Low Overhead Helper Threads for Software-Based Techniques

Gautham K.Dorai and Donald Yeung
Department of Electrical and Computer Engineering and
Institute for Advanced Computer Studies
University of Maryland at College Park

Abstract

Software-assisted performance enhancement techniques perform software instrumentation of the application code–we call these overhead instructions. The effectiveness of software-assisted techniques is often limited by the impact of the overhead instructions on program execution time. This paper proposes a new technique to execute overhead instructions on separate helper threads in a simultaneous multithreading (SMT) processor. We show that a small modification to the fetch and issue policies in an otherwise unmodified SMT enables the SMT processor to effectively schedule the overhead instructions into free slots with only minimal impact on main thread performance. Executing the overhead instructions on separate threads 1) enables the processor to distinguish the overhead instructions from the original program instructions, and 2) allows the processor to give higher fetch and issue priority to the main thread instructions.

To demonstrate the usefulness of our technique, we apply it to software prefetching by performing the prefetch instrumentation in helper *prefetch threads* rather than in the main thread, as is conventionally done [11] in single-threaded software prefetching. Our *SMT prefetching* approach provides two benefits. First, SMT prefetching achieves higher performance gains than conventional single-threaded software prefetching because the overhead of the software instrumentation code is reduced. Second, SMT prefetching achieves performance gains even when the prefetch instrumentation is applied naively to all loops in the application. In contrast, due to high overhead, conventional single-threaded software prefetching often requires detailed cache-miss profiles to selectively instrument those loops with high cache-miss counts in order to manage overhead.

Our results demonstrate that naive SMT prefetching applied to 5 *SPEC* applications, provides a 7.31% performance gain over no prefetching, with a minimal performance degradation in one of the applications. The equivalent naive use of conventional single-threaded software prefetching degrades performance in 4 out of the 5 applications. The additional performance achieved by SMT prefetching is due to a 88.0% reduction in overhead compared to conventional single-threaded software prefetching. In another benchmark, where single-threaded software prefetching achieves significant performance gain, SMT prefetching still performs better because of the reduction in overhead. When conventional single-threaded software prefetching uses profile information, its performance gain in the 5 *SPEC* benchmarks increases to 0.70%, which is still 6.60% less than naive SMT prefetching.

1 Introduction

Software-based performance enhancement techniques such as software prefetching [11], software-managed caches [8], and informing memory operations [15] require software instrumentation of the application code to implement the optimizations. The runtime overhead incurred by these added

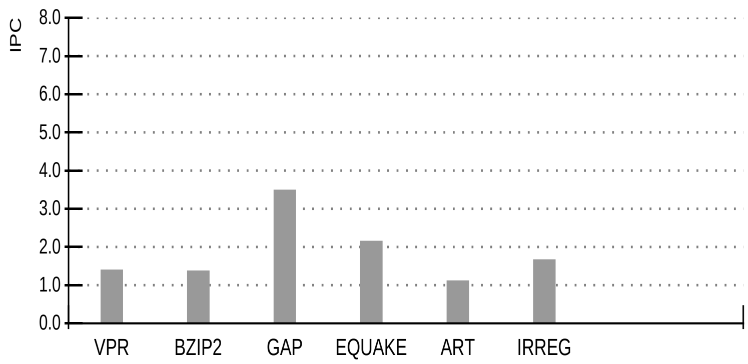

Figure 1: Potential for unused slots

instructions limits the effectiveness for such techniques. Henceforth, we shall call the instructions added by software-based performance enhancement techniques as *overhead instructions*.

The impact of overhead instructions on the execution time of the original program can significantly reduce the overall gain that is achieved by their instrumentation. In some cases, software-based techniques require careful analysis of the program to selectively instrument the overhead instructions so that the performance benefits outweigh the additional overhead. Researchers have looked into selection schemes [9, 10] to minimize these overheads in techniques such as software prefetching. Recent profiling techniques [9] have looked into more fine-grained analysis of paths to a load instruction to enable more accurate determination of when the benefits of memory latency tolerance will outweigh the prefetch overhead. The disadvantage of these schemes is that they require off-line analysis of the program execution, and in some cases very complex compiler optimizations of the code. Special-purpose hardware [12, 14] that implements these software-based techniques have also been proposed as a solution. These schemes introduce minimal or no overhead to the original program, but at the expense of increased processor complexity due to the additional hardware.

In theory, software-based optimizations should benefit from the trend in today's high-performance microprocessors towards wider-issue machines. As processors become wider, the number of free issue slots will increase for those applications that cannot exploit the additional execution bandwidth. Figure 1 shows the average IPC achieved by several important benchmarks on an 8-way issue out-of-order processor. The data illustrates that the execution capacity of the processor is far from saturated, leaving many unused issue slots. If the overhead instructions in software-based optimizations can exploit these free issue slots, much of their runtime cost can be mitigated, increasing their effectiveness. Unfortunately, in practice, naive instrumentation of the overhead instructions into the original program code cannot make effective use of the available free slots for two reasons. First, there is no mechanism by which the processor can distinguish the overhead instructions apart from the main program instructions. This prevents the processor from giving higher priority to the main thread instructions as they pass through the pipeline. As a result, the overhead instructions can take resources away from the main program instructions during periods when resources are scarce, hence impeding the progress of the main program instructions. Second, single-threaded execution of both the overhead and main program instructions requires that they be fetched sequentially. Even if the processor could identify the main program instructions, the processor would not be able to give them higher priority during instruction fetch.

This paper proposes a technique to enable an SMT processor [3, 1] to dynamically schedule the overhead instructions by executing them in separate helper threads. At the heart of our technique is a fetch and issue priority scheme that favors the main thread. The priority scheme enables the original program running on the main thread to proceed with negligible interference from the overhead instructions executing in helper threads. The helper threads also proceed by using up the free slots available. To demonstrate the usefulness of our technique, we apply it to software prefetching, an area in which overhead reduction is the focus of considerable ongoing research. Our prefetching scheme, named SMT prefetching, uses helper threads to execute prefetch code on behalf of the main thread with near-zero overhead.

Figure 2 illustrates how our priority scheme in an SMT processor dynamically allocates free slots to prefetch instructions without impacting the main thread's progress. Figure 2(a) shows the original program without prefetch instrumentation. In conventional single-threaded software prefetching, shown in Figure 2(b), prefetch instructions occupy issue slots that would have otherwise been used by the original program instructions, thereby adversely impacting the execution time of the original program. Finally, Figure 2(c) shows how SMT prefetching steers the overhead instructions exactly into those issue slots where the original program instructions are stalled, hence passing through the pipeline with minimal interference.

Our technique provides two significant advantages to conventional single-threaded software prefetching - 1) SMT prefetching achieves performance gains even when applied naively to all loops in the application, since it introduces very negligible overhead. In contrast, conventional single-threaded software prefetching often requires detailed cache-miss profiles and off-line optimizations

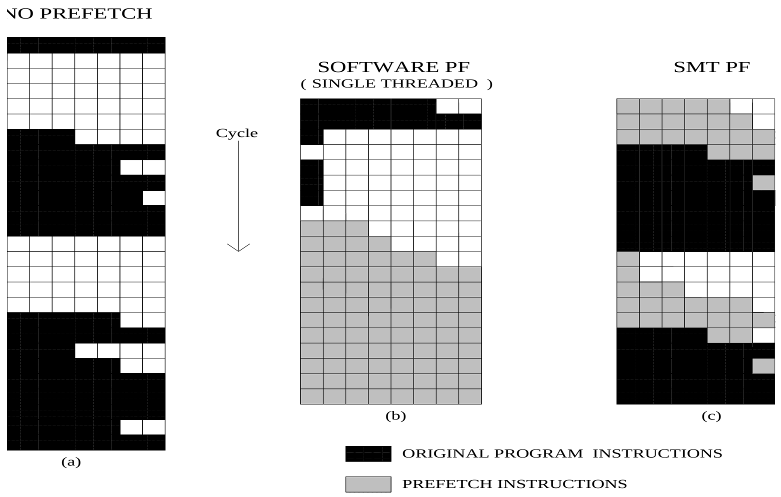

Figure 2: Issue slots snapshot - Microbenchmark - a) No Prefetching b) Single Threaded Software Prefetching c) SMT Prefetching with helper threads

to selectively instrument prefetch instructions, so that overhead does not outweigh performance gain 2) Our approach uses an SMT processor with minimal hardware support to implement a low-overhead software prefetching technique. In our experiments, we compare the performance of SMT prefetching applied naively to all the loops with single-threaded selective software prefetching, which uses detailed cache-miss profile information and off-line analysis to identify and prefetch only the memory-intensive loops. We also compare the performance of naive prefetching to single-threaded software prefetching applied naively to all loops.

SMT prefetching, when applied naively to all loops in 5 *Spec* applications, provides a 7.31% performance gain over no prefetching, with a minimal performance degradation in one of the applications. Our experimental results also demonstrate that the equivalent naive use of conventional single-threaded software prefetching degrades the performance in 4 out of the 5 applications. The additional performance achieved by SMT prefetching is due to an 88.0% reduction in overhead compared to conventional single-threaded software prefetching. Even when single-threaded software prefetching achieves significant performance gain, as seen in another benchmark, SMT prefetching still performs better because of reduction in overhead. When detailed profile information is used to selectively prefetch loops with high cache-miss counts, the performance gain of single-threaded software prefetching increases to 0.70%, with performance degradation in 2 of the 5 *Spec* applications. But, this is still 6.60% less than naive SMT prefetching.

The rest of the paper is organized as follows. Section 2 discusses our technique in detail and investigates how the priority schemes help exploit the free slots available. Next, Section 3 describes the specific issues in the implementation of helper threads on an SMT processor. Then, Section 4 presents the experimental framework used and Section 5, the results. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2 SMT Helper Threads

Section 2.1 discusses the basic concepts behind the implementation of helper threads on an SMT processor, then Section 2.2 gives a detailed description of our priority schemes. Next, Section 2.3 discusses a specific application of these helper threads in software-controlled prefetching. Finally, Section 2.4 gives a detailed reasoning as to how the priority schemes help lower the overhead of helper threads.

2.1 Basic Concepts

An SMT processor allows multiple independent threads to execute simultaneously in the same cycle by sharing the functional units. In our approach, instead of executing independent programs on the multiple threads, we use them as helper threads to enhance the performance of a single application running on the main thread. The additional instructions that we call overhead instructions are executed on these helper threads. The main thread will fork these helper threads whenever required. Before forking, the main thread will communicate any parameters that need to be passed on to the helper threads and also specify the starting PC value of the instruction sequence from where the helper threads will have to start execution. The goal of our technique is to enable the processor to interleave the overhead instructions into free slots of the main thread without impacting the main thread's performance.

Overhead instructions executing on separate helper threads can be fetched in parallel with main thread instructions. This eases the fetch stage bottleneck caused by the sequential fetch of overhead instructions in the single-threaded case. A multithreaded processor can explicitly assign priorities to the different threads. Our approach integrates a fetch priority scheme and an issue priority scheme that favors the main thread over helper threads. A more detailed description of the priority schemes is given in Section 2.2. Our experiments show that a combination of the fetch and issue priority schemes leads to the scheduling of overhead instructions in free slots of the main thread.

2.2 Priority Schemes - How do they work

2.2.1 Fetch Priority Scheme

Multithreaded processors have the unique ability to control the instructions that are fetched from several threads. In the case of applications with distinct main and helper threads, we would like to design the fetch stage in such a way that the helper threads do not steal fetch slots away from the main thread. We would also have to ensure that the helper threads are not starved; otherwise, their utility is lost. Given the frequent branches in current programs, and since we do not fetch beyond a branch in the same cycle, there is sufficient free fetch bandwidth for these helper threads to utilize. An ideal fetch priority scheme would ensure that the helper threads do not interfere with the progress of the main thread, and at the same time issue as many instructions as it needs into the pipeline, by using up the free fetch slots.

Our experiments use an SMT processor with a modified ICOUNT instruction fetch policy [3] that gives the main thread higher priority over helper prefetch threads. The ICOUNT, or number of active instructions in the decode, issue, writeback and commit stages, is the factor that decides the thread priority. No fetch slots are given to a thread that is stalled. We artificially bias the fetch scheme in favor of the main thread by setting the ICOUNT of the prefetch threads to a predefined value, the offset. As the offset is increased, the impact of overhead instructions on the main thread reduces, as the main thread is given a higher priority. It is also important to note that there is the drawback that a high offset value might lead to the helper thread instructions not issuing as early as required. The fetch priority scheme also plays a role in reducing the number of useless instructions that the helper thread injects into the pipeline when it executes code such as busy waiting on a synchronization parameter.

2.2.2 Issue Priority Scheme

The issue stage of the SMT processor has been modified to give priority to main thread instructions over those of the helper thread. The processor issues all the instructions of the main thread from the ready-to-issue instruction queue, and then issues the available helper thread instructions in the remaining slots. Our study in Section 2.4 reveals that the issue priority scheme helps maintain the

```
(a)   /*Prologue Loop*/
        for (i = 0 to PD step S)
            prefetch(&ftab[i]);

      /*Main Loop*/
        for(i=0 to last − PD) {

          if(prefetch−predicate)
            prefetch( &ftab[i+PD]);

          /*original loop code*/
      }

       /*Epilogue Loop*/
         for( last PD iterations ) {

          /*original loop code*/

      }
```

```
b)  Main Thread
      fork(prefetch_loop);
        for(i=0 to last) {
         /*original loop code*/
         sync_code ;
        }
    Prefetch Thread
      prefetch_loop () {
        for(i = 0 to last PD ) {
         if(prefetch−predicate)
          prefetch(&ftab[i]);
          sync_code ;
        }
     }

PD −− Prefetch Distance
 S  −− sizeof(cache_block)/sizeof(ftab)
```

Figure 3: Eg: Bzip2 (SPEC2000 benchmark) - Abstract versions of prefetching code for (a) Single Threaded Software Prefetching (b) SMT Prefetching

issue slot occupancy pattern of the main thread even when helper threads are competing for issue resources.

2.3 An example : Software Prefetching

2.3.1 Single-Threaded Software data Prefetching

Single threaded software prefetching is implemented by using Mowry's compiler algorithm [11]. The original loop is split into the prologue loop, main loop and the epilogue loop. In the case of affine array prefetching, we do locality analysis to identify the array references that are likely to cause cache misses. We then add a prefetch predicate to make sure that we prefetch only for the isolated cache missing references. The algorithm used to prefetch indexed array accesses is similar to that of affine array access, except that the prefetches for the data array reference has to be scheduled before that of the indexed array. The prologue loop, prefetch instructions in the main loop and the additional loop transformation instructions constitute the overhead instructions for single-threaded software data prefetching. A sample code for single-threaded software data prefetching from one of the SPEC2000 benchmarks (Bzip2) is shown in Figure 3(a).

In the case of single-threaded software prefetching, the overhead instructions have a significant impact on the execution time of the original program. Hence, we have to be selective about the loads for which we apply the prefetching technique, so that the additional overhead does not offset the gain due to prefetching. We base our selection scheme on summary cache miss profiles. The selectivity is based on the following predicate - *Prefetch Overhead* $<$ *L1-miss rate* $*$ *L1-L2 latency*

+ *L2-miss rate* $*$ *L1-Memory latency.* In computing the prefetch overhead, we assume that the overhead instruction count to prefetch each load is 12 (8 instructions for the prologue loop + 1 prefetch instruction in the prologue loop + 2 instructions for the prefetch predicate in the main loop + 1 prefetch instruction in the main loop) , and the average IPC for the loop is 1.5.

2.3.2 SMT Prefetching

SMT Prefetching is implemented by combining the prologue loop and the main loop prefetch code from the single-threaded software prefetching case, and executing it on separate helper threads. Henceforth, we shall call these helper threads as prefetch threads, and the prefetching sequence for each main thread loop as prefetch loops. Since all the prefetch code is now run on the prefetch thread, there is no loop transformation to be done on the original loop.

Before the main thread enters a loop, it forks a prefetch thread to execute the corresponding prefetch loop. The prefetch thread then enters the prefetch loop and executes the prefetch sequence for the main thread. Sample abstract code for SMT prefetching compared to single-threaded software prefetching is shown in Figure 3(b). Since the overhead of these prefetch threads are negligible, we do away with cache miss profiling tools and apply this technique universally to all the loops in the simulated region.

2.4 SMT Prefetching - Dynamic Scheduling of Overhead Instructions

Overhead instructions have a negative impact on the execution time of the original program, when they compete for resources and take up slots that would have been used by them. Low instruction level parallelism, branch mispredictions, and memory stall in programs lead to a large number of potentially unused slots for these overhead instructions to utilize. In the case of single-threaded software prefetching, the overhead instructions cannot utilize these free slots. This can be attributed to two reasons. First, in single-threaded software prefetching the overhead instructions are fetched sequentially along with the original program instructions. Hence, the overhead instructions steal fetch slots away from the original program and delays its progress. This is particularly significant in the case of the prologue loop. The overhead instructions serialize with the original program instructions and hence the free slots remain unused. Second, since the overhead instructions cannot be distinguished from those of the original program, the overhead instructions cannot be scheduled to use up the free slots in the original program execution. Overhead instructions preempt slots

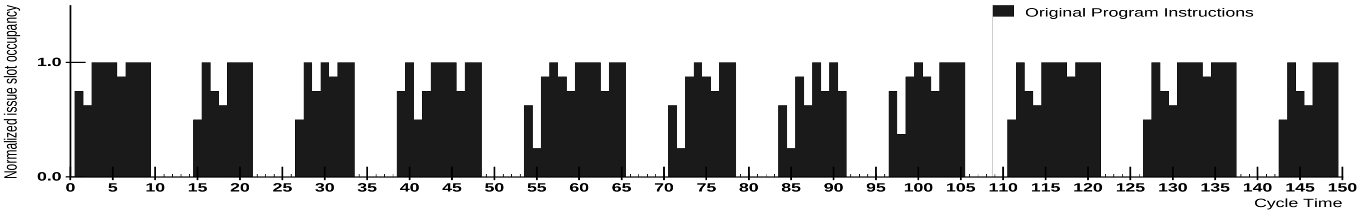

Figure 4: Issue stage scheduling - No prefetching(Microbenchmark)

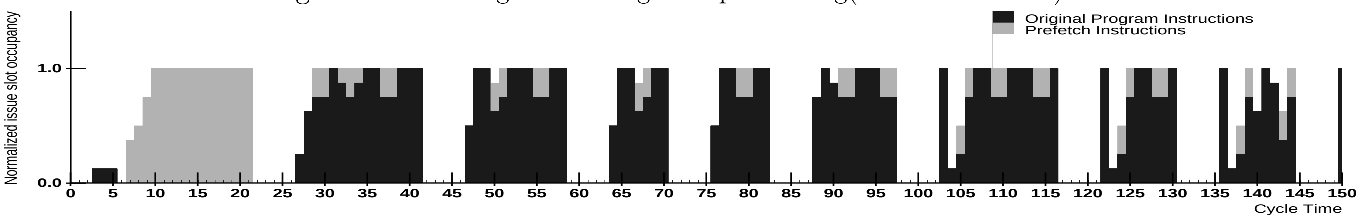

Figure 5: Issue stage scheduling - Software prefetching(Microbenchmark)

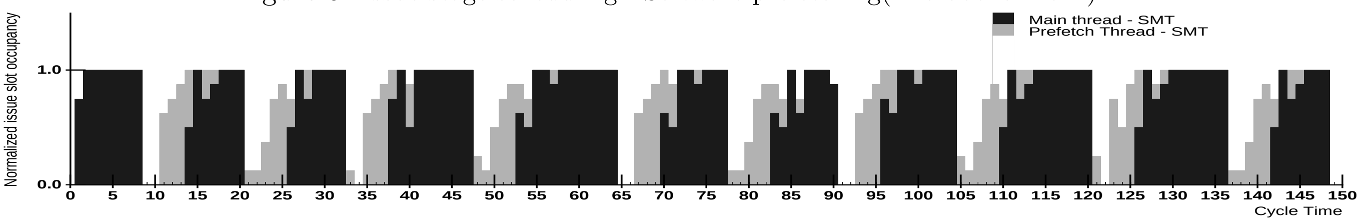

Figure 6: Issue stage scheduling - SMT Prefetching (Microbenchmark)

away from the original program and delay the completion of the original program instructions.

In order to better understand the dynamics of the scheduling of overhead instructions, we looked at the allocation of resources in the issue stage for the two prefetching schemes, on a simple microbenchmark. The microbenchmark has a nested loop structure similar to those in many of the real integer benchmarks. It has an IPC of 2.05 on a perfect memory system. We apply software prefetching to the inner loop. All instructions in the prefetch thread of the SMT prefetching case were taken as overhead instructions, while in the single-threaded prefetching case, the added overhead instructions were explicitly marked for identification. All processor parameters are identical to those presented in Section 4. In order to highlight the impact of the additional instruction overhead on the original program, all experiments were conducted on a system in which all the accesses to the instruction and data caches were assumed to be hits.

Figure 4 shows the use of issue slots by the original program without prefetching. Each block of original program instructions constitute a single iteration of the loop. The snapshots indicate that there are intervals in which the original program does not issue any instructions at all. A look into the pipeline trace reveals that these stalls are due to branch mispredictions in the original

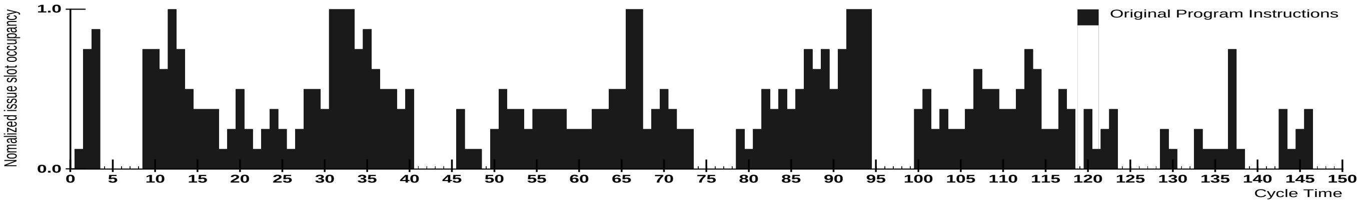

Figure 7: Issue stage scheduling - No prefetching(Spec benchmark-Vpr)

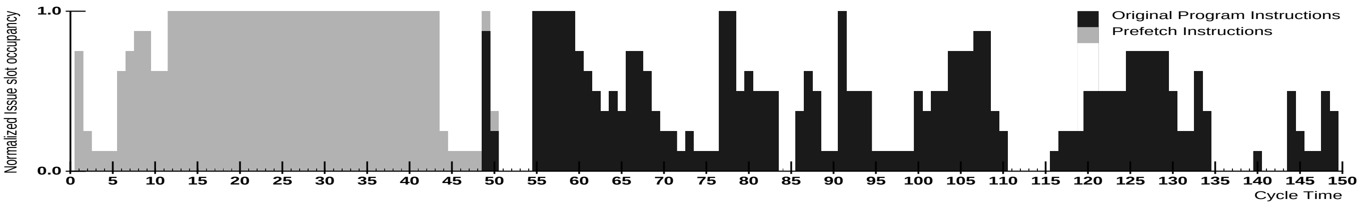

Figure 8: Issue stage scheduling - Software prefetching(Spec benchmark-Vpr)

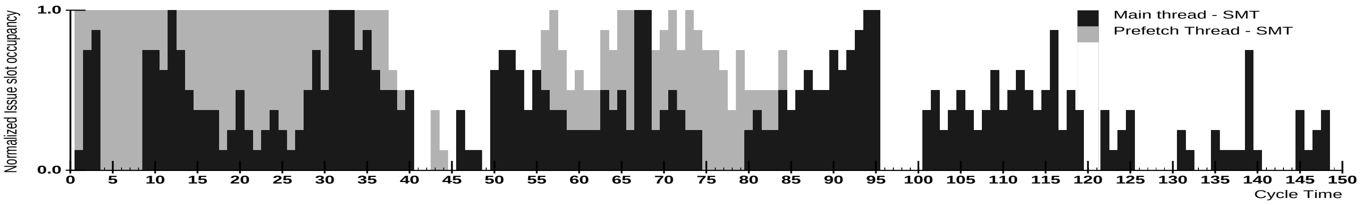

Figure 9: Issue stage scheduling - SMT prefetching(Spec benchmark-Vpr)

program. Overhead instructions utilizing these idle slots will have an almost zero impact on the original program.

Figure 5 shows the same issue slot usage, but for single-threaded software prefetching. The bunch of overhead instructions in the beginning belong to the prologue loop and it delays the start of the first iteration of the original program loop. We also see that the overhead instructions in the loop iterations use slots that the original program instructions would have used, and hence the cycle time for the completion of each iteration is also expanded. The overhead instructions do not find the free issue slots created by the branch mispredictions because - 1) the prologue loop instructions get fetched and executed sequentially before the start of the loop in the original program. 2) the overhead instructions that get fetched along with original program instructions cannot be identified and scheduled to utilize the free slots. Instead, they compete for slots with the original program instructions and complete before the free slots appear.

In the case of SMT prefetching, the prefetch instructions are in a separate prefetch thread. This explicit parallelism that exists between the overhead instructions and those of the original program enables the processor to dynamically schedule the overhead instructions to fill up free slots, thereby not competing for slots used by the original program. Also, since there are no overhead instructions that serializes with the original program, the original program loop does not get delayed. Figure 6 shows the issue slot occupancy of the original program instructions (main thread) and the overhead instructions (prefetch thread) on the SMT prefetching scheme. The utilization of the free slots to issue the overhead instructions is clearly demonstrated. The SMT prefetching case has more overhead instructions than single-threaded software prefetching, since all instructions in the prefetch thread, including synchronization and loop instructions, are taken as overhead instructions. We present the same issue slot usage profiles on one of the SPEC benchmarks (Vpr) in Figure 7, Figure 8, and Figure 9. The traces show that the number of iterations for the loop is less than the prefetch distance, and hence only the prologue loop issues prefetches in the single-threaded software prefetching case. In the SMT prefetching case, the overhead instructions are diffused into free main thread slots. Similar issue slot occupancy traces of the original program instructions on both the SMT prefetching version and the no prefetching version evidence the fact that the prefetch threads have a minimal impact along the original execution path.

3 Prefetch Threads - Implementation Issues

In this section we discuss the basic hardware and software issues involved in implementing prefetch threads on an SMT processor. Section 3.1 discusses the hardware and ISA support on the SMT processor, and Section 3.2 describes the thread creation support. Section 3.2.1 and Section 3.2.2 present specific software implementation issues for prefetching using these helper threads.

3.1 Hardware Support

We use a simultaneous multithreading(SMT) processor [1, 3] as the platform for this study. We assume that the SMT processor provides basic ISA support for multithreading [4]. The ISA should provide a fork instruction that activates a hardware context after initializing its program counter to a specified address. It should also support primitives to suspend and resume hardware contexts by specifying the context ID. A suspend instruction causes a pipeline flush of all instructions belonging to that context. The processor stops fetching and issuing instructions from a suspended context. Suspend and resume instructions take 1 cycle to execute. A read only register that contains a bit map of active and inactive contexts is also available.

(a) Main Thread

```
1 resumeID = LOOP1;
2 resumeContext(cxt_ID);
3   for(i = 0 to N) {
4     /*original loop code*/
5     sync_code;
  }
```

(b) Prefetch Thread

```
  /*dispatcher loop*/
1 void DISPATCHER(/*parameters*) {
2     while(1) {
3       (resumeID)();
4       suspendContext(cxt_ID) ;
      }
  }

  /*prefetch loop*/
5 void LOOP1() {
6     /*prefetch code*/
7     sync_code;
  }
```

Figure 10: Sample Abstract code of (a) Main thread (b) Prefetch thread with dispatcher loop and prefetch loop

3.2 Thread Creation

The cost of thread creation, which includes instructions to initialize the forked thread, is not trivial. We use a *"recycled"* thread model [13] to avoid incurring this overhead every time we fork a thread. When we try to resume a context that has not yet been suspended or vice versa, the resume/suspend instruction essentially becomes a nop. This essentially neutralizes any race conditions caused by the main thread trying to resume/suspend active helper threads.

In the recycled thread model, a thread for each hardware context is created once during the initialization of the application. A sample abstract code for a recycled thread model is shown in Figure 10. The main thread code is shown in Figure 10(a). The main thread forks a prefetch thread before it enters the loop (line 2 Figure 10(a)). In the recycled thread model, the forked thread does not enter the prefetch loop. The forked thread enters a dispatcher loop, from where it jumps indirect to the prefetch loop. This dispatcher loop is shown in Figure 10(b). The starting PC value of the prefetch loop is passed on to the prefetch thread, before it forks the thread (line 1 Figure 10(a)). The dispatcher loop calls the prefetch loop (line 3 Figure 10(b)), where the prefetch sequence for the main thread loop gets executed. On completion of the prefetch sequence, the prefetch thread suspends itself, and waits to get resumed by the main thread (line 4 Figure 10(b)). In such a recycled thread model, many virtual prefetch threads share a single forked thread. The following sub-sections deal with the important issues of inter-thread synchronization and communication.

3.2.1 Inter-Thread Communication

The corresponding prefetch thread is resumed just before the main thread enters the loop. Communication to pass parameters from the main thread to the prefetch thread is done through global memory. The main thread passes parameters to the prefetch thread by storing the values in these global variables before resuming the context. In some cases where the number of arguments that have to be passed are large, additional overhead is incurred on the main thread. However in most of the applications, this overhead is usually small since the number of parameters that must be passed is typically small and the data is normally in the L1 cache.

3.2.2 Inter-Thread Synchronization

To implement synchronization between the main thread and the prefetch thread, we allocate a global counter in memory that is initialized to the prefetch distance PD. During each iteration the main thread increments the counter. The prefetch thread also maintains a private counter (initialized to zero) that increments every iteration. The prefetch thread polls the main thread counter and proceeds only if its counter value is less than that of the main thread. The polling mechanism has the disadvantage of injecting useless instructions into the pipeline. In view of the very few instructions that have to be added to the main thread, we implemented this mechanism.

4 Experimental Framework

Our experiments use the SMT simulator from [4]. It is an out of order simulator model from SimpleScalar v2.0 augmented to simulate the SMT pipeline. In addition, we added our proposed fetch and issue priority schemes to the underlying simulator. An offset value of 48 was experimentally determined (see Section 5.2) to provide enough slots to the prefetch threads to allow them to make progress without impacting the performance of the main thread.

Most of our simulations assume an 8-way SMT with 4 hardware contexts. This configuration is similar in resources to the commercial Alpha 21464 SMT processor [1]. Table 1 presents in detail the parameters used in the model for our experiments. To evaluate the performance benefits of using our low overhead helper prefetch threads, we compare its performance with conventional single threaded software data prefetching. We conducted our experiments on a collection of integer and floating point applications, a majority of which are from the SPEC CPU2000 benchmark suite.

Parameter	Value	Parameter	Value
Hardware contexts	4	BTB Size	2048 entries
Issue Width	8	Instruction Window Size	128
Fetch Queue Size	8	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	4/4
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.

Name	Description	Suite	Data Reference Pattern	Insts. Skipped	Simulated Insts.
Vpr	Integrated circuit computer-aided design program	Spec00(Int)	affine, indexed array & structure array dereferences	233M	60M
Bzip2	Compression algorithm	Spec00(Int)	affine & indexed array	126M	22M
Gap	Language and library designed for computing in groups	Spec00(Int)	affine array using pointers	157M	105M
Equake	Simulates seismic wave propagation in large basins	Spec00(FP)	affine & indexed array	1186M	18M
Art	Models several characteristics of organic neural processing	Spec00(FP)	structure array dereferences	11M	20M
Irreg	Irregular solve over mesh	- (FP)	affine & indexed array	977M	29M

Table 2: Benchmarks - Description and data reference patterns.

As mentioned before, we applied prefetching to array, indexed array references and structure array dereferences. The benchmarks used along with their data reference patterns are given in Table 2.

We used a loop profiling tool to determine all the loops that are accessed in the region of simulation. We apply SMT prefetching to all these loops irrespective of their cache miss rates and compare its performance against single threaded software prefetching applied to the same. We also use detailed summary profile analysis, to compare the performance of SMT prefetching for all the loops, with selective single threaded software prefetching. As mentioned in Section 2.3.1 the selectivity is based on the following predicate - *Prefetch Overhead* $<$ *L1-miss rate* $*$ *L1-L2 latency* $+$ *L2-miss rate* $*$ *L1-Memory latency*.

5 Experimental Results

We now present the results of our simulation studies. We start by evaluating and comparing the performance of SMT prefetching with single-threaded software prefetching. Next, we compare the overall instruction count and throughput of all the prefetching schemes. Finally, we present the experimental results on the basis of which we decide the offset value of our fetch priority scheme.

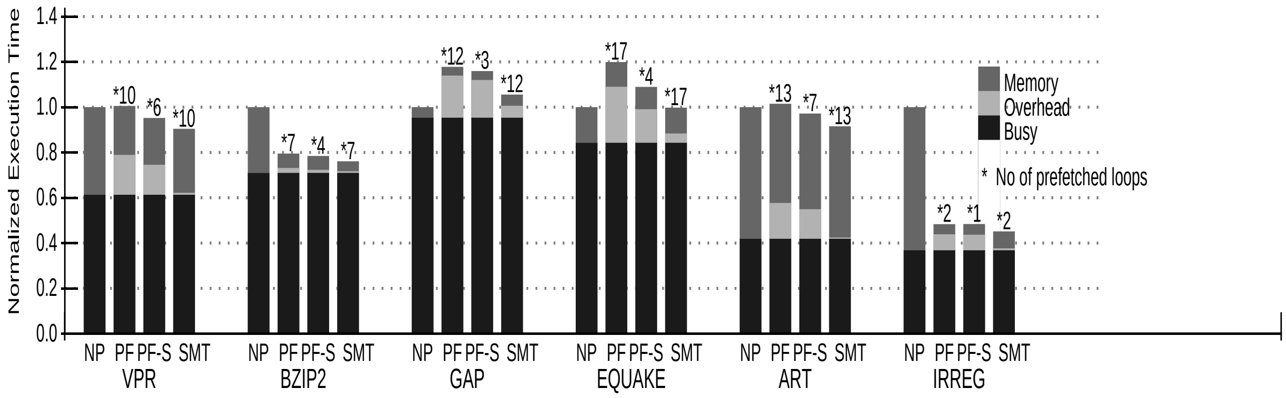

Figure 11: Performance comparison of No Prefetching(NP), Single-Threaded Software Prefetching(PF) (All loops), Single-Threaded Selective Prefetching(PF-S) and SMT Prefetching(SMT) (All loops)

5.1 SMT Prefetching - Evaluation and Comparative Study

Figure 11 presents the performance results for the different prefetching schemes. The different bars for each of the applications represent the baseline no prefetching case (NP) , single-threaded software prefetching when applied to all the loops without profile information (PF), single-threaded selective software prefetching (PF-S) based on the selectivity criterion mentioned in Section 4, and SMT prefetching (SMT) applied to all loops, without any profile information. The label (*) on top of each bar represents the number of loops for which prefetching is applied. We see that there is a significant reduction in the number of prefetched loops from the all-prefetching to selective prefetching case. The bars in the graph are split into three components, labeled *"Memory","Overhead", and "Busy"*. These components represent the time spent stalled on instruction and data memory accesses (*Memory*), time spent executing the overhead instructions (*Overhead*), and time spent executing useful instructions (*Busy*) respectively.

5.1.1 Comparison with Single-Threaded Software Prefetching - No profiling

As we see in Figure 11, SMT prefetching applied to all loops outperforms single-threaded software prefetching in all the applications. Single-threaded software prefetching without any profiling information degrades performance in 4 out of the 6 applications. On the other hand SMT prefetching, degrades performance in only one of the applications (gap), inspite of a significant reduction in the overhead, over single-threaded software prefetching. Gap has very little memory stall and hence is very sensitive to any additional instruction overhead. SMT prefetching not only introduces very

	Execution Time Gain			Prefetch Overhead		
	PF	PF-S	SMT	PF	PF-S	SMT
Vpr	-0.54%	4.76%	9.55%	28.73%	21.60%	1.41%
Bzip2	20.49%	21.54%	23.91%	3.08%	1.88%	1.06%
Gap	-17.80%	-15.94%	-5.56%	19.50%	17.47%	5.53%
Equake	-19.92%	-8.953%	0.21%	29.34%	17.57%	4.87%
Art	-1.50%	2.81%	8.44%	37.72%	31.17%	1.34%
Irreg	51.65%	51.59%	54.85%	19.16%	18.69%	2.36%

Table 3: Simulation Results - Performance Gain and Instruction Overhead (PF - Single-Threaded Software Prefetching (All loops) , PF-S - Single-Threaded Selective Prefetching, SMT - SMT Prefetching (All loops))

negligible overhead, it also achieves impressive gains.

When SMT prefetching is naively applied to all loops in the 5 *Spec* applications, it offers a 7.31% performance gain over no prefetching. The additional performance gains achieved by SMT prefetching, over single-threaded software prefetching is due to an 88.0% reduction in prefetch overhead. Figure 11 also shows that even when single-threaded software prefetching achieves significant performance gain, as in the Irreg benchmark, SMT prefetching still performs better because of reduction in overhead.

5.1.2 Comparison with Single-Threaded Selective Software Prefetching

Figure 11 shows that overhead is high when single-threaded software prefetching is applied universally to all the loops. Hence, we need to use detailed cache-miss profile information and off-line analysis to selectively prefetch for loops, which are memory intensive. When detailed profile information is used to selectively prefetch loops with high cache-miss counts, single-threaded software prefetching degrades performance in only 2 out of the 6 applications. Overall, single-threaded software prefetching achieves a 0.70% performance gain across the 5 *Spec* benchmarks, which is still 6.60% less than naive SMT prefetching applied to all the loops.

SMT prefetching, applied universally to all the loops still performs better than single-threaded selective software prefetching in all the applications. SMT prefetching achieves a 68.38% reduction in overhead across three integer applications and a 85.15% reduction in three floating point applications over single-threaded selective software prefetching. In terms of overall performance gain, SMT prefetching offers a 63.74% speedup over single-threaded selective software prefetching.

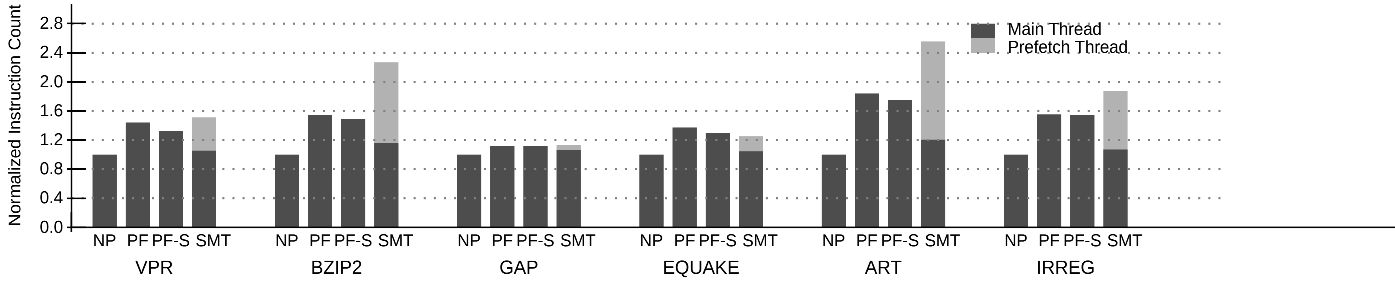

Figure 12: Effective instruction count comparison of No Prefetching(NP), Single-Threaded Software Prefetching(PF) (All loops), Single-Threaded Selective Prefetching(PF-S) and SMT Prefetching(SMT) (All loops)

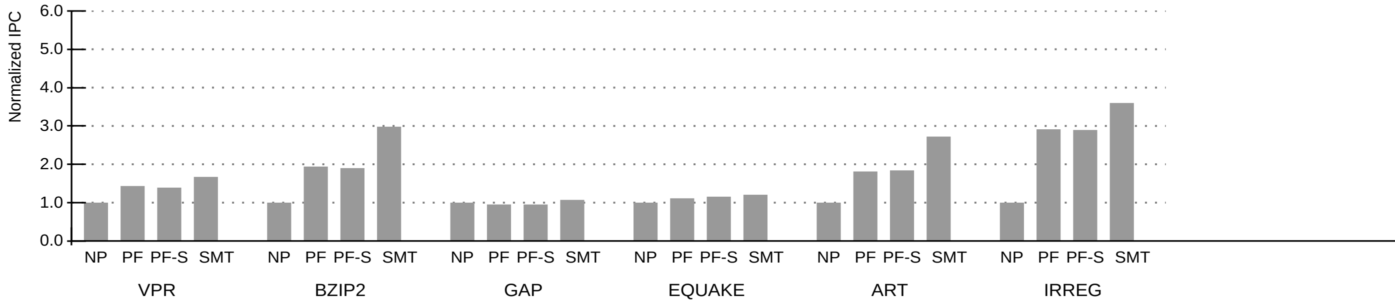

Figure 13: IPC comparison of No Prefetching(NP), Single- Threaded Software Prefetching(PF) (All loops), Single-Threaded Selective Prefetching(PF-S) and SMT Prefetching(SMT) (All loops)

5.1.3 Comparison Study - Instruction Count and Throughput

Figure 12 gives the overall instruction count for all the prefetching schemes in our applications, normalized to the no prefetching case. The different bars for each of the applications represent the different prefetching schemes as mentioned in Section 5.1. The bar representing the SMT prefetching scheme is further split into main thread instructions and prefetch thread instructions. The above comparative study gives insight into the effect of the polling synchronization technique on the overall instruction count in SMT prefetching.

As we see in Figure 12, the overall instruction count for SMT prefetching and single-threaded software prefetching are almost identical in applications such as Vpr, Gap and Equake. Vpr and Gap have very short loops with the number of iterations almost equal to the prefetch distance. Since the counter in the main loop (refer to Section 3.2.2) is initialized to the prefetch distance, the prefetch thread almost never busy waits. Equake does not require any synchronization as the loop bounds are compile-time constants, and are less than the prefetch distance. Bzip2, Art and Irreg have loops that have a very large number of iterations, and hence the prefetch thread has to poll

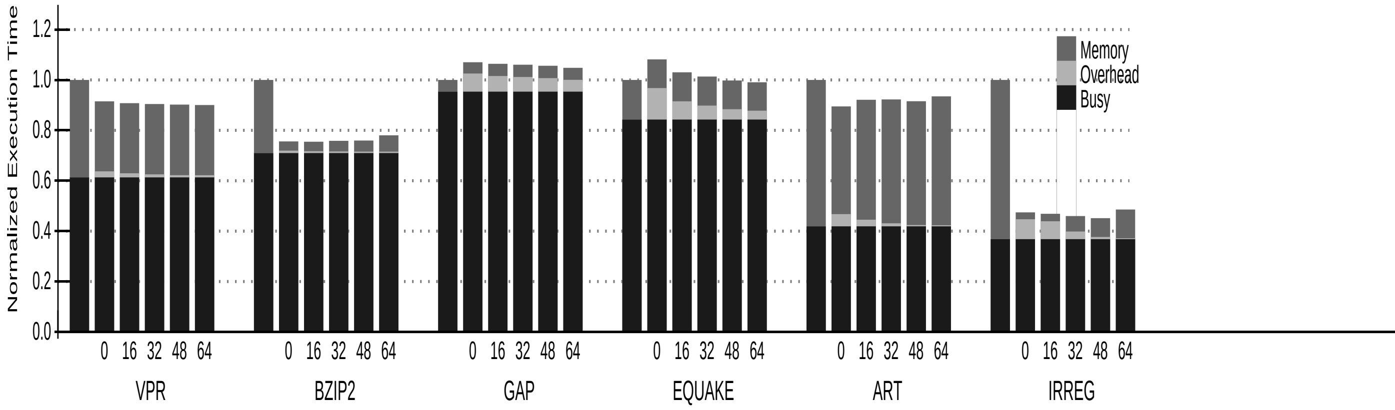

Figure 14: Performance comparison as a function of main thread fetch priority by varying the offset value

on the synchronization primitive every time it tries to get further than the prefetch distance. This is reflected by the increased instruction count in these applications. But these polling instructions do not impact performance due to the fetch and issue priority schemes. Figure 13 shows the normalized IPC for the different prefetching schemes. The graphs reflect our earlier reasoning of effective utilization of unused slots by the prefetch threads, and hence a greater throughput for SMT prefetching.

5.2 Optimizing the Main thread Priority

The magnitude of bias in favor of the main thread by the fetch priority scheme is dependent on the offset value. A low offset value will lead to the helper threads stealing main thread fetch bandwidth, and hence impacting the main thread. On the other hand, a very high offset value might starve the helper threads, and hence prevent them from making forward progress. Figure 14 presents the performance results of SMT prefetching with different offset values added to the ICOUNT of the prefetch threads (refer to Section 2.2.1). We see that the overhead consistently reduces in all applications as we increase the offset. However, in applications such as Bzip2, Art and Irreg, the data stall coverage reduces beyond a certain bias. Based on the results in Figure 14, we used an offset value of 48 for all our experiments.

6 Related Work

Several studies have examined the use of helper threads on SMT processors to assist the main thread execution. Techniques to execute exception handlers on idle threads [5] in parallel with the main thread have been used to overcome the main thread delay caused by serialized exception handlers. Helper threads have also been used in applications such as speeding up the execution of critical instructions that lead to stalls in the main thread [6], and improvement of the main thread branch prediction accuracy [7].

Significant work has also been done in pre-execution techniques, in which an SMT processor uses one or more helper threads to execute a subset of the original program in front of the main thread [16, 17, 18, 19]. These helper threads are used to trigger cache misses ahead of the main thread so that they effectively hide the latency of the cache misses.

Our paper focuses on reducing the impact of these helper threads on the execution of the main thread. Our technique uses fetch and issue priority schemes to dynamically schedule the helper thread instructions in the free slots of the main thread. We apply our low overhead helper threads to software prefetching and demonstrate the increased performance gains over single-threaded software prefetching.

Researchers have looked into ways to reduce the overhead due to prefetch instrumentation. Selective prefetch instrumentation using correlation profiling has been proposed as a solution to reduce prefetch overhead [9]. Hardware prefetching is another technique which introduces almost zero-overhead on the original program [12, 14], but at the expense of additional hardware. Our technique, which uses an SMT processor with minimal hardware support, enables us to naively instrument prefetches to all loops without any profile information.

7 Conclusion

Software based processor enhancements techniques, inspite of their advantages, are limited by their impact on the execution time of the original program. In this paper, we propose a low-overhead technique to enable an SMT processor to execute software instrumentation code on separate helper threads. We demonstrate the usefulness of our technique by performing software prefetching in separate helper threads. We also show that unlike conventional single-threaded software prefetching, SMT prefetching can be applied naively to all loops without any profile information.

Software-assisted performance enhancement techniques, such as software-managed caches, software prefetching, and informing memory operations, offer several advantages over hardware based techniques. Current software-based techniques are limited by the instruction overhead they introduce in the path of execution of the original program. Our results across 5 *SPEC* benchmarks show that when we apply conventional single-threaded software prefetching with detailed profile information to guide prefetch instrumentation, the overall performance gain increases to 0.70%. Naive SMT prefetching, on the other hand, achieves a 7.31% performance gain in the same set of applications. Our results also show that even when conventional single-threaded software prefetching achieves significant performance gain (eg: Irreg), SMT prefetching still performs better.

Our low-overhead helper threads have enormous potential in techniques such as software-managed caches, software prefetching and informing memory operations. With minimal hardware support, we enable these low-overhead helper threads to assist the execution of the original program. In future work we plan to investigate the application of our low-overhead helper threads to other software based techniques.

References

[1] Joel S. Emer. Simultaneous Multithreading: Multiplying Alpha Performance. Microprocessor Forum, October 1999.

[2] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous Multithreading: Maximising On-chip Paralellism. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 392-403, Santa Margherita Ligure, Italy, June 1995. ACM.

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

[4] 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.

[5] Craig B. Zilles and Joel S. Emer and Gurindar S. Sohi. The Use of Multithreading for Exception Handling. Proceedings of the 32nd International Symposium on Microarchitecture, pages 219-229, November 1999.

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

[7] Robert S. Chappell and Sangwook P. Kim and Steven K. Reinhardt and Yale N. Patt. Simultaneous Subordinate Microthreading (SSMT). In Proceedings of the 26th International Symposium on Computer Architecture, pages 186-195, May 1999.

[8] A. R. Lebeck, D. R. Raymond, C. Yang, M. S. Thottethodi. Annotated Memory References: A Mechanism for Informed Cache Management. Euro-Par '99, August 1999

[9] T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. Technical Report CMU-CS-97-175, Carnegie Mellon University, September 1997.

[10] S. G. Abraham and B. R. Rau. Predicting load latencies using cache profiling. Technical Report HPL-94-110, Hewlett-Packard Company, November 1994.

[11] T. Mowry , M. S. Lam, and A. Gupta. Design and evalutaion of a compiler algorithm for prefetching. In Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 62-73, 1992.

[12] Tien-Fu Chen. An Effective Programmable Prefetch Engine for On-Chip Caches. In Proceedings of the 28th Annual Symposium on Microarchitecture, pages 237–242. IEEE, 1995.

[13] J. Gregory Steffan, Christopher B. Colohan and Todd C. Mowry. Architectural Support for Thread-Level Data Speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[14] 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.

[15] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing Loads: Enabling Software to Observe and React to Memory Behavior. Stanford University Technical Report CSL-TR-95-673 (also numbered STAN-CS-95-675), July 1995.

[16] 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.

[17] Dongkeun Kim and Donald Yeung. Using Program Slicing to Drive Pre-Execution on Simultaneous Multithreading Processors. University of Maryland Institute for Advanced Computer Studies. UMIACS-TR-2001-49.

[18] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, John P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. In 28th International Symposium on Computer Architecture, July, 2001.

[19] Craig B. Zilles and Gurindar S. Sohi. Execution-based Prediction Using Speculative Slices. In 28th International Symposium on Computer Architecture, July, 2001.