which Unravel successfully removes. Second, Unravel propagates dependence information across procedure boundaries. Code sliced inside one procedure can enable slicing in its caller procedures by removing dependences communicated through arguments. Since gcc only performs local analysis, it conservatively assumes code associated with all procedure calls is live. Hence, Compiler B misses opportunities to remove code anytime slices span multiple procedures.

From Figure 20, we conclude there exists a *potential* for program slicing to improve performance even when the C compiler performs dead code elimination. However, it still remains to be seen whether this potential can be realized. For our benchmarks, the additional reduction in dynamic instruction count provided by Unravel is modest, and as the "Compiler A" and "Compiler B" bars from Figure 19 indicate, there is a performance improvement in only 1 out of 4 cases where Unravel is not redundant.

7.3.2 *Impact of SUIF Prefetch Converter.* Compiler B does not perform program slicing, but it still uses Unravel to identify candidates for prefetch conversion. If we eliminate Unravel completely, we must offload the prefetch conversion analysis from Unravel to SUIF, as is done in Compiler C. Comparing the "Compiler B" and "Compiler C" bars in Figure 19, we see our SUIF-based algorithm for identifying prefetch conversion candidates presented in Section 7.1.2 performs essentially as well as Unravel's prefetch converter, achieving a speedup of 17.1% (versus 17.3% for Compiler B). This data suggests our simple algorithm (*e.g.*, Equations 3 and 4 from Section 7.1.2) is sufficient to identify the same prefetch conversion candidates as Unravel in almost all cases, and the sophisticated analyses that Unravel performs does not improve prefetch conversion appreciably.

7.3.3 *Impact of Eliminating Cache-Miss Profiles.* Comparing the "Compiler C" and "Compiler D" bars in Figure 19, we see eliminating cache-miss profiles has a negative impact on performance, reducing the overall speedup across our benchmarks from 17.1% to 15.0%. Two factors are responsible for this performance degradation. First, without cache-miss profiles, a significant number of problematic loads are falsely identified, causing our pre-execution region selection algorithm to pick significantly more loops for pre-execution. We found Compiler D selects 62 pre-execution regions across our 13 benchmarks, or double the number of pre-execution regions selected by Compilers A, B, or C which use cache-miss profiles. This occurs despite the modification to throttle pre-execution region selection described in Section 7.1.3. Unfortunately, many of these additional pre-execution regions contain falsely identified problematic loads that infrequently miss in the cache, increasing runtime overhead without improving cache-miss coverage.

Second, not only does the number of pre-execution regions grow with the number of falsely identified problematic loads, but the number of problematic loads within each pre-execution region also increases. Even though these additional loads infrequently miss in the cache, our compiler assumes they are problematic and tries to convert them into prefetches. However, due to the increased number of problematic loads, it becomes less likely the compiler will successfully convert all of them into prefetches. Consequently, several pre-execution regions that are pre-executed using the SERIAL scheme under Compilers A, B, and C are instead pre-executed using the

DoAll or DoAcross schemes due to incomplete prefetch conversion, increasing runtime overhead without improving pre-execution effectiveness. This phenomenon is responsible for much of the performance degradation in AMMP and ART.

While an increased number of falsely identified problematic loads generally leads to performance degradation, in some cases, performance can actually improve. For example, Figure 19 shows BZIP2 experiences a 4% performance boost under Compiler D compared to Compilers A, B, and C. BZIP2 contains several loads that suffer a small number of cache misses. Because these loads do not meet our 3% threshold, they are not identified as problematic loads by our profiler. However, Compiler D picks them for pre-execution. Unlike the majority of cases in Figure 19, for BZIP2, the increased coverage benefit achieved by pre-executing loads with small cache-miss counts outweighs the overhead incurred, resulting in performance gain.

7.3.4 *Impact of Eliminating Loop Profiles.* Comparing the "Compiler D" and "Compiler E" bars in Figure 19, we see eliminating loop-trip count profiles degrades performance in most benchmarks, reducing the overall speedup from 15.0% to 7.7%. The impact is most pronounced in ART where execution time increases by 36%, resulting in a 7% degradation compared to no pre-execution. In ART, there is an inner-most loop containing several problematic loads that executes 1000s of iterations. Since our alternate algorithm assumes all inner-most loops iterate fewer than 25 times (see Section 7.1.4), it pre-executes this loop from its next-outer loop. Unfortunately, this initiates pre-execution threads much earlier than necessary, triggering cache misses far too early and causing thrashing in the data cache. For all other benchmarks, inner-most loops execute a modest number of iterations, so pre-executing them from the next-outer loop works well. Hence, our simple heuristic for estimating loop work is correct for the majority of cases, but can cause pathologic behavior in some instances.

In BZIP2, VPR, TWOLF, PARSER, AMMP, and MESA, Compiler E degrades performance compared to Compiler D due to increased overhead. As described in Section 7.1.4, Compiler E performs a second pass to pick up any inner-most loops that are not pre-executed after the first pass. While this ensures all important inner-most loops get pre-executed, it selects several pre-execution regions containing falsely identified problematic loads. We found Compiler E selects 115 pre-execution regions across our 13 benchmarks, roughly another doubling of selected pre-execution regions compared to Compiler D. These additional pre-execution regions increase runtime overhead without improving cache-miss coverage.

8. MULTIPROGRAMMING RESULTS

Sections 6 and 7 study compiler-based pre-execution assuming single programs only. In this section, we evaluate compiler-based pre-execution in the context of multiprogramming. When performing pre-execution for multiprogrammed workloads, the main computation threads and the pre-execution threads can effectively share the resources of the SMT processor, thus increasing overall processor throughput. On the other hand, it is also possible that supporting pre-execution for multiple programs simultaneously places a large enough pressure on processor resources that throughput degrades. The goal of this section is to understand the tradeoffs that arise when combining compiler-based pre-execution with simultaneous execution in

Table VI. Benchmarks for multiprogramming experiments. TDF values for each benchmark are listed in the 3rd row, and pre-execution initiation schemes used by each benchmark are listed in the last row. Benchmark groupings are indicated in the first row: Group 1 benchmarks have high TDF values, Group 2 benchmarks have medium TDF values, and Group 3 benchmarks have low TDF values.

Group 1			Group 2	Group 3		
em3d	mst	181.mcf	179.art	175.vpr	256.bzip2	188.ammp
99.98%	99.64%	86.18%	63.14%	23.12%	22.22%	18.14%
DA	DX	DX	DA & SE	DA & SE	DA & SE	SE

SMT processors.

8.1 Multiprogramming Simulation Methodology

In our study, we generate pre-execution code using our most aggressive compiler (Compiler A). Furthermore, we model an SMT processor with 5 hardware contexts– 2 to support main computation threads, and only 3 to support pre-execution threads. (Except for the number of hardware contexts, the same machine configuration from Table II is used for all our multiprogramming experiments). Rather than provide enough contexts to support the maximum number of pre-execution threads (8 total contexts would be necessary for 2 main threads), we assume the contexts used for pre-execution are dynamically shared between the 2 main threads. We believe this is a realistic model since it reduces the total number of hardware contexts needed, and it leads to better utilization of the contexts reserved for pre-execution threads.

Because there are a limited number of contexts available for pre-execution, the main threads must arbitrate for hardware contexts before initiating pre-execution. We implement arbitration in software, using a software queue to keep track of the idle hardware contexts available for pre-execution threads. Before entering a pre-execution region, each main thread must acquire a lock to gain exclusive access to the idle context queue, and reserve the necessary number of hardware contexts depending on the thread initiation scheme. The Serial scheme attempts to reserve 1 context, while the DoAll and DoAcross schemes attempt to reserve 3 contexts. Due to contention, reservation requests may fail to acquire the requisite number of contexts. We modified the pre-execution code generated by our compiler to enable each thread initiation scheme to accommodate a variable number of pre-execution threads. To improve throughput, we use a hardware semaphore for the idle context queue lock; pre-execution code uses software semaphores.

The degree of contention for hardware contexts, an important determiner of multiprogramming performance, depends on how actively individual programs employ pre-execution threads. We introduce a metric, called the Threading Duty Factor (TDF), to quantify pre-execution thread activity. We define an application's TDF as the percentage of time pre-execution threads remain active when the application runs on a dedicated 4-context SMT machine, computed as follows:

$$TDF(\%) = \frac{\sum_{i=1}^{3} Execution\ time\ of\ pre-execution\ thread\ i}{Total\ simulation\ time} * \frac{1}{3} * 100 \qquad (5)$$

In Table VI, we report the TDF values and thread initiation schemes for 7 bench-

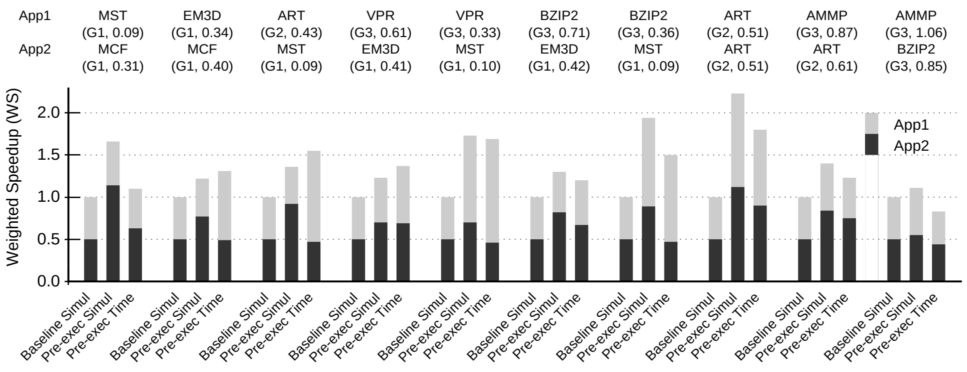


Fig. 21. Weighted speedup of multiprogrammed workloads. "Baseline Simul" denotes simultaneous execution without pre-execution. "Pre-exec Simul" and "Pre-exec Time" denote pre-execution with simultaneous execution and time sharing, respectively. The TDF group that each application belongs to, and the baseline IPC for each application are shown in parentheses.

marks which we use in our experiments. We divide these 7 benchmarks into three groups according to their TDF values: Group 1 benchmarks have high TDF values (EM3D, MST, and MCF), Group 2 benchmarks have medium TDF values (ART), and Group 3 benchmarks have low TDF values (VPR, BZIP2, and AMMP).

8.2 Multiprogramming Evaluation

Figure 21 reports our multiprogramming results. We study 10 multiprogrammed workloads, each consisting of two applications selected from Table VI. For each workload, we perform three experiments. All experiments run for a *fixed number of cycles*, corresponding to a scheduling interval. The first two experiments, labeled "Baseline Simul" and "Pre-exec Simul," run the two applications simultaneously without and with pre-execution, respectively. In these experiments, both applications are active during the entire scheduling interval. For the experiment labeled "Pre-exec Time," we run the two applications with pre-execution in a time-sliced manner, so that each application is active for exactly half the scheduling interval. Figure 21 shows the weighted speedup [Snavely and Tullsen 2000] achieved across the entire scheduling interval for each experiment and workload, computed as follows:

$$Weighted\ Speedup = \frac{1}{2}\left[\frac{IPC_{App1}}{IPC_{BaseApp1}} + \frac{IPC_{App2}}{IPC_{BaseApp2}}\right] \quad (6)$$

where $IPC_{BaseApp1}$ and $IPC_{BaseApp2}$ are the IPCs achieved by each application in the "Baseline Simul" experiment. Individual bars are broken down into two components to show each application's contribution to overall weighted speedup. For all experiments, we use a scheduling interval of 100M cycles.

First, comparing the "Pre-exec Simul" bars against the "Baseline Simul" bars, we see pre-execution improves weighted speedup for all 10 multiprogrammed workloads in Figure 21. On average, the workloads receive a 45.0% boost in weighted speedup when using pre-execution compared to no pre-execution. This demonstrates pre-execution can be gainfully employed for multiple simultaneously executing applications, even when applications must share a limited number of hardware contexts for pre-execution threads.

Although pre-execution seems to be better than no pre-execution, we would also like to study whether combining pre-execution with simultaneous execution is the best choice. Simultaneous execution in an SMT processor tends to boost throughput because multiple applications more effectively exploit processor pipeline resources. However, it also tends to reduce pre-execution effectiveness because simultaneously executing applications must compete for hardware contexts. We quantify this tradeoff by comparing the "Pre-exec Simul" and "Pre-exec Time" bars in Figure 21. Workloads in the "Pre-exec Simul" experiments exploit pipeline sharing because applications execute simultaneously, while workloads in the "Pre-exec Time" experiments exploit contentionless access to hardware contexts because applications execute one at a time.

Figure 21 shows the tradeoff between pre-execution and simultaneous execution depends on the workload. When both applications have either medium or low TDF values, as in the ART-ART, AMMP-ART, and AMMP-BZIP2 workloads, then "Pre-exec Simul" always performs better than "Pre-exec Time." For these workloads, contention for hardware contexts is low in the "Pre-exec Simul" experiments due to the modest TDF values of individual applications. Consequently, applications can exploit pipeline sharing when run simultaneously without paying a performance penalty for context contention, resulting in higher throughput compared to time-slicing. In contrast, when one of the applications in the workload has a high TDF value, pre-execution and simultaneous execution do not always combine symbiotically. High-TDF applications tend to monopolize hardware contexts, limiting the other application's ability to perform pre-execution. Time-slicing can relieve this contention, but at the expense of sacrificing simultaneous execution. If boosting the pre-execution performance of the lower-TDF application outweighs the benefit of pipeline sharing, then "Pre-exec Time" outperforms "Pre-exec Simul." This happens in the EM3D-MCF, ART-MST, and VPR-EM3D workloads. If, however, the benefit of pipeline sharing outweighs boosting the pre-execution performance of the lower-TDF application, then "Pre-exec Simul" outperforms "Pre-exec Time." This happens in the MST-MCF, VPR-MST, BZIP2-EM3D, and BZIP2-MST workloads.

Across all 10 multiprogrammed workloads, "Pre-exec Simul" outperforms "Pre-exec Time" by 9.9% on average. So, for our workload mixes, simultaneous execution is generally more profitable than time-slicing. We also note "Pre-exec Time" outperforms "Baseline Simul" in all workloads except one, providing a boost in weighted speedup of 29.7% on average. This reinforces our earlier claim that pre-execution is better than no pre-execution for the workloads in Figure 21, even when pre-execution applications do not exploit simultaneous execution.

9. RELATED WORK

Previous work has explored automating pre-execution using either compiler [Kim and Yeung 2002], linker [Liao et al. 2002; Roth and Sohi 2002], or hardware [Annavaram et al. 2001; Collins et al. 2001; Sundaramoorthy et al. 2000] approaches. This article is closest to our own earlier paper on compile-time extraction of pre-execution code [Kim and Yeung 2002]. To the best of our knowledge, the work in [Kim and Yeung 2002] is the first to automate pre-execution using a compiler; this article extends our earlier work, providing a more comprehensive study of the compiler approach. More specifically, this article proposes several reduced algorithms for extracting pre-execution code that are less aggressive, and hence simpler to implement, than our original compiler algorithms. These reduced algorithms also eliminate the cumbersome profiling step required by our more aggressive algorithms. Furthermore, compared to [Kim and Yeung 2002], this article provides a more extensive experimental evaluation, including a detailed comparison between our aggressive and reduced algorithms, as well as a study of our algorithms in the context of multiprogramming.

Contemporaneously with our compiler work, Liao *et al* [Liao et al. 2002] propose automating pre-execution using a post-pass binary tool to instrument pre-execution code directly into program binaries. More recently, Roth and Sohi [Roth and Sohi 2002] propose a framework for automatically extracting data-driven threads (also known as *p-threads*) from simulator instruction traces. Compared to the compiler approach, the advantage of such binary-level approaches is that they do not require source code, so they are more transparent. On the other hand, the compiler approach generates source-level pre-execution code that can be compiled onto multiple target ISAs, so it is a more portable approach. Binary-level extraction produces platform-dependent pre-execution code, so a different binary analyzer is needed for each target ISA.

The techniques in [Kim and Yeung 2002], [Liao et al. 2002], and [Roth and Sohi 2002] are software techniques; several researchers have also investigated hardware techniques for extracting pre-execution code. Dynamic Speculative Precomputation [Collins et al. 2001] proposes the Retired Instruction Buffer (RIB). Similar to a trace cache fill unit, the RIB stores traces of committed instructions from which pre-execution traces are extracted. Dependence Graph Precomputation [Annavaram et al. 2001] performs similar analyses to extract pre-execution traces, but only considers instructions in the Instruction Fetch Queue (IFQ). Slipstream Processors [Sundaramoorthy et al. 2000] use a speculative compute engine to automatically get ahead of the main processor, not only for pre-execution, but also for fault tolerance purposes as well. The advantage of these hardware approaches is that they are fully transparent to the user. However, they are more complex than software approaches since code extraction analysis is performed in hardware.

Like the techniques for automating pre-execution code extraction described above, several others have studied how to create effective pre-execution code as well [Collins et al. 2001; Luk 2001; Roth and Sohi 2001; Zilles and Sohi 2001]. However, these studies are not concerned with automation, but instead construct the pre-execution code by hand. Of these, our work is closest to Luk's Software-Controlled Pre-Execution [Luk 2001]. Luk is the first to propose instrumenting pre-execution at the source-code level, which is the basis for our work since we use a source-to-source compiler. One major difference between our two approaches is that Luk's pre-execution threads and main thread execute the same code. In contrast, we generate separate code for pre-execution threads via cloning; hence, our pre-execution code optimizations cannot affect main thread correctness, permitting more aggressive optimizations.

In contrast to Luk's source-level approach, several techniques have explored the binary-level approach (again, through manual code extraction). Three such examples are Speculative Precomputation (SP) [Collins et al. 2001], Data-Driven Multithreading (DDMT) [Roth and Sohi 2001], and Execution-Based Prediction (EBP) [Zilles and Sohi 2001]. SP and DDMT extract pre-execution code by analyzing instruction traces acquired on a simulator using a technique called *backward slicing* [Zilles and Sohi 2000], whereas EBP extracts pre-execution code directly from program binaries. Our algorithms are analogous to SP's and DDMT's algorithms. For example, our program slicer performs the source-level equivalent of backward slicing, and our DoAcross parallelization scheme is similar to SP's chaining triggers. Furthermore, EBP's optimizations, like ours, are not strictly correct, and rely on speculation hardware support to preserve integrity of the main computation. However, because the SP, DDMT, and EBP pre-execution models involve instruction-level analysis, they are better suited for binary analyzers (in fact, Liao's approach [Liao et al. 2002] is based on the SP model), whereas our approach targets source-to-source compilers.

While tolerating memory latency has been the focus of most previous work on pre-execution, several researchers have proposed using pre-execution for other purposes as well. Master/Slave Speculative Parallelization [Zilles and Sohi 2002] proposes pre-executing data values as a replacement for hardware value predictors to enable more accurate speculative parallelization. Chang and Gibson [Chang and Gibson 1999] propose using pre-execution to provide hints to the operating system for I/O prefetching. In addition, several researchers have studied pre-executing hard-to-predict branches. Both EBP and DDMT investigate such branch-based pre-execution, as do Chappell *et al* [Chappell et al. 2002] and Farcy *et al* [Farcy et al. 1998]. In a similar spirit, Roth *et al* [Roth et al. 1999] propose pre-executing virtual function call target addresses. Our work considers pre-executing cache misses only; however, it may be possible to adapt our compiler algorithms to these other uses of pre-execution.

In addition, there are several other related techniques and studies. Most recently, Wang *et al* [Wang et al. 2002] compare out-of-order execution with Speculative Precomputation on an Itanium processor, and show these techniques are complementary. Dependence-Based Prefetching [Roth et al. 1998] proposes an early form of pre-execution for pointer-chasing memory references. To our knowledge, however, the earliest group of work which can be credited with providing the enabling mechanisms for pre-execution includes Simultaneous Subordinate Microthreading (SSMT) [Chappell et al. 1999], Assisted Execution (AE) [Dubois and Song 1998], and Runahead processing [Dundas and Mudge 1997]. SSMT and AE are the first to introduce the notion of helper threads, and Runahead processing is the first to demonstrate execution-based data prefetching.

Finally, our work leverages several previous compiler techniques. Significant work exists in the area of program slicing–a good survey of this area appears in [Binkley and Gallagher 1996]. Previous work has investigated slicing in the context of software debugging, testing, parallelization, and maintenance. We apply program slicing to optimize pre-execution code. In addition, conventional parallelizing compilers have traditionally exploited two forms of loop-level parallelism: `doall` and `doacross` [Cytron 1986; Padua et al. 1980]. Using existing analyses [Padua and Wolfe 1986], our compiler also parallelizes these loop types; however, compared to conventional parallelizing compilers, we can apply parallelization more aggressively since the code our compiler generates is executed only speculatively.

10. CONCLUSION

This article presents several algorithms for creating pre-execution thread code automatically in a source-to-source compiler, and prototypes them using Unravel, SimpleScalar, and the SUIF framework. Using a detailed architectural simulator of an SMT processor, the article measures the performance gains achieved by the different prototype compilers, thus quantifying the effectiveness of the compiler approach for pre-execution, as well as providing insight into the performance tradeoffs between different compiler algorithms.

We draw several conclusions based on our experimental results. First, we find that compiler-based pre-execution achieves significant performance gains. Our most aggressive compiler reduces execution time by 20.9% for 10 out of 13 applications, and delivers an average speedup of 17.6% across all 13 applications. To achieve this performance gain, we employ several speculative optimizations (*e.g.*, store removal and speculative loop parallelization). Fortunately, these optimizations do not frequently compromise the correctness of the cache-miss "traversal kernel;" hence, our pre-execution code correctly generates addresses for a large number of cache misses.

Second, most of the algorithms in our aggressive compiler make significant contributions to overall performance, and thus are all important for achieving performance. Speculative loop parallelization by itself improves the performance of 9 out of 10 applications, and is responsible for all of the performance gain in 3 applications. Surprisingly, program slicing improves performance in only 1 application on top of speculative loop parallelization, but enables prefetch conversion which improves performance additionally for 7 applications. Careful selection of the pre-execution thread initiation scheme also impacts performance significantly. Our results show a 25.9-51.3% performance differential between the best and worst schemes. Fortunately, our compiler chooses the best scheme in the cases we examined.

Third, our aggressive compiler can be simplified, in some cases with only minimal impact on performance. For the benchmarks we study, program slicing can be eliminated with virtually no performance degradation. Program slicing often removes dead code that would have been removed during C compilation anyways; hence, it is frequently redundant. However, our results suggest program slicing may provide more significant gains for other benchmarks, but further work is needed. In addition, eliminating cache-miss profiling degrades performance only modestly, reducing overall speedup from 17.1% to 15.0%. Our compile-time heuristic successfully identifies most of the problematic loads, resulting in good coverage. However, our heuristic can be overly conservative, causing some problematic loads to be falsely identified and increasing pre-execution overhead. Unfortunately, eliminating loop-trip count profiles reduces overall speedup from 15.0% to 7.7%. We conclude loop-trip count profiles are necessary for good performance in our compilers.

Finally, compiler-based pre-execution can benefit multiprogramming as well as single applications. Across 10 multiprogrammed workloads, a 42.9% boost in IPC is achieved when simultaneously executing applications use pre-execution compared to no pre-execution. Whether or not the combination of pre-execution and simultaneous execution outperforms pre-execution and time-slicing depends on how heavily individual programs use pre-execution threads. We introduce a new metric, called the Threading Duty Factor (TDF), to quantify pre-execution thread activity. For workloads containing high-TDF applications, our results show time-slicing can outperform simultaneous execution in some cases. However, overall, pre-execution with simultaneous execution outperforms pre-execution with time-slicing for the workloads we study.

In the future, we plan to improve our compiler algorithms for constructing pre-execution threads. First, we plan to more precisely determine the profitability of pre-execution, so that our compilers can apply pre-execution only when its gain outweighs its cost. We believe using execution time profiles in addition to cache-miss profiles is a promising way for enabling compilers to better evaluate the cost and benefits of pre-execution. Another promising direction is to determine the profitability of pre-execution at runtime. In this approach, the compiler would still construct pre-execution thread code statically, but deciding which pre-execution regions to activate would happen dynamically. This approach is potentially superior to a fully static approach since it can exploit phased behavior. Second, we also plan to develop better algorithms for pre-execution effectiveness. In this article, we proposed using multiple pre-execution threads to tolerate long-latency memory instructions that block. An interesting alternative would be to employ miss clustering techniques, such as unroll-and-jam [Pai and Adve 1999], to increase the number of cache-missing load instructions simultaneously resident in the processor's instruction window from a *single* pre-execution thread. This approach could enable one pre-execution thread to achieve performance similar to multiple pre-execution threads, but with a significantly lower demand on hardware contexts. Finally, in addition to improving our compiler algorithms, we would also like to experiment with more applications to better understand the types of programs that can benefit from pre-execution.

REFERENCES

Abraham, S. G., Sugumar, R. A., Rau, B. R., and Gupta, R. 1993. Predictability of Load/Store Instruction Latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*. IEEE/ACM, Austin, TX, 139–152.

Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S.-T. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E. 1997. Continuous Profiling: Where Have All the Cycles Gone? SRC Technical Note 1997-016a, Digital. July.

Annavaram, M., Patel, J. M., and Davidson, E. S. 2001. Data Prefetching by Dependence Graph Precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. ACM, Goteborg, Sweden, 52–61.

Binkley, D. and Gallagher, K. 1996. *A Survey of Program Slicing*. Academic Press.

Burger, D. and Austin, T. M. 1997. The SimpleScalar Tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison. June.

Chang, F. and Gibson, G. A. 1999. Automatic I/O Hint Generation through Speculative Execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. ACM, New Orleans, LO, 1–14.

Chappell, R. S., Kim, S. P., Reinhardt, S. K., and Patt, Y. N. 1999. Simultaneous Subordinate Microthreading (SSMT). In *In Proceedings of the 26th International Symposium on Computer Architecture*. ACM, Atlanta, GA, 186–195.

Chappell, R. S., Tseng, F., Yoaz, A., and Patt, Y. N. 2002. Difficult-Path Branch Prediction Using Subordinate Microthreads. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. ACM, Anchorage, AK, 307–317.

Chen, T.-F. and Baer, J.-L. 1995. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Transactions on Computers 44,* 5 (May), 609–623.

Cmelik, R. F. and Keppel, D. 1993. Shade: A Fast Instruction Set Simulator for Execution Profiling. TR 93-12, Sun Microsystems. July.

Collins, J. D., Tullsen, D. M., Wang, H., and Shen, J. P. 2001. Dynamic Speculative Precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture*. IEEE/ACM, Austin, TX, 306–317.

Collins, J. D., Wang, H., Tullsen, D. M., Hughes, C., Lee, Y.-F., Lavery, D., and Shen, J. P. 2001. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. ACM, Goteborg, Sweden, 14–25.

Cytron, R. 1986. Doacross: Beyond Vectorization for Multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*. IEEE Computer Society Press, University Park, PA, 836–844.

Dubois, M. and Song, Y. H. 1998. Assisted Execution. CENG Technical Report 98-25, Department of EE-Systems, University of Southern California. October.

Dundas, J. and Mudge, T. 1997. Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss. In *Proceedings of the 1997 ACM International Conference on Supercomputing*. ACM, Vienna, Austria, 68–75.

Farcy, A., Temam, O., Espasa, R., and Juan, T. 1998. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proceedings of the 31st International Symposium on Microarchitecture*. IEEE/ACM, Dallas, TX, 59–68.

Ferrante, J., Ottenstein, K., and Warren, J. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages 9,* 3 (July), 319–349.

Kim, D. and Yeung, D. 2002. Design and Evaluation of Compiler Algorithms for Pre-Execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, San Jose, CA, 159–170.

Liao, S. S. W., Wang, P. H., Wang, H., Hoflehner, G., Lavery, D., and Shen, J. P. 2002. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Berlin, Germany, 117–128.

Luk, C.-K. 2001. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. ACM, Goteborg, Sweden, 40–51.

Lyle, J. R. and Wallace, D. R. May 1997. Using the unravel program slicing tool to evaluate high integrity software. In *Proceedings of 10th International Software Quality Week*. San Francisco, CA.

Lyle, J. R., Wallace, D. R., Graham, J. R., Gallagher, K. B., Poole, J. P., and Binkley, D. W. 1995. Unravel: A CASE Tool to Assist Evaluation of High Integrity Software. NISTIR 5691, National Institute of Standards and Technology. August.

Madon, D., Sanchez, E., and Monnier, S. 1999. A Study of a Simultaneous Multithreaded Processor Implementation. In *Proceedings of EuroPar '99*. Springer-Verlag, Toulouse, France, 716–726.

Moshovos, A., Pnevmatikatos, D. N., and Baniasadi, A. 2001. Slice-Processors: An Implementation of Operation-Based Prediction. In *Proceedings of the International Conference on Supercomputing*. ACM, Sorrento, Italy, 321–334.

Mowry, T. 1998. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *Transactions on Computer Systems 16,* 1 (February), 55–92.

Padua, D. A., Kuck, D. J., and Lawrie, D. H. 1980. High-Speed Multiprocessors and Compilation Techniques. *IEEE Transactions on Computers C-29,* 9 (September), 763–776.

Padua, D. A. and Wolfe, M. J. 1986. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM 29,* 12 (December), 1184–1201.

Pai, V. S. and Adve, S. 1999. Code Transformations to Improve Memory Parallelism. In *Proceedings of the International Symposium on Microarchitecture*. IEEE/ACM, Haifa, Israel, 147–155.

Rogers, A., Carlisle, M., Reppy, J., and Hendren, L. 1995. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems 17,* 2 (March).

Roth, A., Moshovos, A., and Sohi, G. S. 1998. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, San Jose, CA, 115–126.

Roth, A., Moshovos, A., and Sohi, G. S. 1999. Improving Virtual Function Call Target Prediction via Dependence-Based Pre-Computation. In *Proceedings of the 13th Annual International Conference on Supercomputing*. ACM, Rhodes, Greece, 356–364.

Roth, A. and Sohi, G. S. 2001. Speculative Data-Driven Multithreading. In *Proceedings of the 7th International Conference on High Performance Computer Architecture*. IEEE, Monterrey, Mexico, 191–202.

Roth, A. and Sohi, G. S. 2002. A Quantitative Framework for Automated Pre-Execution Thread Selection. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*. IEEE/ACM, Istanbul, Turkey, 430–441.

Snavely, A. and Tullsen, D. M. 2000. Symbiotic Jobscheduling for a Simutaneous Multithreading Processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Cambridge, MA, 234–244.

SPEC. 2000. SPEC CPU2000 V1.2 (http://www.specbench.org/osg/cpu2000/).

Sundaramoorthy, K., Purser, Z., and Rotenberg, E. 2000. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Cambridge, MA, 191–202.

Tullsen, D. M., Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., and Stamm, R. L. 1996. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 1996 International Symposium on Computer Architecture*. ACM, Philadelphia, 191–202.

Tullsen, D. M., Lo, J. L., Eggers, S. J., and Levy, H. M. 1999. Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*. IEEE, Orlando, FL, 54–58.

Wang, P. H., Wang, H., Collins, J. D., Grochowski, E., Kling, R. M., and Shen, J. P. 2002. Memory Latency-Tolerance Approaches for Itanium Processors: Out-of-Order Execution vs. Speculative Precomputation. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE, Boston, MA, 187–196.

Weiser, M. 1984. Program Slicing. *IEEE Transactions on Software Engineering SE-10,* 4 (July).

Zilles, C. B. and Sohi, G. 2001. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*. ACM, Goteborg, Sweden, 2–13.

Zilles, C. B. and Sohi, G. 2002. Master/Slave Speculative Parallelization. In *Proceedings of the 35th International Symposium on Microarchitecture*. IEEE/ACM, Istanbul, Turkey, 85–96.

Zilles, C. B. and Sohi, G. S. 2000. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM, Vancouver, Canada, 172–181.

Received November 2002; Accepted July 2003