Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors

Chi-Keung Luk
VSSAD/Alpha Development Group
Compaq Computer Corporation
Chi-Keung.Luk@Compaq.com

Abstract

Hardly predictable data addresses in many irregular applications have rendered prefetching ineffective. In many cases, the only accurate way to predict these addresses is to directly execute the code that generates them. As multithreaded architectures become increasingly popular, one attractive approach is to use idle threads on these machines to perform pre-execution*—essentially a combined act of speculative address generation and prefetching—to accelerate the main thread. In this paper, we propose such a pre-execution technique for simultaneous multithreading (SMT) processors. By using* software *to control pre-execution, we are able to handle some of the most important access patterns that are typically difficult to prefetch. Compared with existing work on pre-execution, our technique is significantly simpler to implement (e.g., no integration of pre-execution results, no need of shortening programs for pre-execution, and no need of special hardware to copy register values upon thread spawns). Consequently, only minimal extensions to SMT machines are required to support our technique. Despite its simplicity, our technique offers an average speedup of 24% in a set of irregular applications, which is a 19% speedup over state-of-the-art software-controlled prefetching.*

1. Introduction

Multithreading [1, 32] and *prefetching* [8, 20, 22] are two major techniques for tolerating ever-increasing memory latency. Multithreading tolerates latency by executing instructions from another concurrent thread when the running thread encounters a cache miss. In contrast, prefetching tolerates latency by anticipating what data is needed and moving it to the cache ahead of time. Comparing multithreading against prefetching, the main advantage of multithreading is that unlike prefetching, it does not need to predict data addresses in advance which can be a serious challenge in codes with irregular access patterns [20, 26]. Prefetching, however, has a significant advantage that it can improve *single-thread* performance, unlike multithreading which requires multiple concurrent threads. In this paper, we propose a technique which exploits each approach's own advantages to complement the other. More specifically, our technique accelerates single threads running on a multithreaded processor by spawning helper threads to perform *pre-execution*, a generalized form of prefetching which also automatically generates data addresses, on behalf of the main thread.

Pre-execution is generally referred to as the approach that tolerates long-latency operations by initiating them early and speculatively. There are a number of ways to apply pre-execution, including prefetching data and/or instructions [12, 26], pre-computing branch outcomes [15], and pre-computing general execution results [28, 31]. For our purpose, pre-execution is mainly used as a vehicle for speculatively generating data addresses and prefetching—the ultimate computational results are simply ignored. Moreover, unlike several recent pre-execution techniques [4, 5, 9, 26, 28, 31, 36] which pre-execute a shortened version of the program, our technique simply works on the *original* program and hence requires no mechanism to trim the program. In essence, our technique tolerates latency by exploiting a new dimension of pre-execution—running ahead *multiple* data streams simultaneously.

Similar to prefetching, pre-execution can be controlled either by hardware or software. Hardware-based schemes typically look for particular events (e.g., cache misses) to trigger pre-execution, and software schemes rely on the programmer or the compiler to insert explicit instructions for controlling pre-execution. While the hardware-based approach does not pose any instruction overhead, the software-based approach has the major advantage of being able to exploit application-specific knowledge about future access patterns. Since we are most interested in applications whose cache misses are caused by *irregular* data accesses, our focus in this study is on a *software-controlled* pre-execution mechanism.

Among previously proposed multithreaded architectures, a *simultaneous multithreading* processor (SMT) [32] is chosen as the platform for this study. An SMT machine allows multiple independent threads to execute simultaneously (i.e. in the same cycle) in different functional units. For example, the Alpha 21464 [13] will be an SMT machine with four threads that can issue up to eight instructions per cycle from one or more threads. Although pre-execution could be applied to other multithreaded architectures as well, the SMT architecture does offer a unique advantage that resource sharing between the main thread and pre-execution threads can be promptly adjusted to favor the ones that would increase overall performance. For instance, if the main thread has been stalled for cache misses in recent cycles, execution resources would be given up from the main thread to pre-execution threads, thereby allowing them to tolerate more misses.

1.1. Objectives of This Study

This paper makes the following contributions. First, we investigate how pre-execution can be exploited to generate addresses of irregular data accesses early. Through examining a collection of benchmarks, we propose a number of software-controlled pre-execution schemes, each of which is designated for an important class of access patterns. These schemes would be helpful to programmers for writing memory-friendly programs using pre-execution threads, and to the compiler for inserting pre-

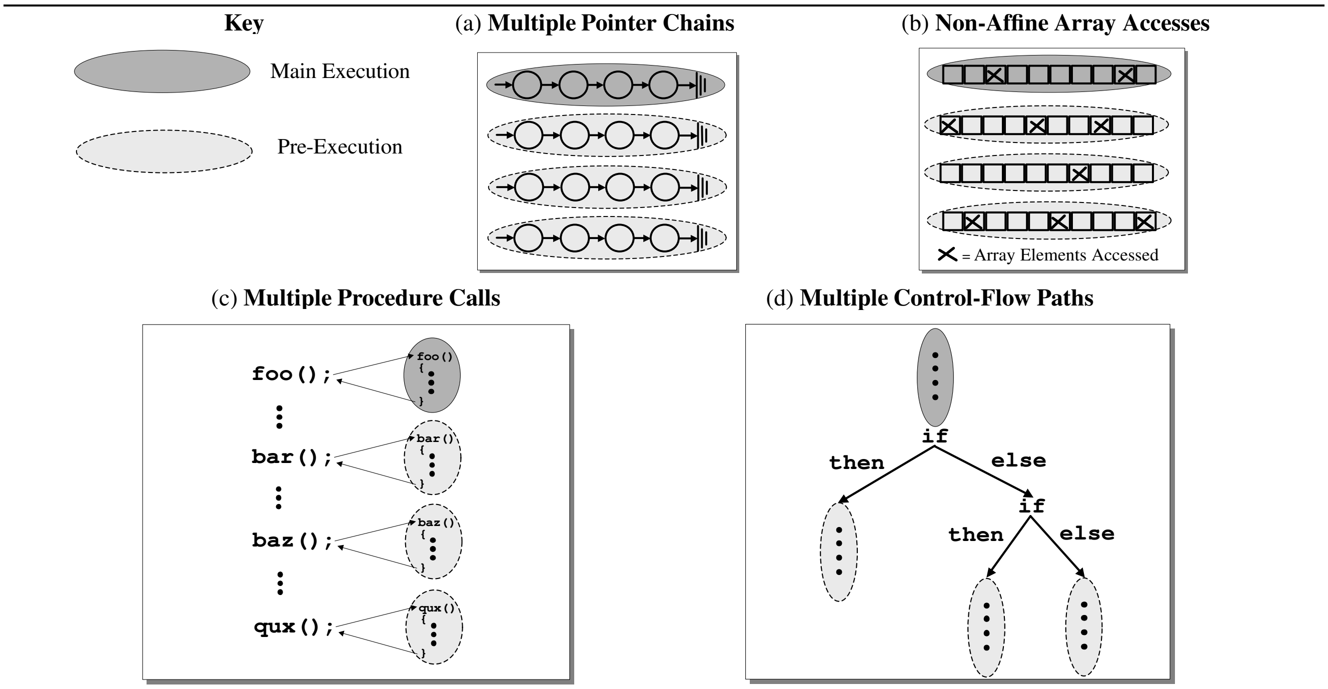


Figure 1. Illustration of four applications of software-controlled pre-execution for tolerating memory latency. The main thread is darkly shaded while pre-execution threads are lightly shaded.

execution automatically. Second, we discuss the hardware and software extensions required for an SMT processor to support pre-execution. Third, we quantitatively evaluate the performance of our pre-execution technique and compare it against the best existing software-controlled prefetching schemes. Our results demonstrate that pre-execution significantly outperforms prefetching in a set of irregular applications, offering at least 30% speedups over the base case in over half of our applications. Finally, by exploring the design space of our technique through experimentation, we show that the performance benefits of our technique can be mostly realized with only very minimal extensions to SMT processors.

2. Software-Controlled Pre-Execution

We now discuss the basic concepts behind software-controlled pre-execution, followed by a number of applications of this mechanism that we found in some commonly used benchmarks.

2.1. Basic Concepts

Software-controlled pre-execution allows the programmer or compiler to initiate helper threads to run ahead of the main thread into parts of the code that are likely to incur cache misses. Thus, the very first thing is to decide where to launch pre-execution in the program, based on the programmer's knowledge, cache miss profiling [23], or compiler locality analysis [24]. Once this decision has been made, new instructions for spawning pre-execution threads are inserted at the right places in the program. Each thread-spawning instruction requests for an idle hardware context to pre-execute the code sequence starting at a given PC. If there is no hardware context available, the pre-execution request will simply be dropped. Otherwise, a pre-execution thread, say T, will be successfully spawned with its initial register state copied from that of the parent thread. After this copying is done, T will start running at the given PC in a so-called *pre-execution mode*. Instructions are executed as normal under this mode except that (i) all exceptions generated are ignored and (ii) stores are not committed into the cache and memory so that speculative actions that happened during pre-execution will not affect the correctness of the main execution. Finally, T will stop either at a pre-determined PC or when a sufficient number of instructions have been pre-executed. At this point, T will free its hardware context and the results held in T's registers are simply discarded (i.e. they will not be integrated back to the main execution). Further details of this mechanism will be given later in Section 3.

2.2. Applications of Pre-Execution

To study how software-controlled pre-execution can be employed to generate data addresses and prefetch well ahead of the main execution, we examine a set of applications drawn from four common benchmark suites (SPEC2000 [16], SPEC95 [10], SPLASH-2 [34], and Olden [25]). A large number of cache misses in these applications are due to relatively irregular access patterns involving pointers, hash tables, indirect array references, or a mix of them, which are typically difficult for prefetching to handle. We categorize these access patterns and suggest pre-execution schemes for them. In the rest of this section, we describe these schemes using our benchmarks as examples.

2.2.1. Pre-Executing Multiple Pointer Chains

A well-known challenge in prefetching pointer-based codes is the *pointer-chasing problem* [20] which depicts the situation where the address of the next node we want to prefetch is not known until we finish with the current load. To tackle this problem, prefetching techniques based on jump pointers [20, 27] have been proposed to record the address of the node that we would like to prefetch at the current node according to past traversals. These techniques would tolerate the latency of accessing a *single* chain if the traversal order is fairly static over time and the overhead of maintaining those jump pointers does not overwhelm the benefit.

Recently, a technique called *multi-chain prefetching* [19] has been proposed to attack the pointer-chasing problem from a different angle. Instead of prefetching pointer chains one at a time, this technique prefetches *multiple* pointer chains that will be visited soon while traversing the current one, whereby the latency of accessing future chains can be overlapped with that of accessing the current one. This technique has the advantage over jump-pointer based schemes that it neither relies on a repeating traversal order nor needs to maintain extra pointers. Of course, the key to the success of multi-chain prefetching is to have enough independent chains that can be traversed in parallel. Fortunately, such chains are not uncommon as we do discover them in three of our applications: `mcf`, `mst`, and `em3d`. Given these chains, we need some mechanism to visit them simultaneously. The original multi-chain prefetching [19] uses a special hardware prefetch engine for this purpose. In our case, this is a natural application of pre-execution: We simply spawn one helper thread to pre-execute each pointer chain. Thus, if there are a total of N hardware contexts supported by the machine, we can potentially pre-execute $N-1$ chains in parallel (one thread is always used by the main execution). An illustration of this scheme can be found in Figure 1(a). To make our discussion more concrete, let us look at two benchmarks in detail.

The Spec2000 benchmark `mcf` spends roughly half of its execution time in a nested loop which traverses a set of linked lists. An abstract version of this loop is shown in Figure 2(a), in which the for-loop iterates over the lists and the while-loop visits the elements of each list. As we observe from the figure, the first node of each list is assigned by dereferencing the pointer `first_of_sparse_list`, whose value is in fact determined by `arcout`, an induction variable of the for-loop. Therefore, even when we are still working on the current list, the first and the remaining nodes on the next list can be loaded speculatively by pre-executing the next iteration of the for-loop.

Figure 2(b) shows a version of the program with pre-execution code inserted (shown in boldface). **END_FOR** is simply a label to denote the place where `arcout` gets updated. The new instruction **PreExecute_Start(END_FOR)** initiates a pre-execution thread, say T, starting at the PC represented by **END_FOR**. Right after the pre-execution begins, T's registers that hold the values of `i` and `arcout` will be updated. Then `i`'s value is compared against `trips` to see if we have reached the end of the for-loop. If so, thread T will exit the for-loop and encounters a **PreExecute_Stop()**, which will terminate the pre-execution and free up T for future use. Otherwise, T will continue pre-executing the body of the for-loop, and hence compute the first node of the next list automatically. Finally, after traversing the entire list through the while-loop, the pre-execution will be terminated by another **PreExecute_Stop()**. Notice that any **PreExecute_Start()** instructions encountered during pre-execution are simply ignored as we do not allow nested pre-execution in order to keep our design simple. Similarly, **PreExecute_Stop()** instructions cannot terminate the main thread either.

In this application, we pre-execute only one list at a time because there are a sufficient number of nodes on each list to pre-execute to hide the latency. However, that would not be the case if the lists are short, as we would expect in chaining-based hash tables. An example of this scenario is found in the application `mst`, as described below.

(a) Original Code

```
register int i;
register arc_t *arcout;
for(; i<trips; ){
   // loop over 'trips" lists
   if (arcout[1].ident != FIXED) {
      ...
      first_of_sparse_list = arcout + 1;
   }
   ...
   arcin = (arc_t *)first_of_sparse_list
                      →tail→mark;
   // traverse the list starting with
   // the first node just assigned
   while (arcin) {
      tail = arcin→tail;
      ...
      arcin = (arc_t *)tail→mark;
   }
   i++, arcout+=3;
}
```

(b) Code with Pre-Execution

```
register int i;
register arc_t *arcout;
for(; i<trips; ){
   // loop over 'trips" lists
   if (arcout[1].ident != FIXED) {
      ...
      first_of_sparse_list = arcout + 1;
   }
   ...
   // invoke a pre-execution starting
   // at END_FOR
   PreExecute_Start(END_FOR);
   arcin = (arc_t *)first_of_sparse_list
          →tail→mark;
   // traverse the list starting with
   // the first node just assigned
   while (arcin) {
      tail = arcin→tail;
      ...
      arcin = (arc_t *)tail→mark;
   }
   // terminate this pre-execution after
   // prefetching the entire list
   PreExecute_Stop();
END_FOR:
   // the target address of the pre-
   // execution
   i++, arcout+=3;
}
// terminate this pre-execution if we
// have passed the end of the for-loop
PreExecute_Stop();
```

Figure 2. Abstract versions of an important loop nest in the Spec2000 benchmark `mcf`. Loads that incur many cache misses are underlined.

The Olden benchmark `mst` makes intensive use of hashing; an abstract version of the hashing-related code is shown in Figure 3(a). The function `HashLookup()` is called by a list of hash tables to perform hashing on each of them. Chaining-based hashing, as the one performed by `HashLookup()`, is known to be challenging to prefetching [20, 27] for two reasons. First, the bucket to be hashed to should be fairly random if the hashing function is well designed. Thus, predicting the bucket address itself tends to be difficult. Second, a good hashing function will also avoid collisions and therefore the average chain length at each bucket should be very small. Thus, jump-pointer based prefetching techniques also would not be effective.

Fortunately, pre-execution offers a unified solution to both problems. By computing N hashing functions at the same time (one through the main execution and $N-1$ through pre-execution), we can potentially reduce memory stall by a factor of N. Figure 3(b) demonstrates such a case with $N = 3$. To facilitate pre-execution, the for-loop in function `BlueRule()` is unrolled twice. Doing this allows the next two iterations of the for-loop to explicitly exist in the code so that they can be used as the targets of the two **PreExecute_Start()**'s. Notice that both `next_1` and `next_2` are assigned twice—once for the pre-execution and once for the main execution. We need this seemingly redundant computation since the values of `tmp→next` and `tmp→next→next` may be modified in between the pre-execution and the main execution. Pre-execution is terminated either after `HashLookup()` is done or after the for-loop exits.

(a) Original Code

```
void BlueRule(Vertex ins,
              Vertex vlist) {
  ...
  for (tmp=vlist→next; tmp;
       tmp=tmp→next) {
       ...
       // look up hash table
       // tmp→edgehash
       dist = (int) HashLookup(ins,
                     tmp→edgehash);
       ...
  } //end-for
}

void *HashLookup(unsigned key,
                 Hash hash) {
  ...
  // hash to the jth bucket
  j = (hash→mapfunc)(key);
  // assign the chain's head to ent
  ent = hash→array[j];
  if (ent) {
     ...
     for ( ;ent→key!=key; ) {
        // search for the key over
        // over the list, which is
        // very SHORT
        ent = ent→next;
        ...
     }
  }
  ...
}
```

(b) Unrolled Code with Pre-Execution

```
void BlueRule(Vertex ins,
              Vertex vlist) {
  ...
  for (tmp=vlist→next; tmp; ) {
     Vertex next_1, next_2;
     next_1 = tmp→next;
     // invoke a pre-execution
     // starting at UNROLL_1
     PreExecute_Start(UNROLL_1);
     if (next_1) {
        next_2 = next_1→next;
        // invoke a pre-execution
        // starting at UNROLL_2
        PreExecute_Start(UNROLL_2);
     }
     ...
     // look up hash table tmp→edgehash
     dist = (int) HashLookup(ins,
                   tmp→edgehash);
     ...
     next_1 = tmp→next;
UNROLL_1: // 1st unrolled
     if (next_1) {
        ...
        // look up hash table
        // next_1→edgehash
        dist = (int) HashLookup(ins,
                      next_1→edgehash);
        // terminate pre-execution
        // after hashing
        PreExecute_Stop();
        ...
     } else
        break;
     next_2 = next_1→next;
UNROLL_2: // 2nd unrolled
     if (next_2) {
        ...
        // look up hash table
        next_2→edgehash
        dist = (int) HashLookup(ins,
                      next_2→edgehash);
        // terminate pre-execution
        // after hashing
        PreExecute_Stop();
        ...
     } else
        break;
     tmp = next_2→next;
  } //end-for
  // terminate any pre-execution
  // passing the end of the loop
  PreExecute_Stop();
}

void *HashLookup(unsigned key,
                 Hash hash) {
  ///// IDENTICAL TO ORIGINAL ////
}
```

Figure 3. Abstract versions of the hashing component in `mst`. Loads that incur many cache misses are underlined.

2.2.2. Pre-Executing Loops Involving Difficult-to-Prefetch Array References

Following the fashion that we pre-execute pointer chains, we can also pre-execute array references across multiple loop iterations (see Figure 1(b) for an illustration). In particular, we are interested in cases that present challenges for the compiler to prefetch. Such cases typically include array references in loops with control-flow as well as references with strides that are not compile-time constants. An example of the latter case is *indirect* array references, which are quite common in scientific and engineering applications such as sparse-matrix algorithms and wind-tunnel simulations. Another common example is *arrays of pointers*. To cope with these cases, the compiler usually needs to heuristically decide how to prefetch, perhaps based on profiling information [22]. In contrast, these cases do not present a problem to pre-execution as it directly runs the code and hence does not need to make any compile-time assumptions.

Among our benchmarks, we find that indirect array references contribute significantly to the cache misses in the Spec2000 application `equake`. On the other hand, arrays of pointers cause many misses in `raytrace`, a Splash-2 application. We apply pre-execution to them and compare it against the best-known compiler algorithm for prefetching indirect references [22]. The results will be presented later in Section 5.2.

2.2.3. Pre-Executing Multiple Procedure Calls

So far, we have been focusing on pre-executing loop iterations (either for pointer dereferences or array references). A straightforward extension is to pre-execute at the level of *procedures*, as pictured in Figure 1(c). This would be particularly helpful in the case where a number of procedures are used to access different data structures (or different parts of a large structure). For example, in the classical binary-tree traversal through recursion, one could pre-execute the right subtree while the main execution is still working on the left one.

We apply this rather simple pre-execution scheme to the Spec2000 benchmarks `twolf`. Most cache misses occur in the procedure `ucxx2()`, which invokes a few other procedures to process various data structures. The ways that these structures are accessed in each of these procedures are extremely complicated: They involve linked lists, arrays, multi-level pointer dereferencing, and complex control flow. Thus, it is very challenging to add prefetches within these procedures. Fortunately, by pre-executing these procedures simultaneously, the latency of touching those complicated structures can be successfully hidden without caring much how they are actually accessed. The detailed results will be shown in Section 5.1.

2.2.4. Pre-Executing Multiple Control-Flow Paths

Our final pre-execution scheme targets the situation where the address of the next data reference (which is likely a cache miss) depends on which control-flow path we are going to take out of multiple possibilities. Instead of waiting until the correct path is known, we can pre-execute all possible paths and hence prefetch the data references on each of them (see the illustration in Figure 1(d)). After the correct path is determined, we can then cancel all wrong-path pre-execution and only keep the one that is on the right path to allow it running ahead of the main execution. A similar idea of executing multiple paths at once [18, 33] has been exploited before to reduce the impact of *branch misprediction*.

Let us illustrate this scheme using the Spec95 benchmark `compress`. In this application, the function `compress()` reads a series of characters from the file being compressed. Each character read is applied a *varying* hash function to form an index to the table `htab`, and most cache misses in this benchmark are caused by looking up `htab`. When we look up `htab` for the current character, there are three possible outcomes: no match, a single match, or a collision which needs a secondary hash. The interesting point here is that the hash function that will be applied to the

$Thread\_ID$ = **PreExecute_Start(**$Start\_PC$**,** Max_Insts**):** Request for an idle context to start pre-execution at $Start\_PC$ and stop when Max_Insts instructions have been executed; $Thread\_ID$ holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction has effect only if it is executed by the main thread.

PreExecute_Stop(): The thread that executes this instruction will be self terminated if it is a pre-execution thread; no effect otherwise.

PreExecute_Cancel($Thread\_ID$**):** Terminate the pre-execution thread with $Thread\_ID$. This instruction has effect only if it is executed by the main thread.

Figure 4. Proposed instruction set extensions to support pre-execution. (C syntax is used to improve readability.)

next character in turn depends on the outcome of the *current* table lookup. Because of this dependency, prefetching has shown little success in this benchmark. Nevertheless, we could potentially pre-execute all the three paths that correspond to the three possible outcomes *before* performing the current table lookup. One of these pre-executions would correctly pre-compute the next hash function and perform the next table lookup in advance, while the other two pre-executions will be eventually canceled after the actual outcome of the current table lookup is known.

In summary, pre-execution can be a powerful weapon to attack the latency problem. Its ability to automatically generate addresses and prefetch, regardless of the complexity of the data access patterns and control flow, could allow it to overcome some of the challenges faced by existing prefetching techniques. The pre-execution schemes we discuss in this section are by no means exhaustive but can handle some of the most important access patterns. Given that pre-execution is so appealing, we address the issues of implementing it in an SMT machine in the following section.

3. Implementation Issues

We now discuss the support that we need from the instruction set, the hardware, and the software to implement software-controlled pre-execution on an SMT machine.

3.1. Extensions to the Instruction Set Architecture

To exploit pre-execution, the machine must have some way to control pre-execution—i.e. specify when and where to start and terminate pre-execution. Rather than taking a purely hardware-based approach as done in some other studies [12, 31], we propose extending the underlying instruction set architecture (ISA) by adding a few instructions which allow software to control pre-execution directly. It is interesting to note that this approach is analogous to *software-controlled prefetching* [21, 22]. The advantages of this approach are its programmability and flexibility. In addition, we expect the software overhead to be low given today's wide-issue superscalar processors, which is generally quite true for software-controlled prefetching.

Figure 4 shows our proposed ISA extensions, which consist of three new instructions with zero to three register operands each. **PreExecute_Start** spawns a pre-execution thread at a particular PC and stops when certain number of instructions have been executed in the pre-execution (Note: We have assumed a large Max_Insts in the **PreExecute_Start** shown earlier in Figures 2 and 3.) It will return the identity of the spawned thread or -1 if there is no context available. **PreExecute_Stop** explicitly terminates a thread itself if it is in the pre-execution mode while **PreExecute_Cancel** terminates the pre-execution running on *another* thread. To simplify the design, only the main thread (i.e. not a pre-execution thread) is allowed to spawn and terminate a pre-execution thread. In other words, *nested* pre-execution is *not* supported in the current design. Also note that the ISA can be further extended to have different favors of control over pre-execution. For example, one could imagine another **PreExecute_Start** instruction with stopping conditions like maximum number of loads, maximum number of cache misses, etc.

3.2. Inserting Pre-Execution into the Application

Once we are provided with the extended instructions for controlling pre-execution, the next step is to insert them into the application. They can be inserted either by the original programmer while the program is being developed or by applying the compiler to existing programs.

In the first approach, since the programmer is already familiar with the program, she/he can potentially insert pre-execution without explicitly performing the analysis that would be done by the compiler. To facilitate the insertion process, we can have an application programming interface (API) for manipulating pre-execution threads. In fact, a similar interface called *PThreads* [6] has long been used to exploit *parallelism* using threads. By providing a new API (perhaps we can call it *MThreads*) or extending the existing PThreads to support pre-execution threads, programmers can use threads to address both the memory latency and parallelism problems. Also, the examples given in Section 2.2 would help programmers design their own pre-execution schemes.

In the second approach, the compiler (with assistances from profiling or program annotations) is responsible for inserting pre-execution. It needs to perform the algorithm shown in the appendix, which is designed in the light of existing prefetching algorithms [20, 22]. The first three steps constitute a *locality analysis* phase which determines which references are likely to cause cache misses and could benefit from pre-execution. The remaining three steps work together as a *scheduling* phase which calculates the pre-execution distance and performs all necessary code transformations.

3.3. Duplication of Register State

When a pre-execution thread is spawned, its register state needs to be initialized with that of the parent thread. Such register-state duplication is also needed in many other related techniques like threaded multi-path execution [33], speculative data-driven multithreading [28], and thread-level data speculation [30], etc. For these techniques, the *entire* register state has to be copied as *fast* as possible. Consequently, special hardware mechanisms have been proposed to accomplish this task. In our case, although we can also make use of these mechanisms, we observe that *software-based* mechanisms may already be sufficient if the startup overhead is small relative to the runtime of the pre-execution thread. We now consider both hardware and software-based duplication in detail.

3.3.1. Hardware-Based Duplication

A mechanism called *mapping synchronization bus* (MSB) has been proposed to copy register state across threads for threaded multi-path execution (TME) [33]. In this approach, it is actually the *register map* (instead of register values themselves) that gets copied across threads. While the MSB correctly synchronizes the register maps *prior to* a thread spawn, there is one complication afterward. Since a physical register can be shared by both threads, it is possible that one thread will free the register while it is still being used by the other thread. One solution is to commit *no* instructions in the pre-execution thread and hence it will never free any physical registers. Although this approach works well for TME, it may not be adequate in our case since this would limit the lifetime of our pre-execution threads (due to running out of physical registers), which tends to be much longer than that of TME threads (because a pre-execution thread would spend the time to cover multiple cache misses while a TME thread is done once the branch is resolved). Another solution which better fits our need is using *reference count*. Here, we associate a reference counter to each physical register to count the number of threads that the register is in use. If a physical register becomes shared by an additional thread due to a thread spawn, its reference counter is incremented by one. This counter will be decremented when the physical register is freed by a thread. Eventually, the last thread that frees the register will return it to the pool of free physical registers for future use.

3.3.2. Software-Based Duplication

An alternative to employing special hardware is using software to perform the duplication. Essentially, we can write a small routine which first saves the register values of one thread to memory and then retrieves them back from memory into the registers of the other thread. The obvious advantage of this approach is its simplicity—virtually no special hardware is required. On the other hand, its potential drawback is a longer thread-spawn latency. Fortunately, the following three observations imply that a software-based approach may be viable.

First, the memory region for passing the register values is small (only 256 bytes for 32 64-bit logical registers of the main thread) relative to the cache size. In addition, it is frequently referenced in the part of the code that invokes pre-execution. Therefore, this region should be found in the cache most of the time while it is being used. Second, since our pre-execution mechanism targets hiding miss latency, it can potentially tolerate a longer thread-spawn latency. For instance, if a pre-execution thread covers 10 L2 misses which take 100 cycles each, then adding 50 cycles for copying register values delays the pre-execution by only 5%. Third, in many cases, it is not necessarily to pass the entire register state. One generally useful heuristic is not to copy floating-point registers at all. In addition, software can decide to copy only the subset of registers that are relevant to data-address generation and the surrounding control flow. For example, for the pre-execution code of `mcf` shown in Figure 2(b), only six registers (those holding `i`, `arcout`, `trips`, and `first_of_sparse_list`, plus the stack and global pointers) needed to be copied from the main thread to the pre-execution thread.

To determine how fast the duplication needed to be done, we experimented with a wide range of thread-spawn latencies. It is encouraging that performance of pre-execution is fairly insensitive to thread-spawn latency. Thus, we assumed a software-based thread-spawn latency (32 cycles) in our baseline experiments. The detailed results will be shown later in Section 5.3.2.

Table 1. Simulation parameters.

Pipeline Parameters	
Number of Hardware Contexts	4
Fetch/Decode/Issue/Commit Width	8
Instruction Queue	128 entries
Functional Units	8 integer, 6 floating-point; latencies are based on the Alpha 21264 [17]
Branch Predictor	A McFarling-style choosing branch predictor like the one in the Alpha 21264 [17]
Thread Prioritization Policy	A modified ICOUNT scheme [32] which favors the main thread
Memory Parameters	
Line Size	32 bytes
I-Cache	32KB, 4-way set-associative
D-Cache	32KB, 4-way set-associative
Miss Handlers (MSHRs)	64 total for data and inst.
Unified L2-Cache	1MB, 8-way set-associative
Primary-to-Secondary Miss Latency	12 cycles (plus any delays due to contention)
Primary-to-Memory Miss Latency	72 cycles (plus any delays due to contention)
Primary-to-Secondary Bandwidth	32 bytes/cycle
Secondary-to-Memory Bandwidth	8 bytes/cycle

3.4. Handling Speculative Results

Due to the speculative nature of pre-execution, it can generate incorrect results. There are three ways that these results could affect the correctness of the main thread. We now consider how each of them can be handled:

Register values: They are automatically taken care of by the underlying SMT architecture. Since each thread has its own set of logical registers, any incorrect computational results produced during pre-execution are only locally visible and cannot affect the main execution.

Exceptions: We can simply ignore all exceptions such as invalid load addresses, division by zero, etc. generated under the pre-execution mode. Thus, the main thread will not notice any additional exceptions.

Stores: There are two possible approaches here. The first approach is to simply discard all stores under the pre-execution mode. The second approach, which is what we assumed in our baseline machine, uses a *scratchpad* to buffer the effect of stores during pre-execution. Instead of writing into the cache and memory, a pre-executed store will write into the scratchpad. And a pre-executed load will look up data in both the cache and the scratchpad. The main advantage of this approach is that pre-executed loads can observe the effect of earlier stores from the same thread, which may be important to generating future data addresses or maintaining the correct control flow. For instance, if a procedure with a pointer-type argument (e.g., the head of a linked list) is called during pre-execution, both the pointer value and the return address could be passed through the stack. Thus, it would be desirable to be able to read both values back from the stack in the procedure so that the correct data item can be fetched and the procedure can eventually return to its caller as well.

Table 2. Application characteristics. Each application was simulated 100M instructions after skipping "Insts Skipped" instructions.

Name	Description	Source	Input Data Set	Pre-Execution Scheme	Insts. Skipped
Compress	Compresses and decompresses file in memory	Spec95 [10]	Reference	Multiple control-flow paths	2591M
Em3d	Simulates the propagation of electromagnetic waves in a 3D object	Olden [25]	2K nodes	Multiple pointer chains	364M
Equake	Simulates the propagation of elastic waves in large, highly heterogeneous valleys	Spec00 [16]	Training	Multiple array references	1066M
Mcf	Schedules single-depot vehicles in public mass transportation	Spec00	Training	Multiple pointer chains	3153M
Mst	Finds the minimum spanning tree of a graph	Olden	2K nodes	Multiple pointer chains	1364M
Raytrace	Ray-tracing program	Splash-2 [34]	Car	Multiple array references	1680M
Twolf	Determines the placement and global connections for standard cells	Spec00	Training	Multiple procedure calls	406M

3.5. Termination of Pre-Execution

We have already discussed in Section 3.1 that the application itself can terminate pre-execution via **PreExecute_Stop** or **PreExecute_Cancel**. In addition, there are a few *system-enforced* terminating conditions for preserving correctness or avoiding wasteful computation.

To preserve correctness, a pre-execution thread must be terminated if its next PC is out of the acceptable range imposed by the operating system. Moreover, to make sure that a pre-execution thread will be eventually terminated, there is a default limit on the number of instructions that a pre-execution thread can execute. Once this limit is reached, the thread will be terminated anyway.

To avoid wasteful computation, we have an option that enables the hardware to terminate a pre-execution thread if it is already caught up by the main thread. Accurate detection of when a pre-execution thread and the main thread meet (if it ever happens) is challenging since they can follow totally different control-flow paths. One heuristic is to have the hardware keep track of the starting PC, say P, and the number of instructions pre-executed so far, say N, for each pre-execution thread. If at some point the main thread has executed N instructions after passing P, then we assume that the main thread and the pre-execution thread meet. While this heuristic is not perfect, we find that it is simple to implement and yet quite useful in practice.

4. Experimental Framework

To evaluate the performance benefits of software-controlled pre-execution, we modeled it in an SMT processor and applied it to a collection of *irregular* applications. We added our proposed ISA extensions to the underlying Alpha ISA by making use of a few unused opcodes. Since the compiler support for automatically inserting pre-execution is still under development, we inserted it manually in this study, following the algorithm shown in the appendix.

We performed detailed cycle-by-cycle simulations of our applications on an out-of-order SMT processor. Our simulator called *Asim* [3] is newly developed for evaluating future Alpha systems. It models the rich details of the processor including the pipeline, register renaming, instruction queue, branch prediction, the memory hierarchy (including tag, bank, and bus contention), and the additional SMT support (including per-thread PC's and return stacks, thread prioritization, etc). Table 1 shows the parameters used in our model for the bulk of our experiments. In addition, since Asim is entirely execution-driven, it can faithfully model the effects down any speculative paths (including both pre-execution paths and predicted paths).

Our applications are shown in Table 2. To avoid lengthy simulations, we first identified a representative region of each application. We then skipped the simulation until that region was reached and ran the simulation for 100M instructions. Since extra instructions were added to the pre-executed version of the application, special care was taken to ensure that it was the same 100M instructions being simulated in the original and pre-executed versions.[1] The applications were compiled using the standard Compaq C compiler version 6.1 with -O2 optimizations under Tru64 Unix 4.0.

5. Experimental Results

We now present results from our simulation studies. We start by evaluating the performance of software-controlled pre-execution. Next, we compare this with the best-known software prefetching techniques for individual applications. Finally, we explore the performance impact of architectural support.

5.1. Performance of Pre-Execution

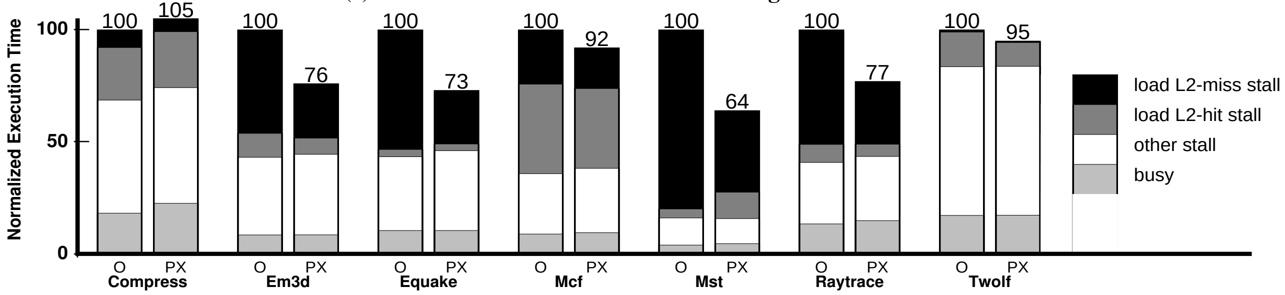
Figure 5 shows the results of our first set of experiments in which a thread-spawn latency of 32 cycles and a per-thread scratchpad of 64 entries were assumed. We will study the performance impact of these two parameters later in Section 5.3.

Figure 5(a) shows the overall performance improvement on the *main execution* offered by pre-execution, where the two bars correspond to the cases without pre-execution (**O**) and with pre-execution (**PX**). These bars represent execution time normalized to the case without pre-execution, and they are broken down into four categories explaining what happened during all potential graduation slots. As we see in the figure, pre-execution offers speedups ranging from 5% to 56% in six out of the seven applications. These improvements are the result of significant reduction in the total load-miss stall (i.e. sum of the top two sections in Figure 5(a)), with four applications (`em3d`, `equake`, `mst`, and `raytrace`) enjoying roughly 50% reduction. Turning our attention to the costs of pre-execution, there are two kinds of pre-execution overhead: (i) the resource sharing between the main execution and pre-execution, and (ii) the additional instructions for controlling pre-execution. Fortunately, Figure 5(a) shows that `compress` is the only case where pre-execution overhead more than offset the reduction in memory stalls; for the other six applications pre-execution overhead increases the sum of the *busy* and *other stall* sections by at most 4%.

Figure 5(b) tabulates a number of statistics specific to pre-execution. The second column is the total number of pre-execution requests, and the third column shows how many of these requests were able to find idle hardware contexts. Overall, at least 60% of these requests were satisfied. They are further divided into three

[1] We first marked some important program points in the region and counted how many times these markers were passed in the original version. We then simulated the same number of markers in the pre-executed version.

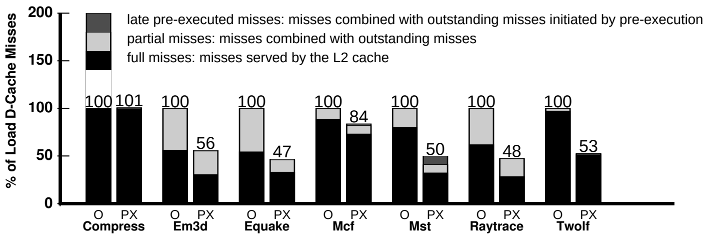
(a) **Execution Time Normalized to the Original Case**



(b) **Performance Details of Pre-Execution**

Application	Pre-Execution Requests	Actual Pre-Executions	Leading Pre-Executions		Lagging Pre-Executions		Wrong-Path Pre-Executions
			Number	Average Lead Time (cycles)	Number	Average Lag Time (cycles)	
Compress	2972214	2483355	403849	58	699982	18	1379524
Em3d	40000	40000	40000	1796	0	0	0
Equake	36470	30423	30423	2139	0	0	0
Mcf	8596	6353	6100	2782	253	11	0
Mst	390377	378992	378560	578	429	9	3
Raytrace	1000000	613929	613928	246	0	0	1
Twolf	68537	55969	41852	680	0	0	14117

(c) **Load D-Cache Misses**



(d) **Traffic between the D-Cache and L2-Cache**

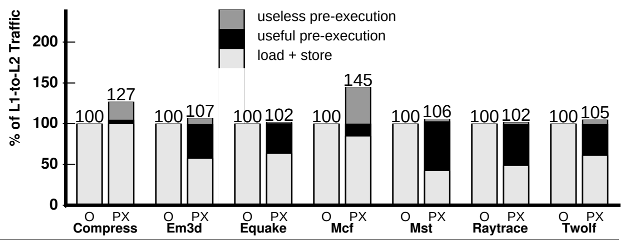


Figure 5. Performance of software-controlled pre-execution (O = original, PX = pre-executed)

categories. "Leading Pre-Executions" are those that actually led the main execution (by the amount shown under "Average Lead Time"). On the contrary, "Lagging Pre-Executions" are those that had not been started by the time that the main execution reached the starting PC of the pre-execution (the lagging amount is shown under "Average Lag Time"). Finally, the rest are "Wrong-Path Pre-Executions" which instead took a path different from the main execution.

To understand the performance results in greater depth, we present two additional performance metrics. Figure 5(c) shows the number of load D-cache misses in the original and pre-execution cases, which are divided into three categories. A *partial miss* is a D-cache miss that combines with an outstanding miss to the same line, and therefore does not necessarily suffer the full miss latency. A *full miss*, on the other hand, does not combine with any access and therefore suffers the full latency. A *late pre-executed miss* is a D-cache miss that combines with an outstanding miss generated by pre-execution (i.e. the pre-execution was launched too late). If pre-execution has perfect *miss coverage*, all of the full and partial misses would have been converted into hits (which do not appear in the figure) or at least into late pre-executed misses. We observe from Figure 5(c) that, except in `compress`, pre-execution reduces the number of load misses in the main execution by 16% to 53%. For `compress`, recall that pre-execution is overlapped with only a *single* lookup of `htab` (refer back to Section 2.2.4 for the details). However, since most of these lookups are actually found in either the D-cache or L2-cache, there is insufficient time for pre-execution to compute the address for the next lookup. For other applications, late pre-execution does not appear to be a problem as the *late pre-executed misses* section is generally small. Even for `mst`, a case where existing prefetching techniques [20, 26, 27] are unable to prefetch early enough, only 9% of misses are pre-executed late.

Figure 5(d) shows another useful performance metric: the amount of data traffic between the D-cache and the L2-cache. Each bar in Figure 5(d) is divided into three categories, explaining if a transfer is triggered by a normal reference (*load+store*; notice that the *full misses* but not *partial misses* in Figure 5(c) are counted in this category), or instead triggered by pre-execution. Pre-execution transfers are further classified as *useful* or *useless*, depending on whether the data fetched gets used by a load or store in the main execution before it is displaced from the D-cache. Ideally, pre-execution will not increase memory traffic, since we expect the same data to be accessed in both the main execution and pre-execution. Nevertheless, Figure 5(d) shows that there are substantial increases in the traffic for both `compress` and `mcf`. The extra traffic in `compress` is due to the fact that multiple paths are being pre-executed—hence data fetched by the wrong paths tends to be useless. In `mcf`, the lists traversed in the while-loop shown in Figure 2(b) can be exceptionally long—with over 1000 nodes in some cases. Thus, nodes pre-executed early became useless if they were displaced from the cache by the nodes that we accessed near the end of the current list. Fortunately, in the other five applications, the additional traffic generated by pre-execution

(a) **Prefetching Schemes**

Application	Prefetching Scheme
Compress	Greedy [20]
Em3d	Chain jumping [27]
Equake	Indirect [22]
Mcf	Chain jumping
Mst	Greedy
Raytrace	Indirect
Twolf	Greedy

(b) **Execution Time Normalized to the Original Case**

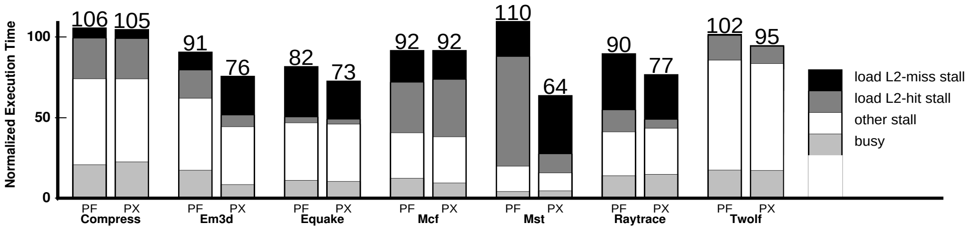


Figure 6. Performance comparison between pre-execution and prefetching (PF = prefetched, PX = pre-executed).

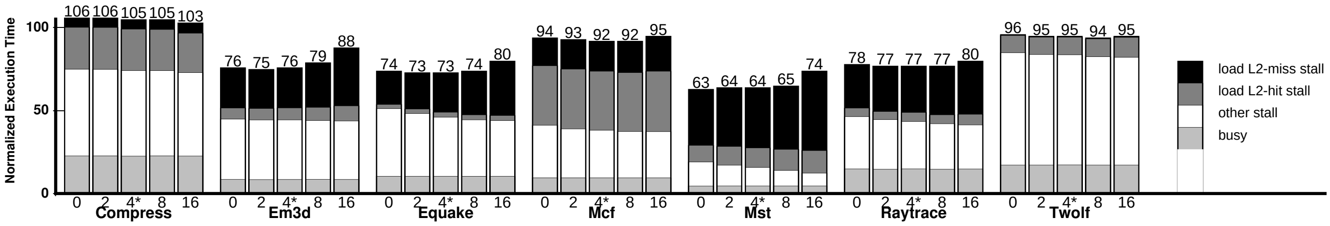


Figure 7. Performance of pre-execution with different thread prioritization (X = adding X to the "ICOUNT" of each *pre-execution* thread; our baseline is 4*). Note that higher ICOUNT values mean *lower* priorities. Execution time is normalized to the original case.

is only less than 7%.

5.2. Performance Comparison with Prefetching

To investigate how well pre-execution performs relative to prefetching, we applied state-of-the-art software-controlled prefetching techniques to our applications. For the two applications involving indirect array references (`equake`) and arrays of pointers (`raytrace`), Mowry's extended algorithm [22] for prefetching these references was used. For the remaining five applications, we experimented with the greedy prefetching and jump-pointer prefetching proposed by Luk and Mowry [20, 21] as well as the extensions of jump-pointer prefetching (full jumping, chain jumping, and root jumping) proposed by Roth and Sohi [27]. In all cases, a wide range of prefetching distances (whenever applicable) were tested, and the best performing one was chosen for the comparison.

Figure 6(a) reports the best software-controlled prefetching techniques we found for individual applications, and Figure 6(b) shows their performance. We first notice that our prefetching results are consistent with Roth and Sohi's [27] for `em3d` and `mst`[2], the two applications that are common to both studies. Comparing pre-execution against prefetching, Figure 6(b) shows that pre-execution outperforms prefetching in six applications. The most prominent case is `mst`, where prefetching is ineffective due to hash-table accesses. On the other hand, prefetching performs as well as pre-execution in `mcf` since the traversal order of its list nodes is repetitive enough to make chain-jumping prefetching effective. The performance advantages of pre-execution observed in `equake` and `raytrace` originate from the improved miss coverage. In contrast, although both pre-execution and prefetching have about the same miss coverage in `em3d`, prefetching incurs significantly higher overhead for maintaining jump pointers. Finally, prefetching covers few misses in `compress` and `twolf`. Overall, we have seen that software-controlled pre-execution can result in significant speedups over prefetching for applications containing irregular data access patterns.

[2]For `mst`, the best software-based jump-pointer prefetching technique found by Roth and Sohi was root jumping. But since greedy prefetching performs better than root jumping in this application, we instead use greedy prefetching.

5.3. Architectural Support

We now explore the impact of three key architectural issues on the performance of pre-execution.

5.3.1. Thread Prioritization

Execution resources are shared by the main execution and pre-execution. However, this sharing does not have to be fair—in fact, it is reasonable to allow the main execution to have a larger share since it directly determines the overall performance. One simple yet effective way to prioritize threads is the ICOUNT scheme [32] previously proposed for choosing which threads to fetch on SMT machines. Under ICOUNT, each thread maintains a priority counter that counts the number of *unissued* instructions belonging to that thread, and fetch priority is given to the thread with the lowest counter value. By controlling instruction fetching this way, threads that issue instructions at faster rates will be given more execution resources across the entire machine than the others.

To give higher priority to the main thread, we bumped up the priority counter of each pre-execution thread by a positive constant. Note that larger counter values result in *lower* priorities. Figure 7 shows the performance of pre-execution with different constants added to the counters. Our baseline is the **4*** cases while the **0** cases correspond to the original ICOUNT scheme. As we see in the figure, the sum of the *busy* and *other stall* components of execution time decreases with larger constants. This implies that our prioritization scheme does help allocate more execution resources to the main thread. However, as we bump up the counter by 8 or above, performance begins to drop in `em3d`, `equake`, and `mst`

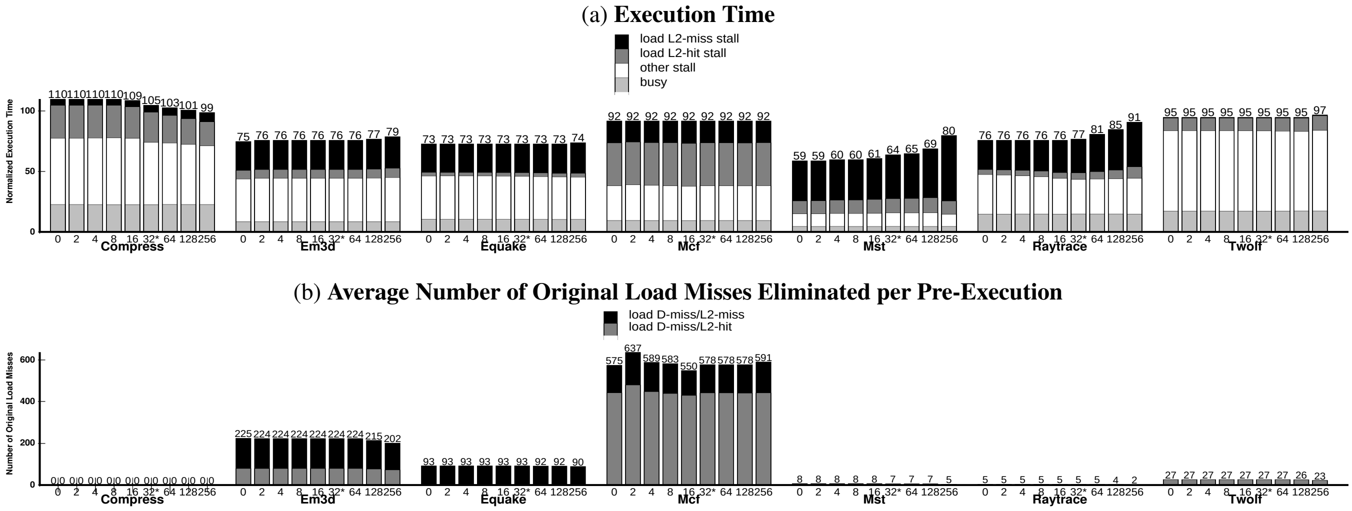


Figure 8. Performance of pre-execution with various thread-spawn latencies (X = X cycles; our baseline is 32*). Part (a) shows the execution time. Part (b) shows the average number of original load misses eliminated per pre-execution.

because pre-execution has too few resources to get its work done. Overall, we find that 4 is a fairly good choice here.

5.3.2. Thread-Spawn Latency

A key implementation issue of pre-execution is the thread-spawn latency, the amount of time required to copy the register state from one thread to another. We have already discussed in Section 3.3 a number of register-copying techniques which involve a tradeoff between hardware complexity and the thread-spawn latency. To determine how fast the copying needed to be done, we experimented with thread-spawn latencies ranging from 0 to 256 cycles. The results are shown in Figure 8. First, we observe from Figure 8(a) that the performance of pre-execution is nearly insensitive to the thread-spawn latency in four applications. Although pre-execution does suffer from larger thread-spawn latencies in `mst` and `raytrace`, they can still achieve speedups of 45% and 18%, respectively, even with a latency as large as 128 cycles. For `compress`, increasing the spawn-latency lowers the chance of generating useless pre-execution and hence actually improves performance. To understand why pre-execution can tolerate such long thread-spawn latencies, we show in Figure 8(b) the average number of original load D-cache misses (further classified into L2-hits and L2-misses) eliminated by each pre-execution. The relatively small number of misses eliminated per pre-execution in `mst` and `raytrace` explains why they are more affected by the thread-spawn latency. In contrast, the other four applications (except `compress`) eliminate an average of at least 20 misses and can eliminate as many as 600. Thus, delaying a thread spawn by a few tens cycles would not significantly affect their performance.

Overall, we see that a very fast register-copying mechanism is not required to exploit the benefits of pre-execution. In order to estimate how much time a software-based copying mechanism would take, we measured the amount of time needed to pass 32 registers from one thread to another through memory using software. Our results indicate that it takes about 24 cycles on average. Therefore, we have assumed a software-based register-copying mechanism that takes 32 cycles in our baseline machine.

5.3.3. Handling Pre-Executed Stores

Our final set of experiments evaluate the impact of the policy for handling stores encountered during pre-execution. Recall from Section 3.4 that we can write these stores into a scratchpad or simply discard them. Figure 9(a) shows how pre-execution performs with three scratchpad sizes: 0, 64, and 128 entries. Surprisingly, pre-execution works equally well with no scratchpad at all (i.e. the **0** cases). There are two possible reasons for this. The first is that computational results that decide which addresses being accessed in pre-execution (and their surrounding control flow) are mostly communicated through *registers*. The second possible reason is that the *store queue* of the underlying machine has already provided sufficient buffering (Note: The store queue is a common piece of hardware in out-of-order machines to hold stores *before* they commit. In contrast, the scratchpad is a special buffer added *beyond* the store queue to hold stores *after* they "commit" during pre-execution.) To find the true reason, we ran an experiment that discarded all pre-executed stores (i.e. they were not written into the store queue at all). The results are shown in Figure 9(b), which indicate that ignoring pre-executed stores does not hurt performance. This evidences that it is the pre-executed stores themselves that do not matter. This is good news since a scratchpad may not be necessary to support pre-execution.

In summary, the experimental results in this section demonstrate that the additional hardware support for our pre-execution scheme can be very minimal: Register copying can be done in software and a scratchpad may not be necessary. Essentially, we only need the support for creating and terminating pre-execution threads as well as that for marking them as non-excepting. In addition, we also show that our modified ICOUNT scheme is quite useful for adjusting the resource sharing between the main execution and pre-execution.

6. Related Work

Dundas and Mudge [12] are among the first who suggested using pre-execution to improve cache performance. In their scheme, the hardware pre-executes future instructions upon a cache miss

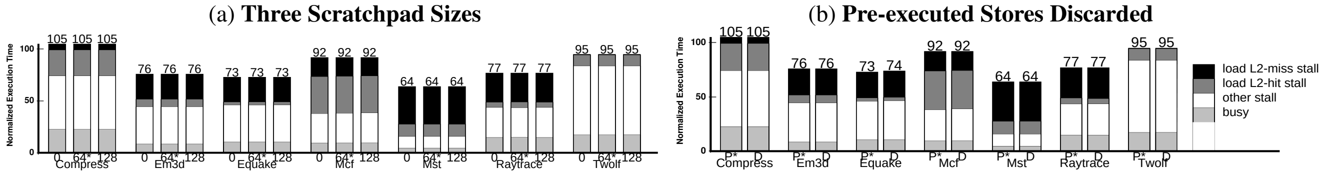


Figure 9. Performance of pre-execution under two different ways to handle pre-executed stores. In part (a), pre-executed stores wrote into a scratchpad; three sizes of the scratchpad are shown (X = X entries; our baseline is 64*). In part (b), all stores were discarded during pre-execution (P* = stores preserved; D = stores discarded).

in a single threaded machine. Compared to typical out-of-order machines, this scheme has the advantage that instructions that depend on the result of the cache miss can be pre-executed before the miss completes. However, it has two drawbacks. First, we must suffer from a cache miss before triggering pre-execution. Second and more seriously, such pre-execution would not be effective for pointer-chasing loads whose addresses actually depend on the values returned from cache misses.

The notion of using helper threads to accelerate the main execution was independently introduced in the form of simultaneous subordinate microthreading (SSMT) [7] and assisted execution [11]. In both schemes, helper threads do not directly run the program: Instead, they are designated to perform some specific algorithms such as stride prefetching and self-history branch prediction. In contrast, since our pre-execution mechanism actually runs the program itself, it is more capable of handling irregular data accesses. In addition, our mechanism is built on top of SMT machines and hence requires less additional hardware support than SSMT and assisted execution do.

Recall that our scheme simply ignores the computational results of pre-execution. More aggressive schemes that actually use the results of speculative threads have also been proposed. Examples of them are the multiscalar architecture [29], thread-level data-speculation (TLDS) [30], threaded multiple path execution (TME) [33], dynamic multithreading (DMT) [2], slipstream processors [31], and speculative data-driven multithreading (DDMT) [28]. Of course, being able to use the results of speculative threads is appealing. Nevertheless, by caring only data addresses but not the final results of pre-execution, our scheme is substantially simpler to implement as we consider the following three aspects. First, our scheme requires no mechanism to integrate the results of pre-execution back to the main execution. Hence, we do not need to verify (and possibly recover from) these speculative results. Second, our scheme can tolerate larger thread-spawn latencies. We have already shown in Section 5.3.2 that it is viable to copy register state using software, thereby eliminating the need of special hardware-based register-copying mechanisms. Third, by relaxing our scheme from concerning the accuracy of pre-execution results, it has higher flexibility in deciding where and when to launch pre-execution in the program.

Several researchers have investigated ways to pre-execute only a subset of instructions (as known as a *slice*) that lead to performance degradation such as cache misses and branch mispredictions. Zilles and Sohi [35] found that speculation techniques like memory dependency prediction and control independence can be used to significantly reduce the slice size. Recently, a collection of schemes [4, 5, 9, 15, 26, 28, 31, 36] have been proposed to construct and pre-execute slices. They differ from ours in two major ways. First, our pre-execution strategies (i.e. those presented in Section 2.2) are designed based on a high-level understanding of data access patterns. Thus, our strategies may be easier for the programmer or compiler to apply. Second, a common theme of our strategies is to pre-execute *multiple* data streams simultaneously while the other schemes focus on pre-executing a *single* data stream/branch as quickly as possible. Nevertheless, we believe that our approach and theirs can be complementary, and we leave an integration of them as potential future work.

7. Conclusions

As multithreaded machines emerge into mainstream computing, it is appealing to utilize idle threads on these machines to improve *single-thread* performance. In this paper, we have examined such a technique: *pre-execution*. With the harnessing of *software*, pre-execution can accurately generate data addresses and fetch them in advance, regardless of their regularity. Experimental results demonstrate that our technique significantly outperforms software-controlled prefetching, offering an average speedup of 24% in a set of irregular applications. Another important finding is that the additional support required by SMT machines in order to enjoy these performance benefits is only minimal: mainly the mechanisms for launching and terminating pre-execution, and for making pre-execution non-excepting. Given these encouraging results, we advocate a serious consideration of supporting software-controlled pre-execution in future SMT machines.

8. Acknowledgments

I thank Joel Emer's encouragement of pursuing a software-based approach to pre-execution (see his HPCA-7 keynote speech [14] for the philosophy behind). Yuan Chou, Robert Cohn, Joel Emer, and Steve Wallace gave insightful comments on early drafts of the paper. In addition, I thank the Asim team for supporting the simulator and the Alpha 21464 team for graciously sharing their computing resources. Finally, I appreciate the documentary help from Artur Klauser, Todd Mowry, and Robert Muth.

References

[1] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. 17th ISCA*, pages 104–114, May 1990.

[2] H. Akkary and M. Driscoll. A dynamic multithreading processor. In *Proc. 31st MICRO*, pages 226–236, Nov 1998.

[3] Alpha Development Group, Compaq Computer Corp. *The Asim Manual*, 2000.

[4] M. M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proc. 28th ISCA*, 2001.

[5] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically allocating processor resources between nearby and distant ILP. In *Proc. 28th ISCA*, 2001.

[6] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[7] R. S. Chappel, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proc. 26th ISCA*, pages 186–195, May 1999.

[8] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5), May 1995.

[9] J. D. Collins, H. Wang, D. M. Tullsen, H. J. Christopher, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proc. 28th ISCA*, 2001.

[10] Standard Performance Evaluation Corporation. *The SPEC95 benchmark suite*. http://www.specbench.org.

[11] M. Dubois and Y. H. Song. Assisted execution. Technical Report CENG Technical Report 98-25, University of Southern California, October 1998.

[12] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proc. 1997 International Conference on Supercomputing*, 1997.

[13] J. S. Emer. Simultaneous Multithreading: Multiplying Alpha Performance. *Microprocessor Forum*, October 1999.

[14] J. S. Emer. Relaxing Constraints: Thoughts on the Evolution of Computer Architecture. *Keynote Speech for the 7th HPCA*, January 2000.

[15] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proc. 31st MICRO*, pages 59–68, Dec 1998.

[16] J. L. Henning. SPEC CPU2000: measuring cpu performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[17] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 microprocessor architecture. In *Proc. International Conference on Computer Design*, October 1998.

[18] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *Proc. 25th ISCA*, pages 250–259, June 1998.

[19] N. Kohout, S. Choi, and D. Yeung. Multi-chain prefetching: Exploiting memory parallelism in pointer-chasing codes. In *ISCA Workshop on Solving the Memory Wall Problem*, 2000.

[20] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proc. 7th ASPLOS*, pages 222–233, October 1996.

[21] C.-K. Luk and T. C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers (Special Issue on Cache Memory)*, 48(2):134–141, February 1999.

[22] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[23] T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proc. 30th MICRO*, pages 314–320, December 1997.

[24] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[25] 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):233–263, March 1995.

[26] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Proc. 8th ASPLOS*, pages 115–126, October 1998.

[27] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proc. 26th ISCA*, pages 111–121, May 1999.

[28] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proc. 7th HPCA*, 2001.

[29] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proc. 22nd ISCA*, pages 414–425, June 1995.

[30] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallellization. In *Proc. 4th HPCA*, February 1998.

[31] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: Improving both performance and fault tolerance. In *Proc. 9th ASPLOS*, Nov 2000.

[32] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. 23rd ISCA*, pages 191–202, May 1996.

[33] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *Proc. 25th ISCA*, pages 238–249, June 1998.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. 22nd ISCA*, pages 24–38, June 1995.

[35] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *Proc. 27th ISCA*, pages 172–181, June 2000.

[36] C. B. Zilles and G. S. Sohi. Execution-base prediction using speculative slices. In *Proc. 28th ISCA*, 2001.

Appendix: Compiler Algorithm for Inserting Pre-Execution

Phase I: Locality Analysis

Step 1: Locate where cache misses occur in the program. This step can be accomplished through some low-overhead profiling tools such as DCPI or through cache simulations.

Step 2: Determine if the misses identified in Step 1 are generated by access patterns that can potentially be pre-executed. For the four important classes of accesses patterns we discussed in Section 2.2, they are all recognizable by the compiler: Previous prefetching work has used compilers to recognize pointer-based data structures [20] and array (both affine and sparse) accesses [22]; control-flow analysis and call-graph analysis [24] can easily discover multiple paths and procedure calls.

Step 3: If Step 2 shows that pre-execution might be beneficial, the compiler has to determine whether the pre-execution and actual execution will access the same data. This can be computed through aliasing analysis, which typically gives three possible answers: "yes", "no", or "maybe". Since the overhead of pre-execution is relative low compared to the cache miss penalty, the compiler can aggressively perform pre-execution for both the "yes" and "maybe" cases. For example, for the `mst` code fragment shown in Figure 3(b), the compiler optimistically assumes that the value of `tmp`$\rightarrow$`next` remains the same during the pre-execution and actual execution and hence can pre-execute the next hash.

Phase II: Scheduling

Step 4: At this point, the compiler must decide how far ahead it needs to pre-execute. Ideally, we would like to pre-execute as far as possible without polluting the cache. However, computing the volume of data brought in by pre-execution is challenging due to non-compile-time constants like unknown loop bounds, the length of linked lists, etc. There are three possible approaches here. The first approach is to assume that pre-execution would access a small volume of data with respect to the cache size. Under this assumption, the compile will pre-execute as far as possible. This is a reasonable assumption for today's machines since their caches (especially the L2 caches) are typically large enough to hold the data accessed within a few iterations of the innermost loop. The second approach relies on the user to specify the expected values of those non-compile-time constants through program annotations. In the third approach, the compiler can generate two versions of pre-execution code: One corresponds to the "small volume" case and the other corresponds to the "large volume" one. In addition, it also includes the code that will adapt to one of these two versions at run-time. A similar approach has been suggested [22] for prefetching loops with unknown bounds.

Step 5: Perform code transformations to facilitate pre-execution. These transformations are mainly for generating explicit target addresses for pre-execution. An example of them is the loop unrolling done in Figure 3(b). Moreover, additional code may be required to preserve the correctness of the main execution. For instance, both `next_1` and `next_2` in Figure 3(b) have to be reloaded prior to the main execution.

Step 6: Finally, insert **PreExecute_Start**'s and **PreExecute_Stop**'s into the program. **PreExecute_Start**'s are inserted at the earliest points in the program where pre-execution can be launched with the correct data addresses. **PreExecute_Stop**'s are put at the expected ends of pre-execution as well as all possible exits.