University of Maryland Inistitute for Advanced Computer Studies UMIACS-TR-2001-49

Using Program Slicing to Drive Pre-Execution on Simultaneous Multithreading Processors

Dongkeun Kim and Donald Yeung
Electrical and Computer Engineering Department
Institute for Advanced Computer Studies
University of Maryland at College Park

Abstract

Pre-execution uses helper threads running in spare hardware contexts to trigger cache misses in front of the main thread, hence hiding their latency. At the heart of pre-execution is the code that runs in the pre-execution threads themselves. The most common approach is for pre-execution threads to run a subset of the instructions executed by the original program, called *backward slices* [18], which are extracted from the main thread at the instruction level.

This paper proposes a new pre-execution technique that uses *program slicing* [2] to extract the code for pre-execution threads. Program slicing performs static analysis on the program source to create slices consisting of source code rather than binary code. Compared to previous techniques, our approach requires less hardware, and is more natural to automate in a compiler. To study the feasibility of our approach, we built a slicing system based on a publicly available program slicer, called *Unravel*, that constructs program slices for pre-execution. We also developed several program slice parallelization techniques that partition our program slices onto multiple pre-execution threads. Our techniques enable pre-execution threads to effectively get ahead of the main thread by exploiting thread-level parallelism. Finally, our work provides an evaluation of program slice driven pre-execution using a detailed simulator of a simultaneous multithreading (SMT) processor. Our techniques achieve a 27.4% speedup across 7 integer applications on an 8-way SMT with 4 contexts, and a 56.7% speedup on an SMT with 9 contexts.

1 Introduction

Processor performance continues to be limited by long-latency memory operations. In the past, researchers have studied prefetching [11, 3] to tolerate memory latency, but these techniques are ineffective for irregular memory access patterns common in many important integer applications. Recently, a more general latency tolerance technique has been proposed, called *pre-execution* [1, 4, 7, 17, 13]. Pre-execution uses idle execution resources, for example spare hardware contexts in a simultaneous multithreading (SMT) processor [16, 15], to run one or more helper threads in front of the main thread. 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.

At the heart of pre-execution is the code that runs in the pre-execution threads themselves. Amongst existing proposals, the most common approach is for pre-execution threads to run a subset of the instructions executed by the original program, called *backward slices* [18]. A backward slice is a sequence of instructions extracted from the original program executable that leads to a cache-missing load. The backward slice contains the instructions upon which the cache-missing load is data or control dependent. Backward slices can be extracted by analyzing either the program binary [17], or a simulator-generated instruction trace [4, 13, 18].

In this paper, we propose a new pre-execution technique that uses *program slicing* [2] to extract the code for pre-execution threads. Like instruction-level backward slices, program slices represent a smaller version of the original program necessary to pre-execute the cache-missing loads. However, instead of extracting the pre-execution code through instruction-level analysis, program slicing performs the code extraction directly on the program source using static code analysis techniques, resulting in a slice that consists of source code rather than binary code.

Compared to previous proposals for slice-driven pre-execution, our program slicing approach differs in two major ways. The first difference, as already mentioned, is that program slicing extracts slices from source code rather than binaries or traces. Because our approach uses high-level information to construct the slices, it is a more natural approach to integrate into a compiler. In fact, researchers have already developed compiler-based tools to extract much of the information we need automatically–our work leverages these techniques. Also, because program slices are themselves source code, they can be created easily by a programmer in the absence of compiler support, and they are portable across architectures. In contrast, instruction-level slices require architecture-specific binary or trace analysis that is cumbersome for humans to carry out.

The second major difference is that our approach uses less hardware. Backward slices rely on special markers, called "triggers," to mark fork points in the main thread's instruction stream. Hardware support is provided to fork pre-execution threads as the triggering instructions move through the pipeline. Also, hardware support is used to pass arguments from the main thread to the pre-execution threads, either by copying mappings in the register rename stage [17, 13] or by using hardware-based communication buffers [4]. In contrast, our approach supports thread creation and communication entirely in software. The source-level nature of our slices makes this software support easy to implement. Since backward slices exist only as binaries, similar software solutions would require binary editing, and may not be possible if the binaries do not contain the proper symbol information. Granted, software solutions like ours incur higher overheads, but we show these overheads are manageable, making the hardware savings of our approach attractive.

This paper investigates the use of program slices for pre-execution on simultaneous multithreading processors. Our work makes three contributions. First, we built an experimental slicing system that extracts program slices for pre-execution. Based on a publicly available program slicer called *Unravel* [8], our slicing system demonstrates that existing program slicers can perform many of the analyses necessary to extract slices for pre-execution automatically. Second, we present several program slice parallelization techniques that partition the program slices onto multiple pre-execution threads. Our techniques enhance the ability of pre-execution threads to get ahead of the main thread particularly for pointer-chasing loops, thus improving the degree of memory latency tolerance. Finally, we conduct an experimental evaluation of program slice driven pre-execution using a detailed simulator of an SMT processor. Our results show pre-execution with program slices achieves an average speedup of 27.4% across 7 applications on a baseline SMT configuration. On a more aggressive SMT, the average speedup increases to 56.7%.

The rest of this paper is organized as follows. Section 2 describes our experimental slicing system. Next, Section 3 presents our program slice parallelization techniques, and Section 4 discusses several implementation issues. Then, Section 5 presents our results, and Section 6 discusses related work. Finally, Section 7 concludes the paper.

2 Program Slicing

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 the original program responsible for computing the slice criterion.

Program slicing has numerous applications in the area of software evaluation, including debugging, testing, parallelization, and maintenance of high-integrity software. Because program slices are more compact than the original program, a programmer can use a program slicing tool to focus his/her attention on the code that is responsible for computing a value of interest, rather than

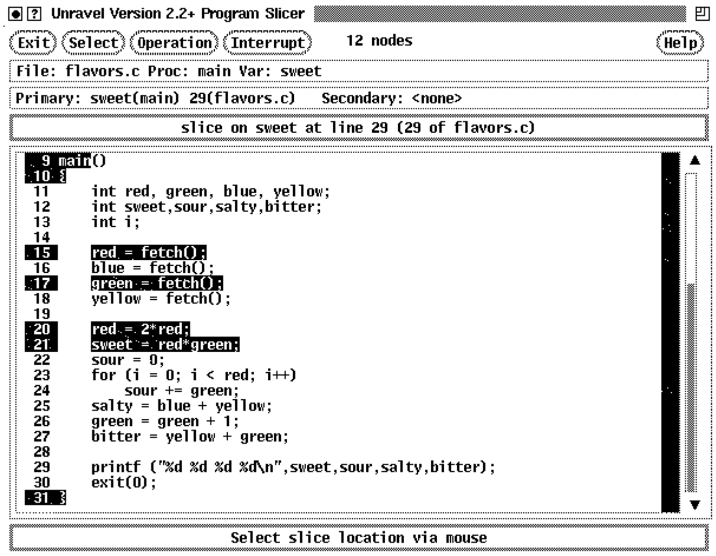

Figure 1: An example interactive session with Unravel.

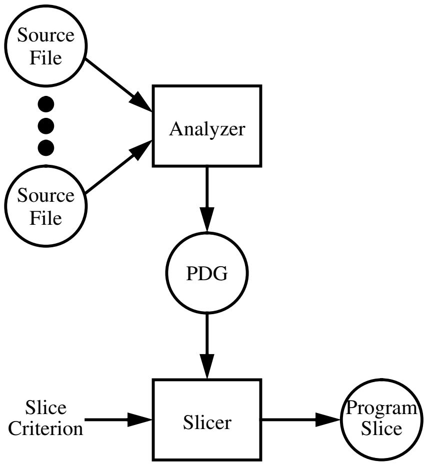

Figure 2: The program slicing engine in Unravel consists of two modules, an analyzer and a slicer.

considering the entire program. Consequently, both the number errors and the amount of effort can be significantly reduced during the evaluation of a complex piece of code by using a program slicing tool to assist the programmer.

Program slicing has been developed in the context of software evaluation; however, we find that existing slicing algorithms can extract slices for pre-execution as well. This section describes an experimental slicing system for pre-execution that we have built based on an existing program slicer, called *Unravel* [8]. Section 2.1 gives a brief overview of Unravel, and Section 2.2 describes our modifications to create program slices for pre-execution.

2.1 Unravel

Unravel is a publicly available program slicer for ANSI C developed at the National Institute of Standards and Technology (NIST).[1] It uses an X-Windows GUI to display the user's program, and allows the user to interactively specify a slice criterion, which consists of a source line number and a variable name. Once a slice criterion has been specified, Unravel computes the program slice, and displays the result interactively by highlighting the source code contained in the slice. Figure 1 shows a typical Unravel session. In the figure, the slice criterion is the local variable "sweet" at line 29 of procedure "main," and the program slice computed by Unravel is highlighted.

[1]NIST provides source code for Unravel at http://www.itl.nist.gov/div897/sqg/unravel/unravel.html.

Underneath Unravel's GUI is the slicing engine which consists of two modules, an analyzer and a slicer, as illustrated in Figure 2. The analyzer parses the .c and .h source files and generates a program dependence graph (PDG) [6] across the entire program. The slicer uses the PDG to compute 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.

In this section, we can only give a brief overview of Unravel. In fact, Unravel is quite sophisticated and can handle many language features found in C. In particular, Unravel can slice across procedure boundaries and source code modules (as long as they are all provided to the analyzer). Also, Unravel handles data dependences through individual structure members, and performs pointer analysis. Unfortunately, a complete exposition of Unravel is beyond the scope of this paper. To obtain more details on Unravel, see [9].

2.2 Slices for Pre-Execution

We use Unravel to compute program slices for "problematic" 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: slicer interface, slice size, slice merging, and side effects. This section describes our modifications using the code example from the MST benchmark [12] shown in Figure 3a. Later in Section 3, we will discuss how our program slices are actually used for pre-execution.

Slicer Interface. We stripped away Unravel's X-Windows GUI and modified the slicer engine to run in batch mode so that we can compute program slices without programmer intervention. To pick the problematic 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

Figure 3: a). MST example code. Labels "1"-"5" indicate the cache-missing memory references selected for slicing. Labels "6" and "7" indicate loops that bound the scope of slicing. Labels "S1," "S2," and "S5" show the slice result for the selected memory references. Label "8" indicates a memory reference that has a side effect. b). Final slice code for MST after all slicing steps have been applied.

translate each of the load PCs into a source code line number and variable name. In Figure 3a, the five most cache-missing memory references identified through profiling appear in bold-face with arrows labeled "1"-"5" pointing to them. The load instructions associated with these five memory references account for 90% of the cache misses in MST. Each of these memory references is used as the slice criterion during a single slicing run.

Slice Size. Slice size determines how much code will get pre-executed for each problematic load instruction, and plays a crucial role in affecting the performance of pre-execution. If the slice is too large, overhead will increase and offset the gains of memory latency tolerance. If the slice is too small, opportunities for exploiting memory parallelism may be sacrificed.

Unravel computes program slices across the entire program. Such slices are too large for pre-execution. We modified the slicer to limit the scope of slicing. As we will see in Section 3, most of our slice optimizations exploit parallelism either in the inner-most loop or the next outer loop.

Hence, we terminate slicing once we have encountered either one or two looping statements above the slice criterion. Section 3 will discuss how to choose amongst these two slice sizes, but for now, we assume the scope of slices is two looping statements. In Figure 3a, memory references 1, 3, and 4 are contained inside the loop labeled "6." The next outer loop, labeled "7," is where slicing terminates for these three memory references. Memory references 2 and 5 are contained inside the loop labeled "7." The next outer loop, which is not shown in Figure 3a, is where slicing terminates for these two memory references.

An important feature of our slice termination policy is that we allow slices to span multiple procedures. From our experience, problematic memory references often occur inside procedures that are called from loops, and loops are often nested across procedure boundaries. This is certainly true for MST. Fortunately, Unravel performs inter-procedure analysis.

In addition to exploiting loop-level parallelism, our slice optimizations also extract parallelism in recursive tree traversals (see Section 3). For recursion, we terminate slicing at the top of the recursive procedure.

Slice Merging. After the slicing analysis completes, we have a program slice for each sliced memory reference. Figure 3a illustrates the slices computed for the five problematic memory references in MST by placing a small arrow to the left of each source code line contained in the slice. The slices for memory references 1, 2, and 5 are specified by the boxed set of arrows labeled "S1," "S2," and "S5," respectively. The slices for memory references 3 and 4 are identical to the slice for memory reference 1, so we omit them from the figure. (Notice that slices S2 and S5 should continue up to the next outer loop, but we only show the portions of these slices associated with the code in Figure 3a). Internally, 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 that they can be pre-executed together using a common set of threads. 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.

Side Effects. Since pre-execution threads run speculatively, none of their instructions should modify any architectural state belonging to the main thread; otherwise, the pre-execution threads could inadvertently crash the main thread. Consequently, program slices should not perform stores to memory locations from which the main thread will subsequently load. Since pre-execution threads have private stacks, slices are allowed to store to the stack. However, stores to static global variables or heap variables must be removed or modified.

We handle slice code stores in one of two ways. First, if it is clear from the slice code that a stored value is used further down the slice, we create a static global variable and modify the store and load instructions to access the new variable, effectively renaming the variable. Renaming is important to preserve the correctness of the pre-execution thread with respect to the main thread. To minimize storage requirements, we only allocate storage for a single scalar for each renamed store-to-load pair, even if the memory references access non-scalar values. Hence, our simple renaming scheme can potentially alias different dynamic instances of the loads and stores, but should properly enforce the dependence locally (*e.g.*, within a single loop iteration). Second, if a stored value has no detectable use within the slice, we remove it and all associated code from the slice. In Figure 3a, the code with the label "8" pointing to it is such a memory reference. This line is removed from the slice, as well as the line previous to it.

Figure 3b shows the final program slice extracted from the MST source code after all the steps outlined above have been applied. This program slice contains 17 lines of C code, compared to 35 lines in the original program. Later in Section 5.2, we will more precisely quantify the reduction in code size achieved by our slicing system.

Currently, our slicing system is semi-automatic. Of the steps discussed in this section, creation of slice criteria via cache profiling, slice extraction, and slice merging are fully automated. The transformations for side effects are performed manually.

3 Pre-Execution Threads

In this section, we discuss how to assign program slices to pre-execution threads. First, Section 3.1 introduces our simplest thread assignment scheme, and illustrates the problem of maintaining the pre-execution thread's distance from the main thread. Second, Section 3.2 presents more sophisticated thread assignment schemes. Finally, Section 3.3 discusses how prefetch instructions can be inserted into slice code to further increase pre-execution thread distance.

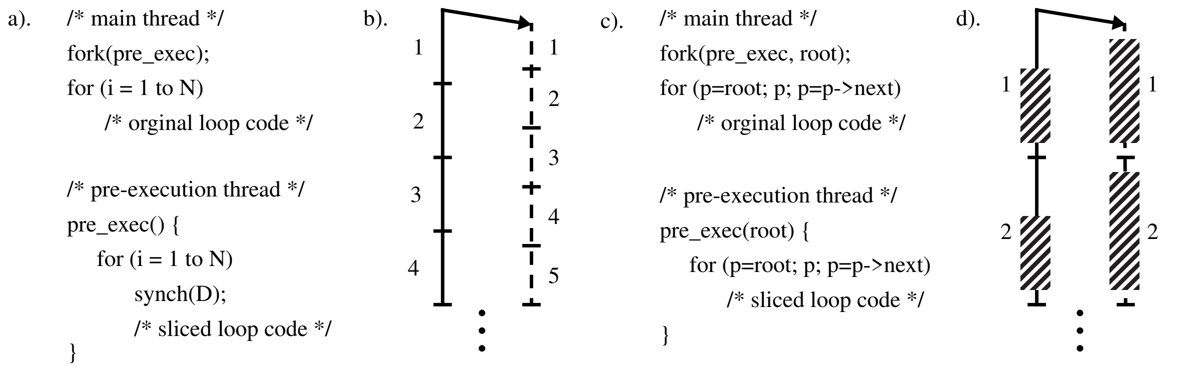

Figure 4: a). SIL scheme code. b). Ideal time-line. c). SIL scheme with pointer-chasing induction variable. d). Time-line with cache misses. Solid lines denote the main thread, dotted lines denote the pre-execution thread, arrows denote thread creation, and hash patterns denote memory stalls.

3.1 Single-Threaded Program Slices

The simplest approach is to execute all the code in a program slice on a single thread. Figure 4a shows the implementation of this thread assignment scheme. In the figure, a single pre-execution thread is forked from the main thread at the point where slicing was terminated–in this case, right above a loop. The pre-execution thread executes the sliced version of the loop, which is contained in the procedure, "pre_exec." As mentioned in Section 2.2, slice termination for loops can occur either at the inner-most loop or the next outer loop. To minimize slice overhead, slice termination at the inner-most loop is the best choice for this simple scheme. For this reason, we call this thread assignment the *sequential inner loop* (SIL) scheme.

Since the sliced loop is a smaller version of the original loop, the pre-execution thread should get ahead of the main thread. Figure 4b shows a time-line of the main thread (solid line) and the pre-execution thread (dashed line). Numbered segments denote loop iterations. The figure shows the pre-execution thread gets ahead of the main thread because it performs less work. To prevent the pre-execution thread from getting too far ahead, we insert a synchronization primitive, "synch(D)," that blocks the pre-execution thread once its lead becomes D iterations (we will discuss the synch primitive later in Section 4). We set D to the number of iterations needed to overlap a cache miss, $\lceil \frac{l}{w} \rceil$, where l is the memory latency and w is the amount of work per loop iteration.

In principle, the pre-execution thread should get ahead of the main thread; however, in many cases this does not happen due to memory stalls. Since program slices contain a large number of problematic memory references by design, the pre-execution thread suffers cache misses frequently. Particularly damaging are cache misses incurred during pointer chasing. Pointer-chasing cache

	a). Parallel Inner Loop	b). Parallel Outer Loop	c). Backbone and Ribs	d). Parallel Depth-First Tree

Main Thread	for (i = 1 to N)
	fork(pre_exec, i+D);
	/* original code */
		for (j = 1 to N)
		fork(pre_exec, A[j+D]);
		for (p=A[j]; p; p=p->next)
		/* original code */
			fork(pre_exec_backbone, root);
			for (q=root; q; q=q->next)
			for (p=q->root; p; p=p->next)
			/* original code */
				fork(pre_exec, root);
				tree(root);
				tree(node) {
				tree(node->left);
				tree(node->right);
				/* original code */
				}

Pre-Execution Thread	pre_exec(i) {
	/* slice code */
	}
		pre_exec(root) {
		for (p=root; p; p=p->next)
		/* slice code */
		}
			pre_exec_backbone(root) {
			for (q=root; q; q=q->next)
			synch(D);
			fork(pre_exec_rib, q->root);
			}
			pre_exec_rib(root) {
			for (p=root; p; p=p->next)
			/* slice code */
			}
				pre_exec(node) {
				while (node)
				fork(pre_exec, node->right);
				node = node->left;
				/* slice code */
				}

Figure 5: Four thread assignment schemes that multithread the program slice. a). Parallel inner loop (PIL), b). parallel outer loop (POL), c). backbone and ribs (BAR), and d). parallel depth-first tree (PDT).

misses must perform sequentially due to the serialization of pointer traversal. Figure 4c shows an example where the loop induction variable is a pointer-chasing reference. Figure 4d illustrates the time-line for this example assuming the pointer reference cache misses every iteration. Since the cache misses must perform sequentially, the pre-execution thread runs at the speed of one cache miss time per iteration at best. Some of the memory latency is hidden from the main thread, but the pre-execution thread cannot get ahead to overlap all the stalls.

3.2 Multithreaded Program Slices

To enable pre-execution to get ahead of the main thread more effectively, we assign each program slice onto multiple pre-execution threads. Multithreaded program slices enable pre-execution to get ahead even when some threads are blocked on cache misses, and handle pointer-chasing references by traversing multiple pointer chains simultaneously. In this section, we present three schemes for program slices with loops, and one scheme for program slices with recursion.

Parallel Inner Loop. Our first multithreaded scheme, called *parallel inner loop* (PIL), executes each iteration of the program slice inner loop in a separate thread. As shown in Figure 5a, the main thread forks a pre-execution thread for each loop iteration to pre-execute D iterations in front of the main thread. Hence, newly forked pre-execution threads jump ahead of the main thread without executing the intermediate loop iterations. Since each iteration executes in a separate pre-execution thread, cache misses can be triggered in parallel and overlapped. Figure 6a shows a time-line assuming $D = 2$. Even though each pre-execution thread runs at a slower rate than the

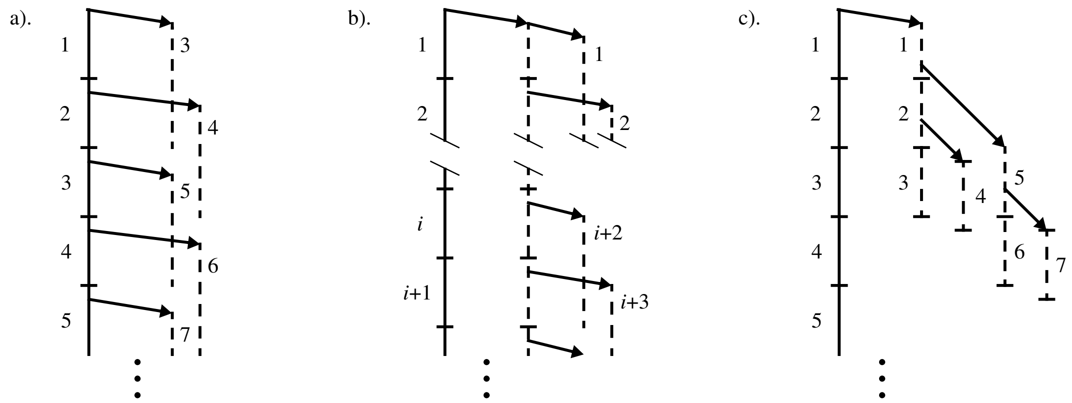

Figure 6: Thread-level parallelism realized by a). parallel inner loop (PIL) and parallel outer loop (POL) schemes, b). backbone and rib (BAR) scheme, and c). parallel depth-first tree (PDT) scheme. Solid lines denote the main thread, dotted lines denote pre-execution threads, and arrows denote thread creation.

main thread due to memory stalls (not shown), the pre-execution threads stay in front of the main thread by jumping ahead on each fork and overlapping their execution.

The PIL scheme has two limitations. First, it requires an arithmetic loop induction variable. PIL does not work for pointer-chasing loop induction variables, as in Figure 4c, since this would serialize loop iterations and prevent the main thread from forking a future iteration. Second, the PIL scheme becomes less effective if the number of inner loop iterations is small, resulting in insufficient work in the inner loop to tolerate memory latency.

Parallel Outer Loop. Our second multithreaded scheme, called *parallel outer loop* (POL), executes each iteration of the program slice outer loop in a separate thread. As shown in Figure 5b, this scheme forks pre-execution threads from within the next outer loop, each starting D iterations in front of the main thread. To provide the nested loop structure, the program slicer should terminate the slice two loop statements above the slice criterion, rather than just one (See Section 2.2). The POL time-line is identical to the PIL time-line, except the loop iteration segments in Figure 6a represent iterations of the outer loop rather than the inner loop.

The POL scheme exploits outer-loop parallelism, overlapping the execution of multiple inner loops in separate threads. Hence, it is ideal for inner loops that cannot be effectively pre-executed using the PIL scheme, including inner loops with pointer-chasing induction variables, as shown in Figure 5b, or small iteration counts.

Backbone and Ribs. Our third multithreaded scheme, called *backbone and ribs* (BAR), handles loops with pointer-chasing induction variables in both the inner and outer loops. As shown in

Figure 5c, the main thread forks a single pre-execution thread, called the *backbone thread*. The backbone thread traverses the pointer chain in the slice code's outer loop. In each iteration, the backbone thread forks another thread, called the *rib thread*, to pre-execute a single inner loop.

Since the backbone thread contains very little work, it will get ahead of the main thread (which has to execute an entire inner loop for each backbone iteration). As it gets ahead, the backbone thread will fork multiple rib threads, allowing the execution of multiple inner loops to overlap. Figure 6b shows the time-line for the BAR scheme. To prevent the backbone thread from getting too far ahead, we insert a "synch(D)" operation to keep it within D iterations of the main thread.

Notice both the POL and BAR schemes exploit parallelism between multiple dynamic instances of the inner loop. Consequently, they require the inner loops to be independent; otherwise, the inner loops must execute sequentially. However, in some cases, it may be possible to use the POL and BAR schemes even when dependences exist between inner loops by *speculating*. In Section 5.1, we will describe how to use speculation to parallelize program slices with inter-inner loop dependences using the BAR scheme.

Finally, the PIL, POL, and BAR schemes all have a D parameter. The D parameter determines how far ahead the pre-execution threads will get in front of the main thread, but it also determines the number of simultaneous threads. We choose D so that the concurrency never exceeds the number of available hardware contexts. Assuming N contexts, $D = N - 1$ for the PIL and POL schemes (leaving 1 context for the main thread), and $D = N - 2$ for the BAR scheme (leaving 1 context for the backbone and main threads each).

Parallel Depth-First Tree. Our fourth scheme, called *parallel depth-first tree* (PDT), assigns threads for program slices that perform tree traversal. To illustrate the approach, Figure 5d shows an example for a recursive depth-first traversal of a binary tree. The main thread forks a single pre-execution thread to execute the procedure, "pre_exec." The pre-execution thread traverses the left child pointer of the tree, and at each node, recursively forks a new pre-execution thread to traverse the right child pointer. Figure 6c shows the time-line for the PDT scheme assuming a tree of depth three (in this figure, the numbered segments represent tree nodes visited in depth-first order). As the figure shows, the pre-execution threads traverse the tree in parallel, getting ahead of the main thread.

In the PDT scheme, the number of threads forked grows exponentially with tree depth. To prevent thread explosion, we modify the main thread code to track the depth of recursion, and initiate parallel pre-execution only when it gets close to the bottom of the tree. Given N available contexts, we begin forking when the main thread reaches $log_2(N)$ levels above the bottom of the tree. This requires knowledge of the tree depth, which we obtain through profiling.

3.3 Prefetch Instructions

Multithreaded slice code enables pre-execution threads to get ahead of the main thread by exploiting thread-level parallelism. Another way to enhance pre-execution distance is to increase the speed of each pre-execution thread by reducing 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.

To insert prefetch instructions, we consider each problematic memory reference in the program slice. 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 replace the memory reference with a prefetch to the same memory address. In addition, we also insert prefetches for array references in sliced loops using a well-known software prefetching algorithm [11].

4 Implementation Issues

This section describes hardware and software support for our pre-execution technique. Section 4.1 discusses modifications to an SMT processor, Section 4.2 describes thread creation support, and Section 4.3 presents inter-thread communication support for our pre-execution threads.

4.1 SMT Support

We use a simultaneous multithreading (SMT) processor [16, 15] to execute our pre-execution threads alongside the main thread. We assume the SMT uses an ICOUNT instruction fetch policy [16] with a modification that gives highest priority to the main thread. We bias the ICOUNT policy to favor the main thread by initializing the instruction counts for the pre-execution threads to a non-zero offset value. The offset artificially lowers the priority of the pre-execution threads, giving more fetch bandwidth to the main thread.

We assume the SMT provides basic ISA support necessary for multithreading. First, we assume a `fork` instruction that specifies a hardware context ID and a PC in two registers. The fork instruction 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 the processor maintains an *active contexts register* containing a bit mask that specifies the state of each hardware context, either active or inactive. We assume all threads can read (but not write) this register.

4.2 Thread Creation

Thread creation accounts for a significant part of pre-execution overhead. Although forking a thread takes a single instruction, the overall cost of thread creation is much higher due to instructions for initializing the forked hardware context (our hand-optimized fork routine requires 25 instructions).

To reduce thread creation overhead, we implement a "recycled" thread model. We create a thread for each hardware context once during program initialization. Each pre-execution thread enters a dispatch loop that suspends itself. To perform a "fork," the thread initiating the fork communicates the PC value (Section 4.3 discusses how arguments are passed), 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. Our forking model assumes that hardware contexts are chosen in a round-robin fashion, managed in software. Before forking a thread, the active contexts register should be checked to see whether the next hardware context is idle. If not, the fork is aborted.

4.3 Inter-Thread Communication

Inter-thread communication occurs during thread creation to pass arguments, and during thread synchronization. In both cases, we perform the communication through memory. Each thread allocates a buffer in memory for arguments. Arguments are passed through loads and stores to the buffer. The overhead for passing arguments through memory is small because only a few arguments are usually passed, and the argument buffers normally remain in the L1 cache. To implement the "synch" primitive introduced in Section 3, we allocate a global counter in memory. During

Parameter	Value	Parameter	Value
Hardware contexts	4	BTB Size	2048 entries
Issue Width	8	Instruction Window Size	128
Fetch Queue Size	32	Load-Store Queue Size	64
Integer/Floating Point Units	8/4	L1/L2 Cache Size	16K/512K
Integer Latency	1 cycle	L1/L2 Block Size	32/64 bytes
Floating Add/Mult/Div Latency	2/4/12 cycles	L1/L2 Associativity	2/4
Branch Predictor	2-bit bimodal	L1/L2 Hit Time	1/10 cycles
Branch Predictor Size	2048 entries	Memory Access Time	70 cycles

Table 1: Baseline SMT processor configuration.

each iteration, the main thread increments the counter. Simultaneously, the pre-execution thread maintains a private counter, and compares its counter to the global counter each loop iteration. The pre-execution thread continues only if the difference between the counters does not exceed D, the argument to the "synch" primitive. Otherwise, the pre-execution thread busy waits. While busy waiting increases pre-execution overhead, our results show this overhead is relatively small.

5 Results

This section reports our experimental results. First, we describe our simulation environment. Next, we present results on a baseline SMT configuration, and then on several different SMT configurations. Finally, we study the contributions of our different techniques to overall performance.

5.1 Simulation Environment

Our evaluation uses the SMT simulator from [10]. This simulator uses the out-of-order model from SimpleScalar v2.0 to simulate the SMT pipeline, augmenting the basic simulator to model multiple hardware contexts and issue logic that selects instructions from one or more threads per cycle. In addition, we implemented the modified ICOUNT fetch policy that gives priority to the main thread by initializing the pre-execution thread instruction counts to a non-zero offset value (see Section 4.1). In our simulations, we use an offset value of 32, which was experimentally found to yield good performance.

Many of our experiments assume a "baseline" SMT configuration consisting of an 8-way SMT with 4 hardware contexts. This configuration is similar in resources to the Alpha 21464 [5], a commercial SMT processor. Table 1 reports in detail the parameters for our baseline SMT.

Application	Compress95	BZIP2	VPR	GZIP	TWOLF	MST	Treeadd
Parallelization Scheme	BAR	SIL	POL	SIL	BAR	BAR	PDT
Simulation Region	44.50	45.77	59.91	122.89	36.65	2.96	33.55
Pre-Execution Region	44.41	21.26	41.98	47.70	22.73	2.70	26.21
Program Slice	17.25	10.51	32.33	27.83	2.61	1.69	22.28
	0.39	0.49	0.77	0.58	0.11	0.63	0.85
Pre-Execution Thread	51.76	36.69	53.82	39.29	6.53	19.28	30.31
	1.17	1.73	1.28	0.82	0.29	7.14	1.16

Table 2: Benchmark characteristics. The rows report the slice parallelization scheme, instructions simulated, instructions executed in pre-execution regions, instructions in program slices, and instructions executed by pre-execution threads. All instruction counts are in millions.

Table 2 lists the applications we use: Compress95 from SPEC '95, BZIP2, VPR, GZIP, and TWOLF from SPEC 2000, and MST and Treeadd from Olden [12]. The first row in Table 2 reports the parallelization scheme used for each application. The second row, labeled "Simulation Region," reports the number of instructions simulated for each application, and the third row, labeled "Pre-Execution Region," reports the number of instructions in the loops instrumented with pre-execution, not including the pre-execution thread instructions. The remaining rows in Table 2 are discussed later in Section 5.2.

As discussed in Section 3.2, the BAR scheme is potentially ineffective when multiple dynamic instances of the inner loop are dependent. A doubly-nested loop in Compress95 has such inter-inner loop dependences. However, this dependence occurs only occasionally, so consecutive inner loops are frequently independent. We pre-execute the outer loop in Compress95 using a backbone thread that speculatively forks a rib thread for *every* inner loop. For inner loops that are independent, the rib threads will correctly pre-execute the inner loop. For all other inner loops, the rib threads will execute incorrectly. To ensure mis-speculated rib threads terminate, we modify the rib thread loop predicates to exit after at most 4 iterations. In Compress95, we find that mis-speculated rib threads do no harm except to bring useless data into the cache. In general, the SMT should ignore exceptions signaled by such speculative threads. This support is easy to provide, but we do not model it in our simulator.

5.2 Baseline Pre-Execution Performance

We first present the performance of our pre-execution technique on the baseline SMT in Figure 7. We report execution time without pre-execution, labeled "NP," and with pre-execution, labeled

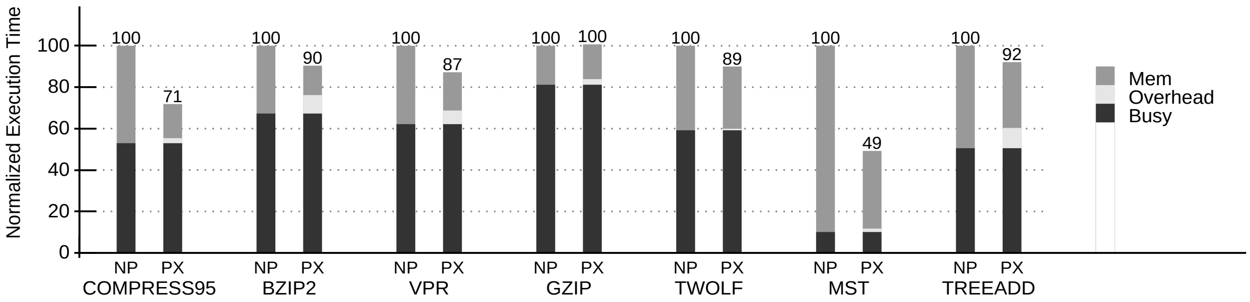

Figure 7: Normalized execution time broken down into busy, overhead, and memory stall components. The "NP" and "PX" bars show performance for no pre-execution and with pre-execution, respectively.

"PX," broken down into three components. "Busy" is the execution time without pre-execution assuming a perfect memory system (*e.g.*, all memory accesses complete in 1 cycle). "Overhead" is the incremental increase in execution time over "Busy" due to pre-execution, again on a perfect memory system. "Mem" is the incremental increase in execution time over "Busy"+"Overhead" assuming a real memory system. All times are normalized against the "NP" bars.

Figure 7 shows our pre-execution technique reduces execution time by 17.4% on average, achieving an average speedup of 27.4%. MST and Compress95 enjoy the largest gains. MST has a short pointer-chasing inner loop with a very long pointer-chasing outer loop, allowing the BAR scheme to effectively exploit memory parallelism across the outer loop. Compress95 is somewhat similar, but the inner loop instances are dependent. As described in Section 5.1, we speculatively fork rib threads to pre-execute each inner loop. Many of these speculations are correct, leading to a speedup of 40.8% for Compress95. BZIP2, VPR, and TWOLF also have very short inner loops. In VPR and TWOLF, we use the POL and BAR schemes to uncover parallelism at the outer loop level, resulting in a speedup of 12.8% across these applications. Treeadd, which performs a recursive binary tree traversal, has abundant parallelism, but the baseline SMT does not provide enough threads to exploit the parallelism. Finally, GZIP's most cache-missing inner loop is serial, and multiple instances of this loop are dependent. Unfortunately, the iterations of the next outer loop are dependent on all previous inner loops, so we cannot pre-execute it speculatively as we did in Compress95. Hence, we use the SIL scheme on the inner loop alone which does not provide any gains due to the lack of memory parallelism.

Figure 7 also shows pre-execution has very low overhead, 5.5% on average. To provide more insight, the last 4 rows in Table 2 present pre-execution instruction counts. The two rows labeled

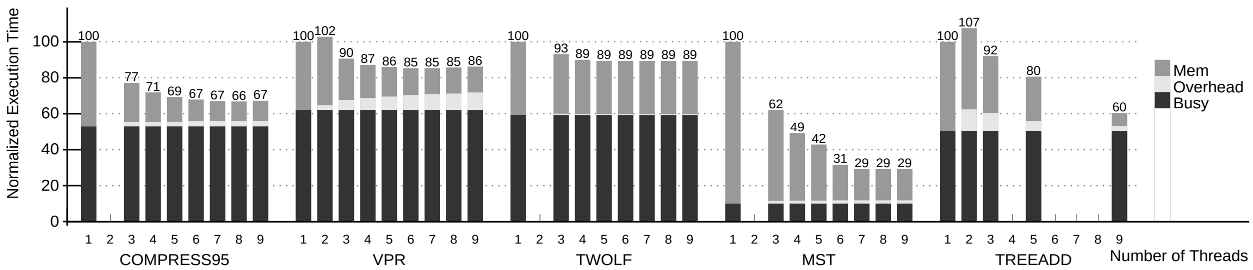

Figure 8: Varying the number of hardware contexts. Missing data points indicate the application cannot exploit the incremental increase in hardware contexts.

"Program Slice" report the number of pre-execution instructions excluding threading overhead (*e.g.*, fork and synch), and the ratio of these instructions to pre-execution region instructions. These numbers show our program slices are on average 53.1% as large as the original code. The last two rows in Table 2 report the number of total pre-execution instructions including threading overhead, and the ratio of these instructions to the pre-execution region instructions. These numbers show that pre-execution threads execute more instructions than the main thread despite the fact that the slice code is quite small, implying that software fork and synch operations have significant cost.

Although the number of pre-execution thread instructions can be significant, the overall impact on overhead is still small for two reasons. First, our applications contain very little ILP, so the baseline SMT provides enough spare issue slots for the pre-execution threads. Second, busy-wait synchronization, responsible for a large portion of the threading overhead, tends to occur when the main thread is stalled on long-latency cache misses causing the pre-execution threads to wait. Hence, busy-wait instructions tend not to significantly increase the critical path of the main thread.

5.3 Sensitivity Study

Next, we study the sensitivity of our results to two parameters: number of contexts and processor issue width. Figure 8 shows normalized execution time as the number of contexts is varied from 1 to 9, keeping all other parameters fixed. (The 1-context data points correspond to performance without pre-execution). Note the BAR scheme requires at least 2 pre-execution threads, and the PDT scheme requires a power-of-two pre-execution threads. The missing data points in Figure 8 indicate these parallelization schemes are unable to exploit the incremental increase in contexts. Also, we omit results for BZIP2 and GZIP since the SIL scheme uses only 1 pre-execution thread.

Figure 8 shows increasing the number of contexts improves the speedup to 56.7% across the

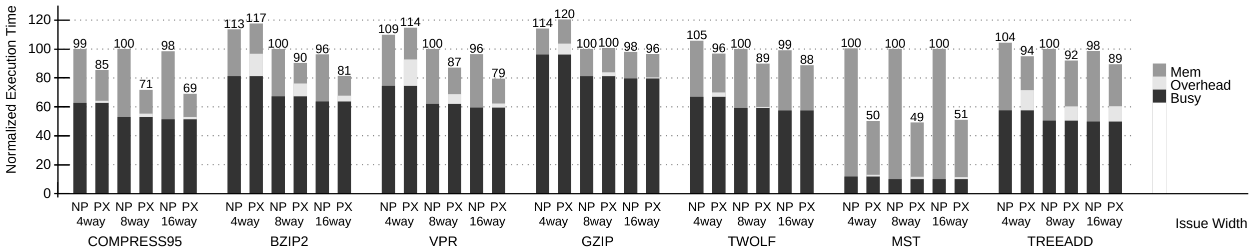

Figure 9: Normalized execution time on a 4-way, 8-way, and 16-way SMT processor. Number of hardware contexts is 4 for all simulations. All results are normalized to the baseline (8-way) configuration without pre-execution.

7 applications. The applications exhibiting the most memory parallelism, Compress95, MST, and Treeadd, benefit the most since they can overlap more memory latency given more threads.

In Figure 9, we vary the issue width between 4, 8, and 16. As the issue width is varied, we set the number of fetch, decode, and commit slots as well as the number of integer functional units to match the issue width. All other parameters are set to the baseline configuration–in particular, the number of hardware contexts is fixed at 4. For each configuration and application, Figure 9 reports the execution time with ("PX" bars) and without ("NP" bars) pre-execution. All execution times have been normalized to the "NP" bars of the baseline SMT.

Figure 9 shows two results. First, the baseline SMT is wide enough to get most of the gain. When going to a 16-way SMT, the average speedup goes up only slightly to 29.6%. Second, performance degrades when going to a narrow 4-way SMT, particularly for BZIP2, VPR, GZIP, and Treeadd. For these applications, overhead increases sharply, reflecting the fact that there are fewer spare issue slots for the pre-execution threads.

5.4 Contributions of the Techniques

Finally, we study the individual contributions of the following 3 techniques to performance: pre-execution alone, program slicing, and prefetch instructions. Figure 10 shows the result of applying each technique incrementally on the baseline SMT. The "NP" bars show execution time without any techniques. In the "P" bars, we create pre-execution code as normal, but we *put back* all the program statements that were removed by the slicer and we do not insert prefetch instructions. (Note we cannot put back any side-effect program statements since these would crash the main thread). Hence, the "P" versions benefit from pre-execution using our parallelization schemes, but the pre-execution code is not optimized with slicing or prefetch instructions. In the "PS" bars,

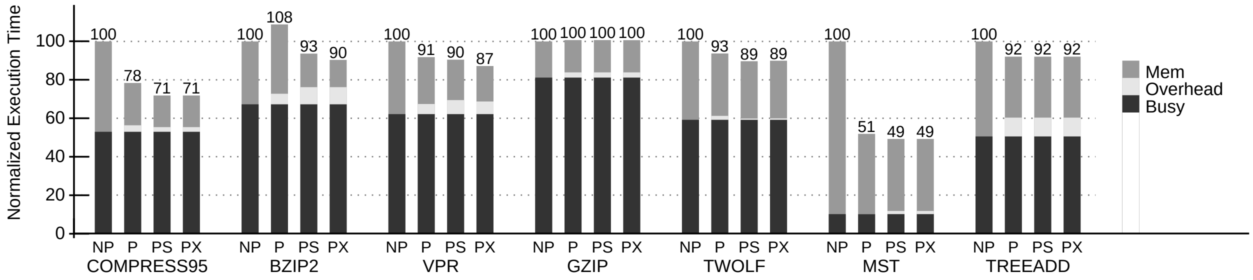

Figure 10: Contributions of different techniques. The "P" bars show pre-execution alone, the "PS" bars show pre-execution with slicing but without prefetch instructions, and the "PX" bars have all the techniques applied.

we perform program slicing on the pre-execution code, but we do not insert prefetch instructions. Lastly, the "PX" bars show execution time with all the techniques applied.

For MST and Treeadd, most or all of the gain is due to pre-execution alone. In these codes, there are plenty of issue slots due to the high frequency of memory stalls, so good performance is achieved even without program slicing. Also, there are no opportunities for inserting prefetch instructions in these codes. Consequently, the "PS" and "PX" bars are very close or identical to the "P" bars. For the other applications, pre-execution alone provides a significant part of the gain for Compress95, VPR, and TWOLF, but worsens performance for BZIP2.

Figure 10 shows program slicing is important for Compress95, BZIP2, VPR, and TWOLF. In these applications, program slicing accounts for 39.7% of the total performance gain of "PX." Interestingly, the importance of program slicing is not due to reduced instruction overhead since the reduction in the overhead components of Figure 10 are not proportional to the incremental gain between "P" and "PS" for these applications. Program slicing has another benefit: it enables the pre-execution threads to run faster because they execute less code. This enhances their ability to get ahead of the main thread and hide more memory latency.

Amongst all the techniques, prefetch instructions mattered the least. Only BZIP2, VPR, and TWOLF provided opportunities for inserting prefetch instructions at all, with BZIP2 and VPR benefiting the most. Prefetch instructions account for 25% and 22.8% of the total gain achieved by "PX" in these applications.

6 Related Work

Our work is related to four recent proposals for pre-execution. The first three, Speculative Precomputation (SP) [4], Execution-Based Prediction [17], and Data-Driven Multithreading (DDMT) [13] use instruction-level backward slices [18] to drive pre-execution. These techniques use hardware to fork and pass arguments to pre-execution threads. In addition, SP uses hardware to synchronize pre-execution threads with the main thread. In contrast, we perform thread management in software, requiring less hardware support. Also, SP proposes a hardware *chaining trigger* mechanism that forks multiple pre-execution threads along an induction variable. Our PIL, POL, and BAR parallelization schemes achieve the same thread-level parallelism, but use a software approach instead. Execution-Based Prediction and DDMT also pre-execute hard-to-predict branches in addition to cache misses. Our research focuses only on problematic memory references.

In terms of hardware support, our work is closer to Software-Controlled Pre-Execution [7], which uses software support to fork and pass arguments to pre-execution threads. However, this technique pre-executes the *original* program code instead of extracting slices. Hence, the pre-execution threads execute unnecessary instructions, and code transformations to enhance pre-execution must not disturb the correctness of the main thread (which prevents using our parallelization schemes). Also, Software-Controlled Pre-Execution still requires more hardware support than our approach because it must buffer pre-execution stores in hardware to prevent them from modifying architectural state. Since we use program slices, we can modify or remove these stores from the slice code.

In addition to these four recent proposals, there are two other related techniques. Dependence Graph Precomputation [1] not only pre-executes cache misses, but provides hardware support to extract an instruction-level backward slice from the processor's fetch queue. Slipstream Processors [14] use a speculative compute engine to automatically get ahead of the main processor for pre-execution purposes as well as for fault tolerance. Both of these techniques require more hardware support than previous pre-execution techniques, including our own proposal.

7 Conclusion

This paper proposes using program slicing to drive pre-execution. We built an experimental slicing system to extract program slices for pre-execution, and we developed several slice parallelization techniques that enable the pre-execution threads to get ahead of the main thread. Our results show pre-execution using program slices achieves a 27.4% average speedup on an 8-way SMT with 4 contexts. On an SMT with 9 contexts, the average speedup increases to 56.7%. Our results also show that increasing the issue width of the processor beyond 8-way does not provide significant additional gains. However, a 4-way SMT can suffer significant overhead due to the pre-execution thread instructions. Finally, our results show program slice parallelization is responsible for a large part of the performance gains achieved by our techniques. The removal of unnecessary instructions through program slicing is also important because it increases the speed of pre-execution threads.

We believe program slicing is an attractive technique for pre-execution. Not only does it achieve good performance, but compared to previous work, it requires the least hardware support: a small modification to a standard fetch selection policy, and basic ISA support for multithreading. In addition, program slicing is natural to integrate into existing compilers especially since program slicing tools already exist. However, further work is necessary, particularly to automate slice parallelization, before compiler integration becomes possible.

References

[1] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data Prefetching by Dependence Graph Precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001. ACM.

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

[3] Tien-Fu Chen and Jean-Loup Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Transactions on Computers*, 44(5):609–623, May 1995.

[4] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001.

[5] Joel S. Emer. Simultaneous Multithreading: Multiplying Alpha Performance. *Microproessor Forum*, October 1999.

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

[7] Chi-Keung Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001. ACM.

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

[9] James R. Lyle, Dolores R. Wallace, James R. Graham, Keith B. Gallagher, Joseph P. Poole, and David W. Binkley. Unravel: A CASE Tool to Assist Evaluation of High Integrity Software. NISTIR 5691, National Institute of Standards and Technology, August 1995.

[10] Dominik Madon, Eduardo Sanchez, and Stefan Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *Proceedings of EuroPar '99*, pages 716–726, Toulouse, France, August 1999. Springer-Verlag.

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

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

[13] Amir Roth and Gurindar S. Sohi. Speculative Data-Driven Multithreading. In *Proceedings of the 7th International Conference on High Performance Computer Architecture*, pages 191–202, January 2001.

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

[15] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995. ACM.

[16] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 1996 International Symposium on Computer Architecture*, Philadelphia, May 1996.

[17] Craig Zilles and Gurindar Sohi. Execution-Based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Goteborg, Sweden, June 2001.

[18] Craig B. Zilles and Gurindar S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 172–181, Vancouver, Canada, June 2000.