Appears in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS-X)*, October 2002, San Jose, CA.

Design and Evaluation of Compiler Algorithms for Pre-Execution

Dongkeun Kim and Donald Yeung

Department of Electrical and Computer Engineering
Institute for Advanced Computer Studies
University of Maryland at College Park

{dongkeun, yeung}@eng.umd.edu

ABSTRACT

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 paper investigates a source-to-source C compiler for extracting pre-execution thread code automatically, thus relieving the programmer or hardware from this onerous task. At the heart of our compiler are three algorithms. First, *program slicing* removes non-critical code for computing cache-missing memory references, reducing pre-execution overhead. Second, *prefetch conversion* replaces blocking memory references with non-blocking prefetch instructions to minimize pre-execution thread stalls. Finally, *threading scheme selection* chooses the best scheme for initiating pre-execution threads, speculatively parallelizing loops to generate thread-level parallelism when necessary for latency tolerance. We prototyped our algorithms using the Stanford University Intermediate Format (SUIF) framework and a publicly available program slicer, called *Unravel* [13], and we evaluated our compiler on a detailed architectural simulator of an SMT processor. Our results show compiler-based pre-execution improves the performance of 9 out of 13 applications, reducing execution time by 22.7%. Across all 13 applications, our technique delivers an average speedup of 17.0%. These performance gains are achieved fully automatically on conventional SMT hardware, with only minimal modifications to support pre-execution threads.

1. INTRODUCTION

Processor performance continues to be limited by long-latency memory operations. In the past, researchers have studied prefetching [5, 16] to tolerate memory latency, but these techniques are ineffective for irregular memory access patterns common in many important applications. Recently, a more general latency tolerance technique has been proposed, called *pre-execution* [1, 6, 7, 11, 12, 20, 23]. Pre-execution uses idle execution resources, for example spare hardware contexts in a simultaneous multithreading (SMT) processor [22], 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.

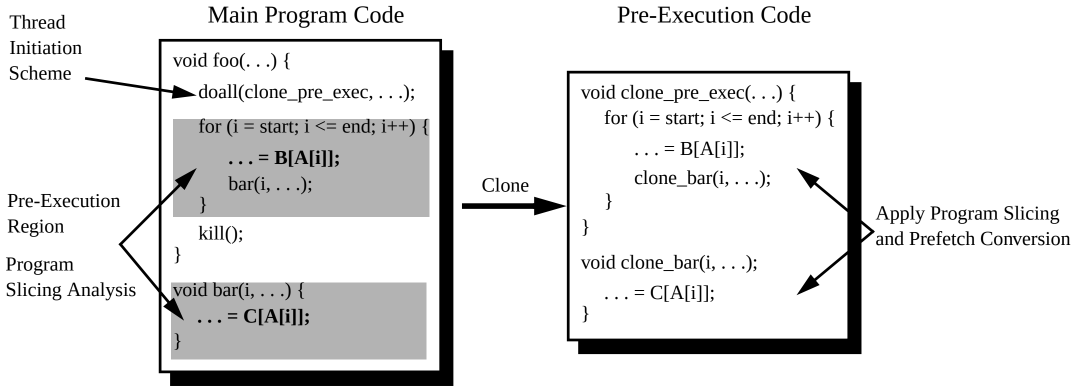
A critical component of pre-execution is the construction of the pre-execution thread code, a task that can be performed either in software or in hardware. *Software-controlled pre-execution* extracts code for pre-execution from source code [12] or compiled binaries [7, 11, 20, 23] using off-line analysis techniques. This approach reduces hardware complexity since the hardware is not involved in thread construction. In addition, off-line analysis can examine large regions of code, and can exploit information about program structure to aid in constructing effective pre-execution threads. In contrast, *hardware-controlled pre-execution* [1, 6] extracts code for pre-execution from dynamic instruction traces using trace-processing hardware. This approach is transparent, requiring no programmer or compiler intervention, and can examine runtime information in an on-line fashion.

Despite significant interest in pre-execution recently, there has been very little work on compiler support in this area. Without a compiler, the applicability of software-controlled pre-execution is limited. Generating pre-execution thread code is labor intensive and prone to human error. Even if programmers are willing to create pre-execution code by hand, doing so would reduce code maintainability since modifications to application code would require rewriting pre-execution code as well. Consequently, manual instrumentation of software-controlled pre-execution is viable only when a large programming effort can be justified. In contrast, compiler support would enable pre-execution for all programs. Pre-execution compilers can also potentially benefit hardware-controlled pre-execution by providing compile-time information to assist hardware thread construction, reducing hardware complexity.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS X 10/02 San Jose, CA, USA
Copyright 2002 ACM 1-58113-574-2/02/0010 ...$5.00.



```
void foo(. . .) {
    doall(clone_pre_exec, . . .);
    for (i = start; i <= end; i++) {
        . . . = B[A[i]];
        bar(i, . . .);
    }
    kill();
}
void bar(i, . . .) {
    . . . = C[A[i]];
}
```

```
void clone_pre_exec(. . .) {
    for (i = start; i <= end; i++) {
        . . . = B[A[i]];
        clone_bar(i, . . .);
    }
}
void clone_bar(i, . . .);
    . . . = C[A[i]];
}
```

Figure 1: Overview of compiler-based pre-execution. Profiling identifies cache-missing memory references (bold faced code) and a potential pre-execution region (shaded code). Program slicing analysis identifies non-critical code and opportunities for prefetch conversion. Loop analysis selects a thread initiation scheme. Finally, pre-execution thread source code is generated by cloning and applying optimizations.

This paper presents the design, implementation, and evaluation of a source-to-source C compiler for pre-execution. To our knowledge, this is the first source-level compiler to automate pre-execution. Guided by profile information, our compiler extracts pre-execution thread code from application code via static analysis. At the heart of our compiler are three algorithms: program slicing, prefetch conversion, and threading scheme selection. These algorithms enhance the ability of the pre-execution threads to get ahead of the main thread, thus triggering cache misses sufficiently early to overlap their latency.

To demonstrate their feasibility, we prototype our algorithms using the Stanford University Intermediate Format (SUIF) framework, and a publicly available program slicer, called Unravel. Using our prototype, we conduct an experimental evaluation of compiler-based pre-execution on a detailed architectural simulator of an SMT processor. Our results show compiler-based pre-execution improves the performance of 9 out of 13 applications, providing a 22.7% reduction in execution time on average. Across all 13 applications, our technique delivers an average speedup of 17.0%. These performance gains are achieved fully automatically on conventional SMT hardware, with only minimal modifications to support pre-execution threads.

The rest of this paper is organized as follows. Section 2 presents an overview of our compiler-based pre-execution technique. Next, Sections 3 and 4 present our algorithms for generating pre-execution code. Then, Section 5 discusses several implementation issues, and Section 6 presents our results. Finally, Section 7 concludes the paper.

2. COMPILER-BASED PRE-EXECUTION

2.1 Overview

Compiler-based pre-execution consists of several steps, as illustrated in Figure 1. First, we gather cache-miss and loop iteration count profiles. Our compiler uses the cache-miss profiles to identify frequent cache-missing memory references, as well as the loops that contain them. We refer to each identified loop as a potential *pre-execution region* which defines the scope for pre-executing the cache misses within the loop. For example, the reference `B[A[i]]` in the main program of Figure 1 is a frequent cache-missing memory reference, and the shaded loop defines a potential pre-execution region for the reference. Notice a pre-execution region spans multiple procedures whenever loops call procedures containing cache misses. In Figure 1, the memory reference `C[A[i]]` in the `bar` procedure also cache misses frequently, and is included in the pre-execution region since `bar` is called from the same loop.

After cache-miss and loop iteration count profiles have been acquired, our compiler performs a series of analyses. First, *program slicing* identifies non-critical code for computing the cache-missing memory references. Program slicing also identifies cache-missing memory references that can be converted into prefetches. Second, our compiler determines the set of pre-execution regions to instrument, and then selects a *pre-execution thread initiation scheme* for each region. We propose three schemes: Serial, DoAll, and DoAcross. The last two schemes use *speculative loop parallelization* to initiate multiple pre-execution threads. Our compiler performs induction variable analysis, and uses program slicing information and loop iteration count profiles to select the best scheme for each pre-execution region.

Finally, pre-execution source code is generated by cloning each pre-execution region, including both the loop and called procedures. Program slicing and prefetch conversion optimizations are applied to the cloned code, and thread initiation code is inserted into the main program according to the selected thread initiation scheme. Figure 1 illustrates these steps.

2.2 Speculative Pre-Execution Model

As in previous pre-execution techniques [6, 7, 11, 12, 20, 23], 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 ensures that 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, and 3. `kill` instructions executed by the main thread (shown in Figure 1) terminate active and possibly incorrect pre-execution threads after the main thread leaves a pre-execution region.

Hardware support for speculation significantly relaxes the correctness assurances required from our compiler. Because pre-execution threads share memory with the main thread, our compiler must guarantee pre-execution code never writes to main thread data structures. We perform *store removal* to eliminate memory side effects. However, aside from removing memory side effects, there are no other correctness considerations for pre-execution code. Although many of the compiler optimizations described in Section 2.1 are not legal or safe under all circumstances, our compiler can apply them aggressively due to the speculation hardware support, permitting our compiler to make performance tradeoffs freely.

2.3 Related Work

Liao *et al* [11] was the first to automate software-based pre-execution. Their approach uses a post-pass compiler tool to instrument pre-execution thread code into program binaries. In contrast, we propose a source-to-source compiler to instrument pre-execution at the source-code level. Because our instrumentation is itself source code, it is portable and can be easily compiled onto multiple targets. Furthermore, our approach permits compiler designers to focus their efforts on a single tool since improved compiler algorithms benefit all platforms automatically. Binary instrumentation, on the other hand, must be performed separately for each binary, sacrificing portability and requiring designers to retarget the analysis tools for each platform. The advantage of binary analyzers, however, is that they can optimize programs even when source code is not available.

Several researchers have developed pre-execution techniques for SMT machines. Our work is closest to Luk's Software-Controlled Pre-Execution [12]. Luk proposes several schemes for instrumenting pre-execution into source code, and applies them by hand on several applications to study their effectiveness. Our approach also involves source-level transformations, but our work focuses on performing transformations using compiler algorithms. Another difference is that Luk's pre-execution threads and main thread execute the same code. We generate separate code for pre-execution threads; hence, our pre-execution code optimizations cannot possibly affect main thread correctness, permitting more aggressive optimizations.

Speculative Precomputation (SP) [7] and Data-Driven Multithreading (DDMT) [20] analyze instruction traces to extract minimal sequences of data-dependent instructions, or *backward slices* [24], to pre-execute cache-missing loads. 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 (see Section 4.1) is similar to SP's chaining triggers. Compared to SP and DDMT, however, our algorithms are more suitable for compiler implementation. Because the SP and DDMT pre-execution models involve instruction-level analysis, they are natural targets for binary analyzers (in fact, Liao's post-pass tool [11] is based on the SP model).

Execution-Based Prediction (EBP) [23] is another binary-level pre-execution technique similar to SP and DDMT, but extracts pre-execution code directly from program binaries rather than instruction traces. EBP's optimizations, like ours, are not strictly correct, and rely on speculation hardware support to preserve integrity of the computation. In addition to pre-executing cache misses, both EBP and DDMT also pre-execute branches, and DDMT allows integration of partial computations into the main thread. We consider pre-executing cache misses only.

In addition, there are several other related techniques. Dynamic Speculative Precomputation [6] and Dependence Graph Precomputation [1] not only perform pre-execution, but extract pre-execution code using trace-processing hardware. Slipstream Processors [21] use a speculative compute engine to automatically get ahead of the main processor for pre-execution and fault tolerance. Dependence-Based Prefetching [19] proposes an early form of pre-execution for pointer-chasing memory references. Simultaneous Subordinate Microthreading [4] and Assisted Execution [9] introduced the notion of helper threads, and Runahead processing [10] was 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 [2]. 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` [8, 17]. Using existing analyses [18], 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.

3. PROGRAM SLICING

Pre-execution threads need only execute the critical computations leading up to cache-missing memory references; all other computations can be removed or "sliced away," allowing pre-execution threads to run more efficiently. Previous pre-execution techniques perform slicing on instruction traces or binaries [7, 11, 20, 23]. In Sections 3.1 and 3.2, we describe *program slicing* [2], a slicing technique that operates on source code, thus allowing easy integration into a source-level compiler. In addition to removing non-critical code, program slicing also enables *prefetch conversion*, described later in Section 3.3, and the result of both program slicing and prefetch conversion drive other optimizations described in Section 4.

3.1 Slicing Basics

The goal of program slicing is to extract a code fragment, or *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.

Several experimental program slicers have been developed. Our system is based on *Unravel* [13], a publicly available program slicer for ANSI C from the National Institute of Standards and Technology (NIST).[1] Unravel, illustrated in Figure 2, consists of two modules: an analyzer and a slicer. The analyzer parses all .c and .h source files in the application and generates a program dependence graph (PDG) across the entire program. Given a slice criterion, the slicer

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

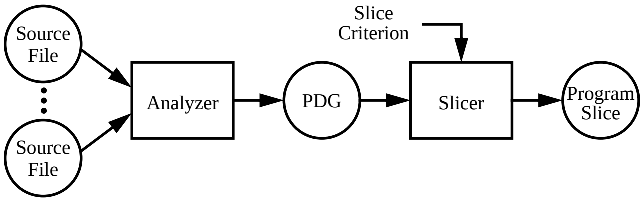


Figure 2: High-level structure of Unravel.

traverses the PDG iteratively, performing data-flow and control-flow analysis to extract the program slice.

The slicer computes program slices in the following manner. First, the program slice is initialized to the statement associated with the slice criterion, and an *active set* is initialized to the variable specified by the slice criterion. Next, all statements in the PDG that are predecessors to the statement(s) in the current slice are added to the current slice. This includes statements that assign values to variables in the active set, as well as statements that control the execution of any statement in the current slice. Then, the assigned variables in the slice criterion are removed from the active set, and the variables referenced by the newly identified predecessor statements are added to the active set. This process repeats until no new statements are added to the current slice, at which time, the current slice is output as the program slice.

Unravel performs several sophisticated analyses to handle many language features found in C. In particular, Unravel performs inter-procedure analysis to construct slices spanning procedure boundaries and source code modules (as long as all modules are provided to the analyzer). Also, Unravel handles data dependences through individual structure members, and performs pointer analysis. Unfortunately, a complete description of these analyses is beyond the scope of this paper. To obtain more details on Unravel, see [14].

3.2 Slices 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 four issues related to our memory-driven program slices: slice criterion specification, store removal, slice termination, and slice merging. This section describes our modifications using the code example in Figure 3 from VPR, a SPECInt CPU2000 benchmark.

Slice Criterion Specification. To pick the memory references that the slicer should analyze, we perform simulation-based cache profiling to record the number of cache misses incurred by each static load instruction in the application, and identify the top cache-missing loads. Using debugging information, we translate each of the load PCs into a source code line number and variable name. In Figure 3, four frequent cache-missing memory references identified through profiling 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.

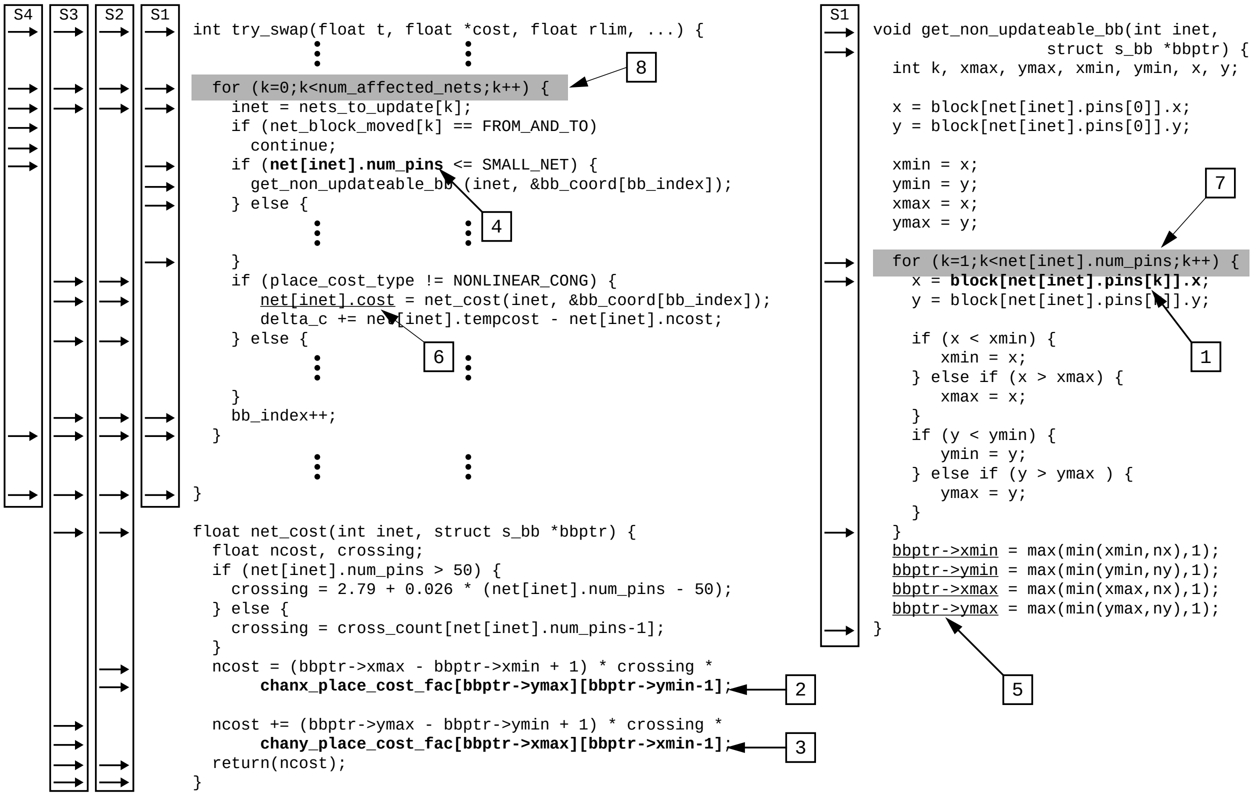
Store Removal. As discussed in Section 2.2, pre-execution threads should never modify memory state visible to the main thread to ensure correct main thread execution. Our SUIF compiler, described in Section 5, removes all stores to statically allocated global variables, and stores to heap variables through 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 the SUIF compiler. Hence, before running the slicer, we delete all DEFs to global and heap variables in the PDG produced by Unravel's analyzer. 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 that dataflow through global or heap variables within a pre-execution region rarely leads to cache-missing memory references (for example, memory references "1" and "4" are unaffected by store removal). 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 criteria identified by cache-miss profiling. 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 within the potential pre-execution region, so we modified the slicer to limit the scope of slicing. A pre-execution region, as discussed in Section 2, is defined by a loop containing the cache-missing memory reference. As we will see later in Section 4.2, we select either the inner-most loop or the next-outer loop encompassing a cache-missing load to serve as its pre-execution region. Hence, we terminate slicing once we have encountered two nested looping statements above the slice criterion (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 crucial because loops are often nested across procedure boundaries, 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 outer loops where slicing terminates.



```
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);
}
```

Figure 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.

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.

To minimize the duplication of slice code and hence the pre-execution overhead, we merge individual slices whose bitmasks intersect so they can be pre-executed together as a single pre-execution region. We simply "OR" together the bitmasks with intersection, forming a single bitmask that represents the merged slice. To maintain a small slice size, we also clear any bits that lie outside of the second lowest loop nest in the merged bitmask. Merging the bitmasks in Figure 3, we see our slicing technique removes 29 out of 58 lines across the three procedures. We find our slices are quite efficient, generating a cache miss every 10.9 instructions on average for the applications we study in Section 6.

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 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.

We perform a *prefetch conversion* optimization using data dependence information computed by program slicing. This optimization considers each cache-missing memory reference in the pre-execution regions after program slicing. If the data accessed by the memory reference is not needed by the slice code (*i.e.*, the dependent program statements have been removed by the program slicer), we convert the blocking memory reference into a non-blocking prefetch. In Figure 3, memory references "1"-"3" can be converted into prefetches.

4. PRE-EXECUTION INITIATION

Program slicing and prefetch conversion, described in Section 3, optimize pre-execution code. Before pre-execution code can be generated, however, we must determine how to initiate the pre-execution threads. This section describes our pre-execution thread initiation schemes, our algorithms for assigning initiation schemes to pre-execution regions, and how our compiler generates code for each initiation scheme assignment.

4.1 Thread Initiation Schemes

Our compiler employs three schemes for initiating pre-execution threads: Serial, DoAll, and DoAcross. Figure 4 illustrates these schemes.

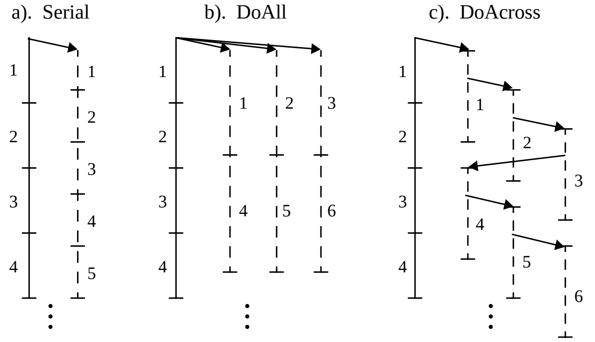


Figure 4: Three pre-execution thread initiation schemes: a) Serial, b) DoAll, and c) DoAcross. Solid lines denote the main thread, dotted lines denote pre-execution threads, arrows denote thread spawning, and numeric labels denote loop iteration count.

Serial. This scheme initiates a single pre-execution thread for each pre-execution region. As shown in Figure 4a, 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. The problem is blocking loads that program slicing and prefetch conversion are unable to remove (*e.g.*, memory reference “4” in Figure 3). If blocking loads appear in the pre-execution code, the pre-execution thread will 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 overlap their long-latency memory operations. Our compiler extracts thread-level parallelism by parallelizing loops. 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 compiler does not need to guarantee correctness thanks to the speculative nature of pre-execution threads, permitting us to parallelize loops *speculatively.* Our compiler performs 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 compiler recognizes 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 compiler encounters an affine loop, it assumes the loop is fully parallel, and generates code to pre-execute the loop iterations independently. As shown in Figure 4b, 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.

```
a). Given: Global loop nest graph, $G_L$
    Compute: Pre-Exec Region Set, P

    P = Φ;
    do {
      if (level(L) == INNER_MOST) {
         if (iteration_count(L) ⩾ 25)
            P = P ∪ {L};
      } else {
         if {P ∩ nested_loops(L) == Φ)
            P = P ∪ {L};
      }
    } ∀ loops L in $G_L$ from inner-most
                            to outer-most;
    do {
      if (P ∩ outer_loops(L) == Φ)
         P = P ∪ {L};
    } ∀ inner-most loops L in $G_L$;
```

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

    SE = DA = DX = F = Φ;
    do {
      if (num_blocking_load(L) == 0)
         SE = SE ∪ {L};
      else {
         if (induction(L) == AFFINE)
            DA = DA ∪ {L};
         else
            DX = DX ∪ {L};
      }
      F = F ∪ called_procedures(L);
    } ∀ loops L in P;
```

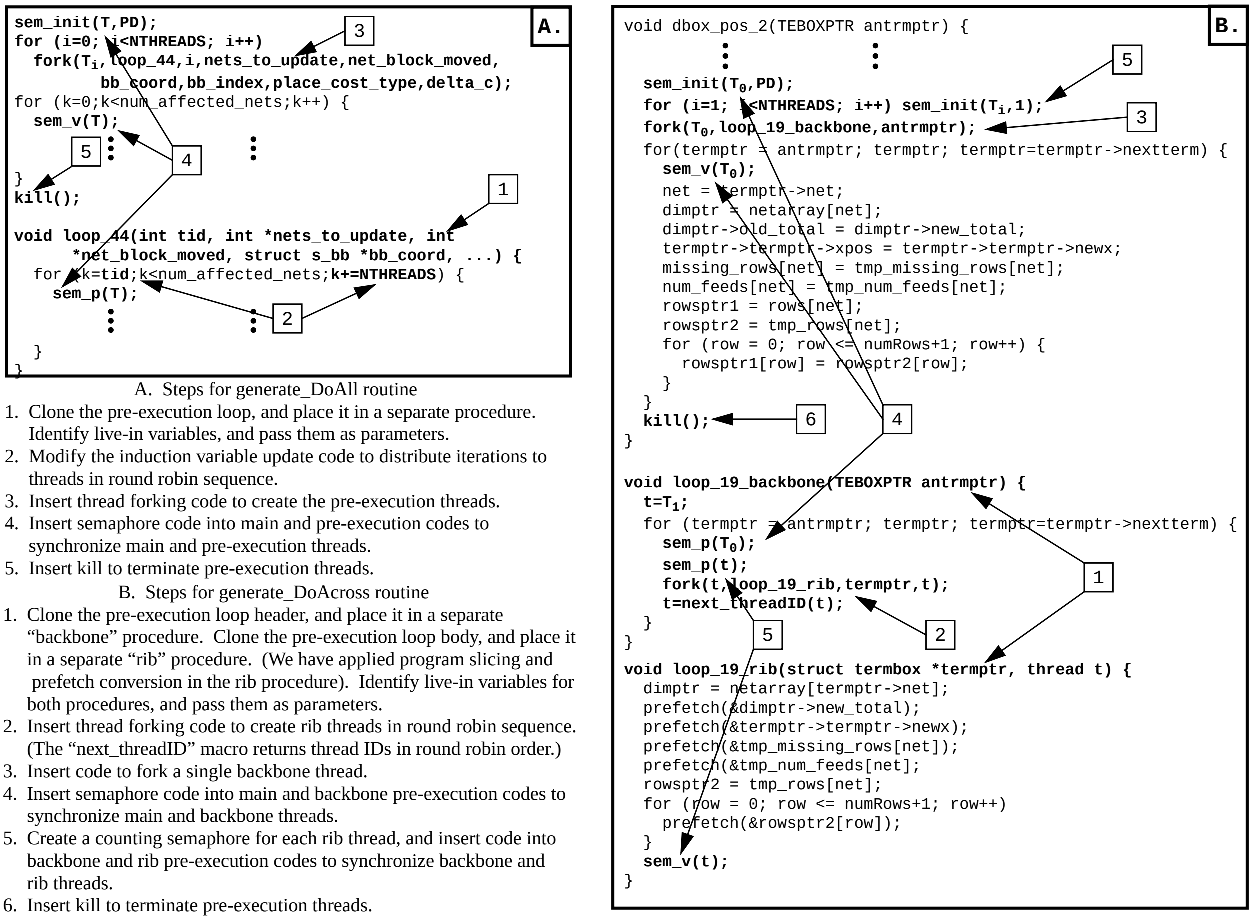
```
c). Given: SE, DA, DX, and F computed in b)

    do { generate_Serial(L); } ∀ loops L in SE;
    do { generate_DoAll(L); } ∀ loops L in DA;
    do { generate_DoAcross(L); } ∀ loops L in DX;
    do { clone_and_optimize(f); } ∀ procedures f in F;
```

Figure 5: Algorithms for instrumenting 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. c) Code generation.

DoAcross. The second parallelization scheme, DoAcross, speculatively parallelizes 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 updating the induction variable early in the loop iteration, and initiating a thread for the next iteration before entering the loop body. As shown in Figure 4c, only the induction variable computations 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 Region and Scheme Selection

Figure 5a-b presents our algorithms to determine the thread initiation schemes for pre-execution. First, we compute the set of pre-execution regions, P, from which we will initiate pre-execution threads. Each pre-execution region is either the inner-most loop or the next-outer loop containing a group of merged program slices. Which loop we select depends on two factors. 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, the loop must contain enough work to amortize pre-execution startup costs. This favors selecting next-outer loops. We use loop iteration count profiles to approximate the work of inner-most loops. If an inner-most loop iterates at least 25 times on average, we select it as a pre-execution region. Otherwise, we try to select the next-outer loop as a pre-execution region.



```
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);
            ⋮              ⋮

  }
}
```

A. Steps for generate_DoAll routine

1. Clone the pre-execution loop, and place it in a separate procedure. Identify live-in variables, and pass them as parameters.
2. Modify the induction variable update code to distribute iterations to threads in round robin sequence.
3. Insert thread forking code to create the pre-execution threads.
4. Insert semaphore code into main and pre-execution codes to synchronize main and pre-execution threads.
5. Insert kill to terminate pre-execution threads.

B. Steps for generate_DoAcross routine

1. Clone the pre-execution loop header, and place it in a separate "backbone" procedure. Clone the pre-execution loop body, and place it in a separate "rib" procedure. (We have applied program slicing and prefetch conversion in the rib procedure). Identify live-in variables for both procedures, and pass them as parameters.
2. Insert thread forking code to create rib threads in round robin sequence. (The "next_threadID" macro returns thread IDs in round robin order.)
3. Insert code to fork a single backbone thread.
4. Insert semaphore code into main and backbone pre-execution codes to synchronize main and backbone threads.
5. Create a counting semaphore for each rib thread, and insert code into backbone and rib pre-execution codes to synchronize backbone and rib threads.
6. Insert kill to terminate pre-execution threads.

```
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_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);
}
```

Figure 6: Code generated by the compiler, appearing in boldface, to implement the A. DoAll **scheme for the VPR benchmark and B.** DoAcross **scheme for the TWOLF benchmark.**

In Figure 5a, we show the algorithm for selecting the pre-execution regions. Our algorithm takes as input the global loop nest graph, G_L (this graph specifies loop nesting within procedures as well as between procedure calls, using the program's procedure call graph to construct the loop nest structure across the entire program). We visit all loops in graph G_L in inner-most to outer-most order. Inner-most loops that iterate at least 25 times are selected outright as pre-execution regions. Next-outer loops are selected as pre-execution regions as long as they do not contain any inner-most loops already selected as pre-execution regions (*i.e.*, we do not permit nesting of pre-execution regions). Finally, after all loops are traversed, we visit all the inner-most loops again, adding to the list of pre-execution regions those inner-most loops whose parent loops have not been selected as a pre-execution region (this handles inner-most loops that iterate fewer than 25 times, but are not pre-executed from the next-outer loop because a "sibling loop" in graph G_L was selected as a pre-execution region).

After selecting the pre-execution regions, we choose a thread initiation scheme for each pre-execution region using the algorithm in Figure 5b. This algorithm chooses Serial for pre-execution regions where program slicing and prefetch conversion have removed all blocking loads. 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, our algorithm speculatively parallelizes the loop at the top of the pre-execution region to tolerate the long-latency memory stalls, using DoAll and DoAcross for affine and pointer-chasing loops, respectively.

4.3 Generating Code

Figure 5c shows the calls to code generation routines for each thread initiation scheme type, and Figure 6 illustrates the steps involved in generating code for two of the schemes. In Figure 6A, we illustrate the steps for generating DoAll scheme code, applying DoAll to the outer loop of the VPR benchmark from Figure 3. In addition to inserting thread initiation code, notice a counting semaphore is inserted as well. 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. Figure 6B illustrates the steps for generating DoAcross scheme code, using a pointer-chasing loop from the TWOLF benchmark as an example. Our DoAcross implementation performs the serialized loop induction variable updates in a separate thread, called the "backbone" thread, which initiates other threads to perform the loop body computations in parallel, called "rib" threads. A semaphore is used to keep the backbone thread from getting too far ahead of the

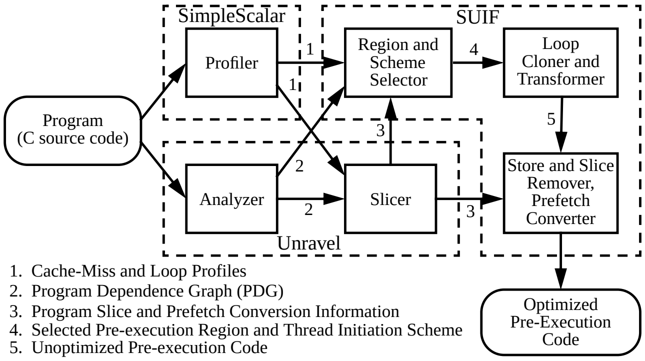

SimpleScalar
SUIF
Profiler
Region and Scheme Selector
Loop Cloner and Transformer
Program (C source code)
Analyzer
Slicer
Store and Slice Remover, Prefetch Converter
Unravel
Optimized Pre-Execution Code
1
1
2
2
3
3
4
5
1. Cache-Miss and Loop Profiles
2. Program Dependence Graph (PDG)
3. Program Slice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

Figure 7: Major components in the prototype compiler, and how they interact.

main thread, and multiple semaphores are used to synchronize the backbone and rib threads during communication of each induction variable value.

The algorithm in Figure 5c calls two other code generation routines that we omit to conserve space. The `generate_Serial` routine is very similar to the `generate_DoAll` routine illustrated in Figure 6A because the SERIAL scheme is a degenerate case of the DOALL scheme. Finally, the `clone_and_optimize` routine clones all procedures called within pre-execution regions (computed in Figure 5b), and applies slicing, store removal, and prefetch conversion to the cloned code.

5. IMPLEMENTATION

This section discusses implementation issues. First, Section 5.1 describes our prototype compiler. Then, Sections 5.2 and 5.3 discuss the ISA and thread-level support assumed by our compiler.

5.1 Prototype Compiler

We prototyped the algorithms from Sections 3 and 4 in a compiler consisting of three major components. First, we use a modified cache simulator from the SimpleScalar toolset [3] to acquire the cache-miss and loop iteration count profiles required by our algorithms. Second, we modify Unravel to implement the program slicing algorithms described in Section 3.2. This tool also performs the prefetch conversion analysis from Section 3.3. Finally, we use SUIF to implement the pre-execution thread initiation algorithm presented in Section 4.2, and to perform the code transformations described in Section 4.3. When producing the final pre-execution code, our SUIF tool removes stores and sliced code, and converts blocking loads to prefetches, all guided by the program slicer output. Figure 7 illustrates these modules and shows how they interact.

5.2 ISA Support

Our compiler assumes the SMT processor provides 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 a `suspend` and `resume` instruction. Both instructions specify a hardware context ID to suspend or resume. The processor state of suspended contexts remain in the processor, but the associated thread discontinues fetching and issuing instructions after the suspend. Both instructions execute in 1 cycle; however, `suspend` causes a pipeline flush of all instructions belonging to the suspended context. Finally, we assume a `kill` instruction that terminates all currently active pre-execution threads. Only the main thread can execute `kill` instructions.

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 jumps indirect through the PC argument. Upon completion, the forked thread returns to the dispatch loop and suspends itself until the next fork, thus recycling the thread.

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, effectively blocking.

6. EXPERIMENTAL RESULTS

This section reports our experimental results. First, we describe our methodology. Next, we present performance results for our compiler-based pre-execution technique. Then, we study our compiler algorithms in depth. Finally, we examine architectural support for our pre-execution threads.

6.1 Methodology

To evaluate compiler-based pre-execution, we generate pre-execution code for several applications. First, we acquire cache-miss and loop iteration count profiles using our profiler. Then, our compiler performs program slicing analysis, thread initiation scheme selection, and pre-execution code generation following the algorithms described in Sections 3 and 4. To reduce pre-execution overhead, our compiler only considers loops whose aggregate cache-miss counts exceed 3% of the total cache misses simulated when selecting pre-execution regions (*i.e.*, when using the algorithm in Figure 5a). All profile runs and compiler analyses are performed automatically, without any manual intervention.

Table 1 lists the 13 applications used in our study. Unfortunately, there are 5, 10, and 8 applications from the SPECInt, SPECfp, and Olden suites, respectively, that we could not study. One SPECInt and all 10 SPECfp remaining codes are not written in C. Of the other SPECInt codes,

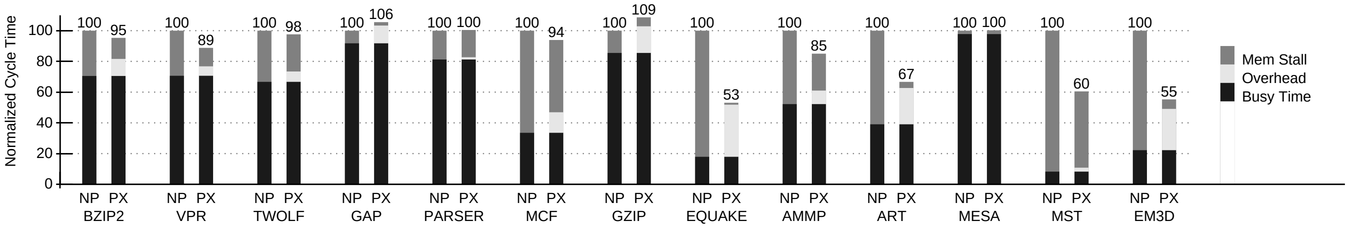


Figure 8: Normalized execution time broken down into busy, overhead, and memory stall components. The "NP" and "PX" bars show performance for no pre-execution and with pre-execution, respectively.

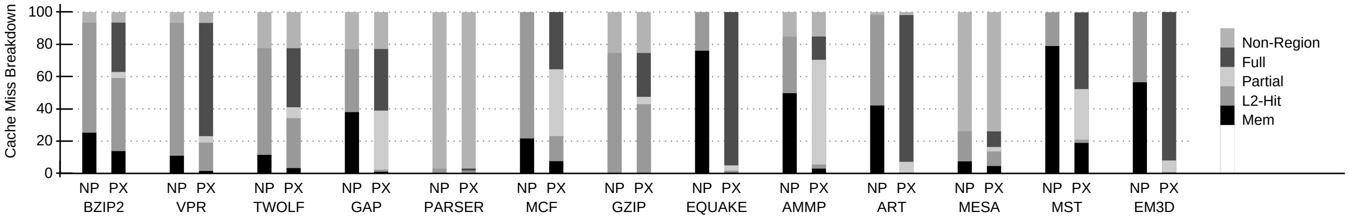


Figure 9: Cache-miss coverage broken down into fully covered and partially covered by pre-execution, labeled "Full" and "Partial," respectively. Remaining misses either hit in the L2, or are satisfied from main memory, labeled "L2-Hit" and "Mem," respectively.

Suite	Name	Input	FastFwd	Sim	PXR
SPECInt 2000	256.bzip2	reference	186,166,461	123,005,773	3
	175.vpr	reference	364,172,593	130,044,367	3
	300.twolf	reference	124,205,119	111,068,899	6
	254.gap	reference	147,924,013	127,932,004	1
	197.parser	reference	245,292,554	137,427,886	4
	181.mcf	reference	12,149,459,578	137,280,363	1
	164.gzip	reference	162,221,430	135,946,580	1
SPECfp 2000	183.equake	reference	2,570,651,646	21,850,552	1
	188.ammp	reference	2,439,723,993	129,357,604	3
	179.art	reference	12,899,865,395	113,811,999	7
	177.mesa	reference	262,597,285	52,683,249	1
Olden	mst	1024 nodes	183,274,940	24,361,256	1
	em3d	20K nodes	53,331,921	108,341,604	1

Table 1: Benchmark characteristics.

Processor Pipeline:	8-way issue SMT processor with 4 hardware contexts;
	Instruction fetch queue = 32 entries; Load-store queue = 64 entries;
	RUU size = 128 entries; 8 integer, 4 floating-point functional units;
	Latency - int add = 1 cycle; int mult = 3 cycles; int div = 20 cycles;
	fp add = 2 cycles; fp mult = 4 cycles; fp div = 12 cycles;

Branch Predictor:	Gshare predictor = 2K entries; Return of stack = 8 entries;
	Branch target buffer = 2K entries, 4-way set-associative;

Memory Hierarchy:	Split L1 I & D cache: 32KB, 2-way set-associative, 32 byte block;
	Unified L2 cache: 1MB, 4-way set-associative, 64 byte block;
	L1 hit time = 1 cycle; L2 hit time = 10 cycles;
	Main memory access time = 122 cycles;

Table 2: SMT simulator settings.

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 codes perform recursive tree traversals which our compiler does not analyze. In Table 1, the column labeled "Input" reports the inputs used to run each application. The next two columns, labeled "FastFwd" and "Sim," specify the number of skipped and simulated instructions, respectively, in our simulation regions. 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. Finally, the last column, labeled "PXR," reports the number of pre-execution regions our compiler identifies based on the cache-miss profiles. There are 33 in all.

We run our compiler-generated pre-execution threads on the SMT simulator from [15], which is derived from SimpleScalar's out-of-order processor model [3]. 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 [22]. Finally, the ISA support described in Section 5.2 has been provided. Table 2 reports the simulator settings used for our experiments.

6.2 Baseline Pre-Execution Performance

Figure 8 presents the performance of our compiler-based pre-execution technique. We report execution time without pre-execution, labeled "NP," and with pre-execution, labeled "PX," broken down into three 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 times are normalized against the "NP" bars.

Our compiler-based pre-execution technique reduces execution time for 9 out of 13 applications. Execution time is reduced between 2% and 47%, providing a 22.7% reduction on average for all 9 applications. Of the remaining 4 applications, 2 do not receive any gain, and 2 experience a degradation in performance by 7.5% on average. Overall, our technique delivers an average speedup of 17.0% across all 13 applications.

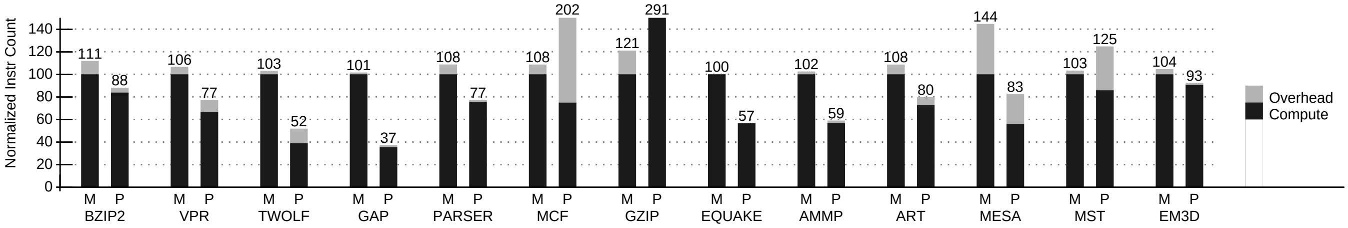


Figure 10: Number of dynamic instructions removed by program slicing. The "M" and "P" bars report instruction counts for the main and pre-execution threads, respectively. Each bar is broken down into compute and overhead components.

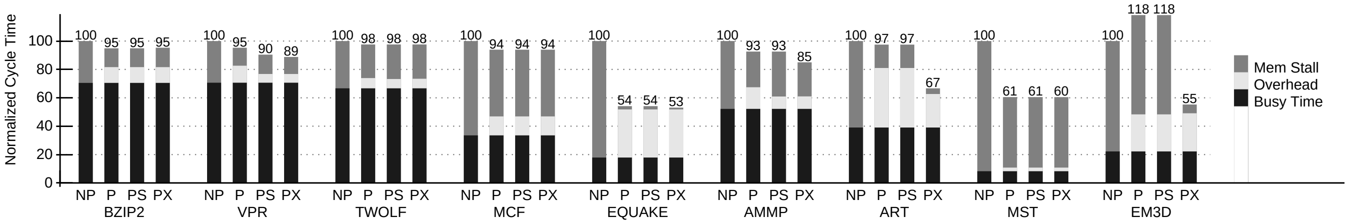


Figure 11: Performance impact of program slicing and prefetch conversion. The "P" bars show pre-execution alone, the "PS" bars show pre-execution with slicing but without prefetch conversion, and the "PX" bars show pre-execution with both program slicing and prefetch conversion.

To provide more insight, Figure 9 reports cache-miss coverage. The "NP" bars in Figure 9 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 "PX" 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.

Our compiler effectively covers cache misses for VPR, GAP, MCF, EQUAKE, AMMP, ART, MST, and EM3D, converting 84.9% of the main thread's misses 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%. Three factors contribute to reduced coverage. First, our compiler does not consider loops with fewer than 3% of the total cache misses. This is responsible for the "Non-Region" components in Figure 9, 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.

The third factor contributing to reduced cache-miss coverage is incorrect pre-execution code resulting from store removal or speculative loop parallelization. Section 3.2 already discussed the correctness issues regarding store removal. In speculative loop parallelization, our compiler performs data dependence analysis on loop induction variables only (see Section 4.1); hence, sometimes we mistakenly parallelize loops containing loop-carried dependences through other variables. Although our compiler's speculative transformations cannot compromise main-thread correctness, they may prevent pre-execution threads from faithfully mimicking the main thread's memory reference stream, reducing cache-miss coverage. Out of 33 pre-execution regions, we found pre-execution code failed to cover some cache misses in only 3 regions due to either store removal or speculative loop parallelization (these occurred in VPR, MCF, and GZIP).

Why are speculative transformations successful most of the time? Control and data flow leading up to cache-missing memory references 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 computations involving induction variables is a necessary (though not sufficient) condition for high cache-miss coverage. Since store removal only removes stores to global or heap variables, it is unlikely to remove anything critical for computing induction variables which are typically kept in registers or on the stack. Furthermore, since our loop parallelization techniques perform induction variable analysis, we usually capture the necessary control and data flow for computing induction variable values correctly.

6.3 Evaluating Program Slicing

Having quantified the performance gains of compiler-based pre-execution, we now study the compiler algorithms in greater detail to understand how each individually contributes to overall performance. This section examines program slicing. We begin our study by measuring the number of instructions program slicing removes. Figure 10 reports normalized dynamic instructions executed in the main thread ("M" bars) and pre-execution threads ("P" bars), broken down into compute and overhead components. The latter reports 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.

Without slicing, the main thread and pre-execution threads should execute the same number of instructions (excluding overhead). Comparing the compute components in the "M" and "P" bars from Figure 10, we see pre-execution threads execute 33.8% fewer instructions than the main thread across 12 applications, due to program slicing. 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). If we include overhead instructions, we see pre-execution threads execute 29.7% 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. Overall, Figure 10 shows program slicing can effectively reduce computation performed in pre-execution threads.

Although program slicing effectively removes non-critical instructions, the bottom line is the impact this has on application performance. Figure 11 addresses this important issue. In Figure 11, we apply optimizations incrementally to the 9 applications our compiler speeds up, and measure the change in performance provided by each optimization. The "P," "PS," and "PX" bars report pre-execution performance without any optimizations, with program slicing only, and with both program slicing and prefetch conversion, respectively. Note, store removal is always applied (except for "NP"); otherwise, pre-execution threads would crash the main thread. Also, the Serial scheme is never used in "P" and "PS" since pre-execution regions always contain blocking loads for these experiments (DoAll and DoAcross schemes are selected instead).

In 5 out of the 9 applications, program slicing does not play a role in improving performance. Figure 11 shows the entire speedup for BZIP2, TWOLF, MCF, EQUAKE, and MST is achieved when going from "NP" to "P"; no additional performance gain is achieved by program slicing and prefetch conversion. In contrast, half of the performance gain in VPR and AMMP, and all of the performance gain in ART and EM3D occur when going from "P" to "PX." Surprisingly, however, it is the combination of prefetch conversion with program slicing that accounts for most of this gain. While program slicing alone speeds up VPR, all of the gain in AMMP, ART, and EM3D is attributable to prefetch conversion (*i.e.*, going from "PS" to "PX"). In these applications, the pre-execution threads cannot tolerate all of the memory stalls; hence, by reducing the frequency of stalls in each pre-execution thread, prefetch conversion improves performance. From this data, we conclude program slicing is important, especially since it enables prefetch conversion.

6.4 Evaluating Pre-Execution Initiation

This section evaluates our pre-execution thread initiation schemes. In Figure 12, 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 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 region, we apply the Serial, DoAll, and DoAcross schemes, labeled "SE," "DA," and "DX," respectively, and compare their performance. The schemes selected by our compiler appear in boldface.

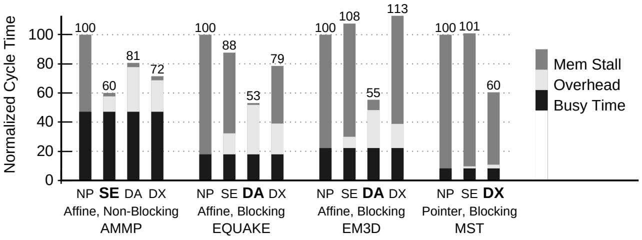


Figure 12: Comparing pre-execution thread initiation schemes. "NP" bars report execution time for no pre-execution. "SE," "DA," and "DX" bars report the execution time when the Serial, DoAll, and DoAcross schemes are applied, respectively. The schemes selected by our compiler appear in boldface.

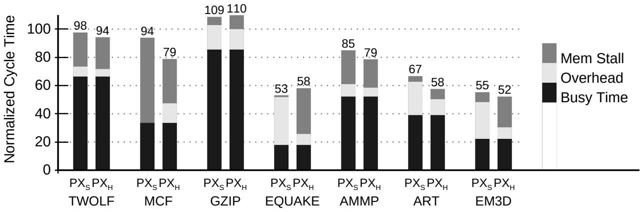


Figure 13: Impact of hardware support for synchronization. "PX_S" and "PX_H" bars report execution time with software and hardware synchronization, respectively. All bars are normalized to the "NP" bars from Figure 8.

Figure 12 shows three results. First, choosing the right thread initiation scheme has a first-order impact on pre-execution performance. In Figure 12, the best scheme for each pre-execution region outperforms the worst scheme by 25.9%, 39.8%, 51.3%, and 40.6% for AMMP, EQUAKE, EM3D, and MST, respectively.

Second, Figure 12 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 12, thus validating the accuracy of the selection algorithm presented in Section 4.2.

6.5 Architectural Support for Pre-Execution

Thus far, we have evaluated compiler-based pre-execution assuming only conventional SMT hardware. This section studies the impact of hardware support for synchronization on pre-execution performance. We implemented hardware semaphores that permit single-cycle "P" and "V" operations to be performed on hardware semaphore registers. Furthermore, we block and resume threads in hardware, replacing the busy-waiting code used in our software synchronization (see Section 5.3). Figure 13 compares pre-execution with software and hardware synchronization primitives, labeled "PX_S" and "PX_H" respectively, for 7 applications (for the remaining 6 applications, there is no difference). All bars are normalized to the "NP" bars from Figure 8. Figure 13 shows hardware support improves performance by 4.4%. Across all 13 applications, we find hardware semaphores increase the overall speedup of pre-execution from 17.0% to 20.4%.

7. CONCLUSION

This paper presents algorithms for creating pre-execution thread code in a compiler, and prototypes them using the SUIF framework and Unravel. On a detailed SMT simulator, we show our compiler-generated pre-execution threads reduce execution time by 22.7% for 9 out of 13 applications, and delivers an average speedup of 17.0% across all 13 applications. We find these performance gains are due in part to the fact that cache-miss addresses are frequently derived from induction variables. While our store removal and loop parallelization transformations are speculative, they preserve the correctness of induction variable code most of the time; hence, our pre-execution code correctly generates addresses for a large number of cache misses.

In studying our algorithms individually, we find program slicing and prefetch conversion are responsible for half to all of the performance gains in 4 out of the 9 applications our compiler speeds up. Moreover, while program slicing is crucial, its primary value lies in enabling prefetch conversion rather than reducing pre-execution overhead. We also find careful selection of the pre-execution thread initiation scheme 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.

We believe our work demonstrates that compilers can effectively orchestrate pre-execution in SMT processors. At the same time, we view this paper only as a starting point. Future work includes improving algorithms to construct more effective pre-execution threads as well as to detect when pre-execution is ineffective, developing static analyses to supplement or possibly replace profile information, and conducting further applications studies.

8. ACKNOWLEDGMENTS

We thank Seungryul Choi for providing tools to gather loop profile information and blocking load information, and Gautham Dorai for helpful discussions regarding the implementation of the SMT simulator. We also thank Chau-Wen Tseng and the anonymous reviewers for their constructive comments on previous drafts of this paper.

9. REFERENCES

[1] M. Annavaram, J. Patel, and E. Davidson. Data Prefetching by Dependence Graph Precomputation. In *28th International Symposium on Computer Architecture*, June 2001.

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

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

[4] R. Chappell, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). In *26th International Symposium on Computer Architecture*, May 1999.

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

[6] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *34th International Symposium on Microarchitecture*, December 2001.

[7] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *28th International Symposium on Computer Architecture*, June 2001.

[8] R. Cytron. Doacross: Beyond Vectorization for Multiprocessors. In *International Conference on Parallel Processing*, August 1986.

[9] M. Dubois and Y. Song. Assisted Execution. CENG TR 98-25, University of Southern California, October 1998.

[10] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss. In *International Conference on Supercomputing*, July 1997.

[11] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[12] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *28th International Symposium on Computer Architecture*, June 2001.

[13] J. Lyle and D. Wallace. Using the unravel program slicing tool to evaluate high integrity software. In *10th International Software Quality Week*, May 1997.

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

[15] D. Madon, E. Sanchez, and S. Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *EuroPar '99*, August 1999.

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

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

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

[19] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[20] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *7th International Conference on High Performance Computer Architecture*, January 2001.

[21] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving Both Performance and Fault Tolerance. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, May 2000.

[22] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *23th International Symposium on Computer Architecture*, May 1996.

[23] C. Zilles and G. Sohi. Execution-Based Prediction Using Speculative Slices. In *28th International Symposium on Computer Architecture*, June 2001.

[24] C. Zilles and G. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *27th International Symposium on Computer Architecture*, June 2000.