

Runtime Support for Integrating Precomputation and Thread-Level Parallelism on Simultaneous Multithreaded Processors

Tanping Wang

Filip Blagojevic

Dimitrios S. Nikolopoulos

Department of Computer Science
The College of William and Mary
McGlothlin-Street Hall
Williamsburg VA 23187-8795
{twang,filip,dsn}@cs.wm.edu

ABSTRACT

This paper presents runtime mechanisms that enable flexible use of speculative precomputation in conjunction with thread-level parallelism on SMT processors. The mechanisms were implemented and evaluated on a real multi-SMT system. So far, speculative precomputation and thread-level parallelism have been used disjunctively on SMT processors and no attempts have been made to compare and possibly combine these techniques for further optimization. We present runtime support mechanisms for coordinating precomputation with its sibling computation, so that precomputation is regulated to avoid cache pollution and sufficient runahead distance is allowed for the targeted computation. We also present a task queue mechanism to orchestrate precomputation and thread-level parallelism, so that they can be used conjunctively in the same program. The mechanisms are motivated by the observation that different parts of a program may benefit from different modes of multithreaded execution. Furthermore, idle periods during TLP execution or sequential sections can be used for precomputation and vice versa. We apply the mechanisms in loop-structured scientific codes. We present experimental results that verify that no single technique (precomputation or TLP) in isolation achieves the best performance in all cases. Efficient combination of precomputation and TLP is most often the best solution.

1. INTRODUCTION

Since the introduction of simultaneous multithreaded processors in mainstream computing [3, 4], two forms of execution have been investigated to leverage multithreading: thread-level parallelization (TLP) and speculative precomputation. TLP amounts to parallelizing a program, either

manually or with the assistance of a compiler, and assigning different threads to different hardware execution contexts in the processor [18, 19]. TLP can accelerate the execution of a program by taking advantage of multiple execution units and higher ILP, as well as by hiding memory latency. Speculative precomputation (SPR) amounts to having one of the threads in the processor perform software prefetching to hide memory latency and eliminate as many cache misses as possible in the other simultaneously executing thread. Speculative precomputation can be effected in a number of ways, the most common of which is to replicate the code of the main computation thread in an unused thread and strip out all instructions except the delinquent loads that are likely to miss in the cache and the instructions upon which delinquent loads depend [5, 6, 21].

The research presented in this paper is motivated by the following observations:

a) There have not been direct comparisons between the two multithreaded execution modes (TLP and SPR) with parallel codes running on SMT and multi-SMT systems. Such a comparison would indicate which technique is the most appropriate for any given program running on a SMT processor. Recent experimentation with the Intel Hyperthreading processor and simulated processor cores has provided indications that SPR is effective for sequential, pointer-based codes and a few scientific codes with indirect array accesses, whereas TLP is a natural choice for regular scientific codes. Both techniques can yield speedup that ranges between 1.1 and 1.4 on a two-thread execution core [5, 18, 19]. Whether SPR can improve performance further if used in conjunction with TLP remains an open question.

b) SPR and TLP could be used conjunctively in the same program, given proper synchronization and coordination mechanisms. This option may be advantageous in several cases. A program may proceed through phases with different execution properties. Some of these phases may be sequential, in which case they may benefit from SPR but not from TLP. Other phases may be parallel, thus benefiting more from TLP. Yet other phases may be parallel, but their parallel execution may impose conflicts and queuing

in shared resources (such as execution units and caches), making SPR a potentially better candidate, since it is less resource-demanding.

The contribution of this paper is a set of runtime support mechanisms that enhance SPR and enable flexible and efficient use of SPR and TLP in the same program. We currently target these mechanisms to scientific codes with loop-intensive execution structure. The main innovation in these mechanisms is that they provide capabilities to coordinate accurately precomputation and sibling computation and to switch between precomputation and computation and vice-versa in the same thread at runtime. We present a protocol for coordinating precomputation and computation threads and trigger precomputation across basic block boundaries to improve its timeliness. We also present a task queue mechanism which synchronizes precomputation and computation and allows any thread to switch role (from SPR to TLP and vice versa) and control the distance between precomputation and sibling computation.

We have implemented these mechanisms in a runtime system which we customized for Intel processors with Hyperthreading technology. We tested the mechanisms on a multiprocessor server with four Hyperthreading processors. Our results show that neither SPR nor TLP can always achieve the highest performance and that combination of SPR and TLP is the most effective solution in many cases. Although the execution time improvements we observed are modest in absolute terms (integration of TLP and SPR outperforms TLP alone by 11% and SPR alone by 8% on average), we consider them as noteworthy given the architectural limitations of the processor we used for experimentation. Considering also the experiences reported so far from experiments with the same processor [5, 18, 19], we conclude that our mechanisms achieve significant improvements over previous work.

The rest of this paper is organized as follows: Section 2 provides further motivation and outlines the main ideas behind our runtime support mechanisms. Section 3 discusses the runtime mechanisms in more detail. Section 4 presents a sample of our experimental results and Section 5 concludes the paper.

2. MOTIVATION

Section 2.1 gives a brief overview of related work on multithreading and prefetching on processors with multiple execution contexts. The following section discusses the potential limitations of TLP on current SMT processors. Using microbenchmarks, we show how conflicts between threads on shared resources can nullify the benefits of TLP. Due to these limitations, SPR may be a better alternative, even in programs with a seemingly highly parallel structure. Section 2.3 describes schematically schemes that can increase the effectiveness of SPR and help utilize SPR and TLP in the same program.

2.1 Related Work

Several papers have presented static and dynamic schemes for software SPR on SMT processors [5, 6, 7, 9, 21]. A compiler-assisted implementation of SPR was used in these studies. A compiler identifies either statically or with the assistance of a profile the memory loads that are likely to

cause cache misses with long latencies. These loads are called *delinquent loads*. Once delinquent loads are identified, the compiler generates a helper thread that executes the delinquent loads and the instructions that produce their addresses, if any. Precomputation is controlled by triggering the helper thread while the main computation thread is running. In early implementations, the helper thread was triggered to run continuously, however more recent work [6] has shown that periodic triggering and throttling of the precomputation thread achieves less intrusive and more efficient SPR. SPR may also be implemented using sophisticated hardware mechanisms [2, 15, 16]. Current processors do not have such a capability.

SPR yields measurable performance improvements in codes which are dominated by pointer chasing and irregular memory access patterns. A pencil and paper comparison between TLP and SPR shows that on two-thread execution cores, neither SPR nor TLP is clearly more effective [19, 21]. TLP has some advantages on architectures in which individual threads get sufficient resources to execute with high ILP [13]. However, on SMT processors, most of the hardware resources are shared and resource limitations reduce the potential speedup from parallel execution. Section 2.2 elaborates more on this issue.

Although direct comparisons between SPR and TLP on SMT processors have not appeared in the literature, an earlier study which investigated the limits and operational areas of multithreading and prefetching in parallel programs [8], had a similar objective. This study was conducted on a medium-scale shared-memory multiprocessor and indicated that there are conditions under which multithreading is preferable to prefetching and vice-versa. The study assumed the hardware and latencies of shared-memory multiprocessors at the time (1996), therefore it is difficult to adopt its conclusions on modern hardware without further investigation. The architectural properties of state-of-the-art SMT processors and the multithreaded processors studied in [8] differ in many aspects. SMTs are coarse-grain multithreaded processors, in which threads are created and managed entirely in software. The processors used in [8] supported extremely fine-grain multithreading with thread switching implemented in hardware and triggered upon any long-latency event (such as a cache miss or a branch misprediction). The context switching latency of SMTs amounts to thousands of processor cycles, whereas fine-grain multithreading processors switch threads in a handful of cycles.

Research on compiler algorithms for prefetching [10] has considered the use of multithreading in conjunction with prefetching, but did not provide mechanisms and algorithms to achieve effective integration of the two techniques. The initial experimental results of this work did not indicate any benefit in naïvely combining multithreading and prefetching, but characterized the integration of the two techniques as a viable option [10, 11]. One of the objectives of the work presented in this paper is to investigate whether multithreading and prefetching can be integrated using mechanisms that employ both TLP and SPR in the same processor core.

Speculative TLP [1, 13, 15] is another hybrid form of TLP, in which threads are used for parallelization of all itera-

	FMUL	FDIV	FADD	FSUB
FMUL	7.0	7.0	7.2	7.1
FDIV	43.1	86.0	43.2	43.1
FADD	5.5	5.3	5.1	5.1
FSUB	5.5	5.3	5.1	5.1

Table 1: Average number of cycles per floating point operation using only registers and no data loaded from memory. Latencies that indicate conflicts between threads are shown in boldface.

tive structures in the program, regardless of whether they are statically parallelizable. Speculative TLP requires additional hardware support for rollbacks, whenever speculation proves to be wrong. The key to effective speculative TLP is the minimization of rollbacks. Compared to speculative TLP, SPR is a less intrusive form of speculation, which also poses less requirements on the hardware. This work considers the integration of SPR with non-speculative TLP and leaves the investigation of SPR in conjunction with speculative TLP as future work. We also perform physical experimentation with SPR and TLP on real processors, to draw optimization guidelines that will benefit programmers and compiler designers.

2.2 Limitations of Thread-Level Parallelism

It is not without reason that TLP can not provide easily linear speedup on SMT processors. Since threads share a significant amount of state on the processor (including caches, execution units, and instruction queues), they can not leverage the full potential of application-level parallelism because of conflicts and queuing on shared resources.

We use two microbenchmarks, briefly referred to as MB1 and MB2, to illustrate the impact of conflicts between threads on an Intel Xeon MP processor. This processor uses Intel’s Hyperthreading technology, which is largely based on simultaneous multithreading [20]. MB1 forks two threads, and each thread performs a stream of 100 million floating point operations. Each floating point operation uses the same set of registers from the floating point stack. The processor has two sets of floating point registers, one for each thread. The threads can be configured so that the two streams execute the same (e.g. FMUL/FMUL) or different (e.g. FMUL/FADD) operations. Care is taken so that when threads issue multiplications and divisions, no overflows occur. There is no sharing of data or other forms of dependencies between threads.

Table 1 shows the average number of cycles per floating point operation, as measured with MB1. The four columns show the number of cycles per floating point operation, when two threads issue operations concurrently. Note that all combinations of floating point operations can proceed concurrently without conflicts, except from the combination FDIV/FDIV. When both threads attempt to use the division unit, threads are sequentialized and the division latency is doubled from 43 cycles to 86 cycles.

MB2 is identical to MB1, with the exception that each stream of floating point operations is issued to a vector of 250000 doubles (2 MB of data). Each thread uses a private

	FMUL	FDIV	FADD	FSUB
FMUL	63.7	77.3	63.8	62.8
FDIV	80.6	96.3	81.3	80.2
FADD	63.4	75.2	64.1	64.3
FSUB	64.7	77.0	67.9	63.0

Table 2: Average number of cycles per floating point operation using doubles loaded from memory. Latencies that indicate conflicts between threads are shown in boldface.

vector and executes 400 iterations walking over the entire vector. Note that the arrays are reused but do not fit in the cache of the processor, at any level. Table 2 shows the cycles per floating point operation, when threads are loading data from memory into floating point registers. The results indicate that queuing of threads happens when one thread tries to use the floating point unit, while its sibling performs divisions. In this case threads are slowed down by almost 20%, despite the independence of their operations.

Another indication of the impact of sharing resources on the SMT processor can be observed by measuring cache performance. We executed two versions of MB2, one in which each thread accesses a private vector and a second in which both threads access the same vector, thus sharing data from the cache. The version with thread-private vectors incurred 7% to 20% more L1 cache misses than the version with the thread-shared vector, when the vector sizes exceeded 3 Kilo-bytes. The increase in the cache misses happens due to inter-thread conflicts in the L1 cache. Conflicts occur even when the working set sizes of the two threads combined do not exceed the size of the L1 cache, a problem attributed in part to the memory allocator, which neglects the implications of sharing the cache between threads.

Our microbenchmarks have exemplified several cases in which conflicts between threads on shared resources incur significant performance penalties. Real programs have exhibited modest efficiencies of 55–70% (speedups of 1.1 to 1.4) on Intel’s 2-way Hyperthreading processors [19], primarily due to conflicts. Interestingly, the TLP efficiencies reported for scientific codes tend to be lower than those reported for other benchmarks, such as desktop and server workloads. In general, the Hyperthreading processors tend to exhibit better and more predictable performance with multiprogram workloads, rather than with parallel workloads. The non-complementarity of the resource requirements of threads in parallel workloads incurs contention for shared resources and eventually lower performance.

2.3 Flexible Multithreaded Execution Modes

The limitations of resource sharing may degrade significantly the performance of TLP and make SPR a better alternative. Precomputation, if engineered properly, requires less resources and has the additional advantage of being usable in sequential codes. On the other hand, precomputation targets only memory latency and does not exploit parallelism in the execution units of the processor. Therefore, SPR must be avoided or used with caution in codes with high parallelism and good data locality.

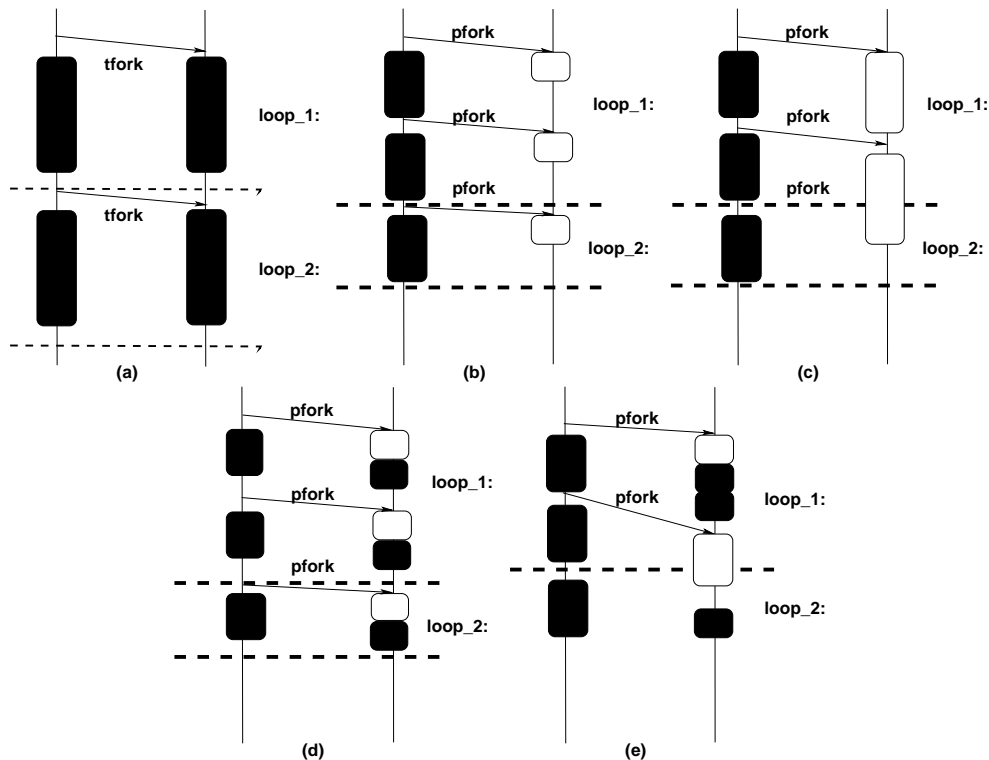


Figure 1: Alternatives for using SPR and TLP on a 2-way SMT processor. Computation chunks are shown as black boxes and SPR chunks are shown as white boxes. The dashed lines indicate barriers between parallel loops. The *tfork* calls indicate computation thread spawns. The *pfork* calls indicate triggers of precomputation threads, which can be either thread spawns or, more frequently, thread wake-up calls.

Figure 1 illustrates some multithreaded execution modes that can be used to run a single program on a SMT processor with two execution contexts. For purposes of illustration we assume that the two threads execute parallel loops, although the techniques discussed here apply to other cases with some modifications. Case (a) illustrates the standard TLP mode of execution, whereas case (b) illustrates the standard SPR mode of execution. Typically, with SPR, the second thread is utilized only a fraction of the time, as much as needed to issue the delinquent loads. If the loop has a large enough trip count, the precomputation thread can run ahead of the main computation thread and prefetch data early so that the computation thread suffers less cache misses. As shown in Figure 1(b), SPR may need to be regulated so that there is no excessive prefetched data that evict useful data from the cache. In such cases, precomputation is decomposed into chunks, each of which is triggered separately at computation points distanced appropriately from each other. The first runtime mechanism presented in this paper aims at regulating precomputation, by controlling the runahead distance between the precomputation and the sibling computation thread, as well as the amount of data fetched with precomputation. Precomputation may also span multiple loops, as shown in case (c) in Figure 1, under the rationale that the runtime system can leave sufficient runahead distance between precomputation and computation in some cases. The trade-off in this SPR scheme is to select a distance large enough to prefetch timely but small enough to avoid cache pollution and interference with earlier computation.

The second runtime mechanism presented in this paper attempts to combine SPR and TLP by allowing both threads to switch roles at runtime. Examples of this technique are depicted in case (d) and case (e) of Figure 1. In case (d), a thread is assisting its sibling by first prefetching data and then participating in the execution of the main computation. In case (e), a similar technique is applied for precomputation across loop boundaries. The second set of runtime mechanisms presented in this paper achieves integration of SPR and TLP.

The goal of integrating SPR and TLP is to utilize better the execution contexts of the processor and achieve higher speedup, by taking advantage of latency hiding and parallel execution. Some of the mechanisms presented here can improve SPR alone, while others can improve both SPR and TLP. Programs with sequential and parallel sections can take advantage of SPR during sequential execution and TLP during parallel execution. Irregular applications with inherent load imbalance, or even regular applications that suffer from load imbalance due to unpredictable resource conflicts in the processor, can benefit from the flexibility of these mechanisms, by prefetching during idle periods. We explore some of these optimization opportunities in this paper, hoping to open up a new direction of software optimizations for multithreading processors in the near future.

3. RUNTIME SUPPORT

The following discussion assumes a two-thread SMT processor. To implement SPR, we used the scheme suggested in [5], in which the precomputation thread is derived from a replica of the sibling computation thread, by keeping only delinquent loads and instructions that generate their addresses. We have used code profiling and a cache simulator that simulates the cache of our target processor to identify these loads. Precomputation is triggered either before or within loops (depending on the runtime mechanism used) and its targeted scope may cross the boundary of the loop in which it is triggered. This is a central difference compared to previous work and enables more timely precomputation, as well as precomputation for loops that use TLP. We present a runtime protocol used for regulating SPR, followed by a mechanism designed to combine SPR and TLP.

3.1 Regulating Precomputation

Precomputation needs to function like any effective prefetching mechanism. The precomputation thread must run ahead of the sibling computation thread, so that data is prefetched in the cache before the compute thread uses it, but without evicting other useful data. The first task of the runtime system is to provide a mechanism to guarantee that the precomputation thread runs sufficiently ahead of the sibling computation thread and prefetches data timely. The common practice used in current precomputation tools is to start the precomputation thread at the beginning of a loop and keep it running until it finishes precomputing for all iterations. This naïve approach has two problems: the first is that in loops with large working sets, the precomputation thread may run too far ahead from the computation thread and start polluting the cache with data before the sibling computation thread can actually use the data that were fetched earlier. The second problem is opposite to the first. There may not be enough time for the precomputation thread to run sufficiently ahead of its sibling computation thread.

Our runtime mechanisms regulate both the amount of data fetched in precomputation phases and the distance between precomputation phases and sibling computation. They also enable the overlap of precomputation with computation in earlier phases. An illustration of the mechanisms is given in Figure 2. Both precomputation and sibling computation are split in chunks, which may correspond to subsets of individual loop iteration spaces, or unions of complete or partial loop iteration spaces. The span of a precomputation chunk (denoted with $s(pc_{ij})$) in Figure 2 is selected so that the amount of data prefetched does not exceed a predefined threshold. Once a target data set size for precomputation is fixed, the computation covered by the span is split into corresponding phases, with each phase having a memory footprint as large as the amount of data fetched by the enclosing precomputation chunk. Within phases, computation can be further split into chunks, to give the runtime system opportunities for triggering further precomputation and overlap the latency of the SPR triggers with earlier computation.

We discuss the issue of delimiting a precomputation span based on the cache misses of sibling computation in Section 3.1.1. We then discuss the issue of decomposing computation to create new trigger points for precomputation in Section 3.1.2.

3.1.1 Selecting a Precomputation Span

A natural span for precomputation is a sequence of dynamic instructions that has a memory footprint equal to the size of the L2 cache. In most cases, prefetchers target the outermost level(s) of cache hierarchies. Precomputation can therefore target one of these levels by using a span with a footprint equal to the corresponding cache size. A more refined option is to consider smaller precomputation spans, taking into account the conflict misses that happen due to limited cache associativity. The rationale is that although a certain computation fragment may have a small memory footprint, the amount of data fetched from the lower level of the memory hierarchy may be large due to conflicts. Unless the precomputation thread has chances to re-fetch data evicted due to conflicts, precomputation can not be very efficient. Following this rationale, we select precomputation spans with a memory footprint equal to a fraction $\frac{1}{S}$ of the targeted cache size, where S is the degree of associativity.

Given an initial threshold for the amount of data brought in with precomputation, the computation enclosed by the span is defined using a profile of cache misses in the targeted code. The memory footprint is defined as all the distinct cache lines that a specific dynamic sequence of instructions fetches from memory. Once the computation enclosed in a precomputation span is identified, a precomputation thread is constructed from the instructions that incur cache misses in the span and the slices of code that produce the effective addresses of these instructions.

In a given precomputation span, some cache lines may be fetched from memory multiple times, due to conflict misses. Should this be the case, the precomputation span is split further in half. The memory footprints of the new half-sized spans are equal to or smaller than the memory footprint of the original unified span. However, if a cache line happens to be evicted in both half-sized spans, the precomputation threads have a chance to fetch this cache line twice, once in each span, and save a conflict miss. Splitting of precomputation spans can continue recursively to eliminate more conflict misses. Splitting needs to be throttled, since many small spans will require a large number of precomputation triggers, the overhead of which is significant. We use a simple cost-benefit criterion to resolve this problem and stop splitting once the amount of data fetched in a precomputation span goes below a threshold.

3.1.2 Computation Decomposition

In current implementations, precomputation starts always concurrently with sibling computation. The anticipated behavior is that the precomputation thread will run sufficiently ahead of the sibling computation thread. Practical prefetching algorithms though require that prefetching starts at a fixed distance ahead of the targeted computation [10]. In the case of SPR on a multithreaded processor, regulating the distance between precomputation and computation requires the runtime system to have some mechanism to synchronize the precomputation and the computation thread. One option is to trigger the precomputation thread at the end of a previous span and start the sibling computation using another trigger from the precomputation thread. This option requires two thread triggers (spawns or wake-ups), which are too expensive on SMT processors. On our experi-

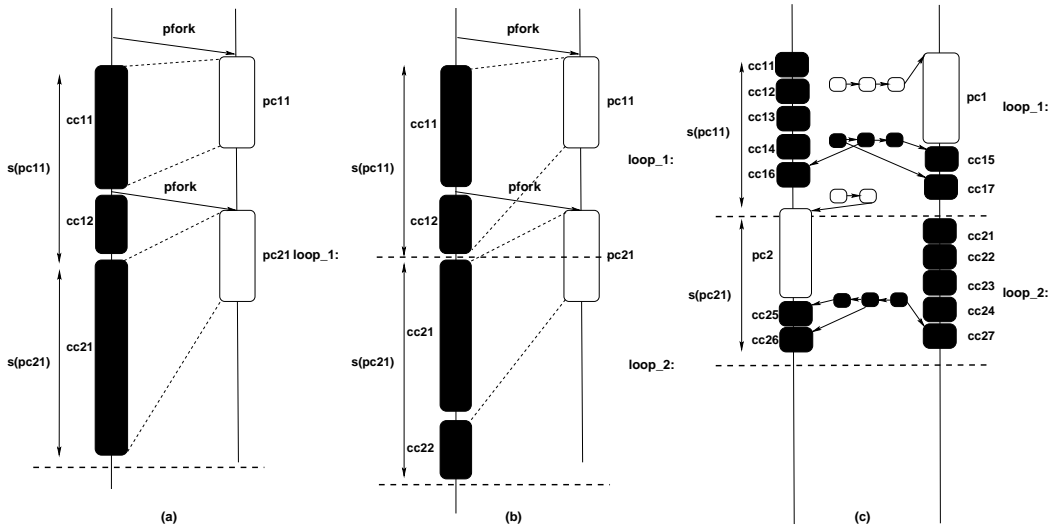


Figure 2: Mechanisms for regulating precomputation. Case (a) shows regulated SPR for a single loop. pc_{ij} represents a precomputation chunk j of loop i and cc_{ij} represents a computation chunk j of loop i . The sets of precomputation and computation chunks may not have the same cardinality. $S(pc_{ij})$ indicates the computation span targeted by precomputation chunk pc_{ij} . Rightbound arrows show triggers of SPR threads and horizontal dashed lines indicate barriers at loop exit points. Case (b) shows regulated SPR for two loops, with precomputation/computation overlap across the loops. Case (c) shows a task queue mechanism for integrating SPR with TLP.

mental platform, each thread creation/wake-up amounts to more than one thousand cycles.

We adopt an alternative strategy in which the precomputation thread can be triggered from within a previous span, as shown in Figure 2(b). To this end, we fix a runahead distance between a precomputation chunk and its span, which we measure in cache lines fetched upon misses. This runahead distance is calculated using the profile of the computation. Given the runahead distance for a precomputation span i , we identify a point in precomputation span $i - 1$, which is spaced logically by as many cache line fetches as the target runahead distance from subsequent computation. We split the precomputation span of $i - 1$ at the triggering point and insert a wakeup interrupt. Since we operate on loops, computation spans are defined as subsets of iteration spaces, or unions of subsets of iteration spaces. To effect splitting, we identify the latest loop nest iteration $i'_1, \dots, i'_n - 1$ before a triggering point using the profile and decompose the loop in two parts, one running from iteration $1, \dots, 1$ to iteration $i'_1, \dots, i'_n - 1$ and a second running from the following iteration to the end of the loop. The awoken thread polls a shared flag and commences precomputation when the flag is set. The mechanism uses one instead of two triggers and overlaps the latency of triggering a precomputation thread with computation in the earlier precomputation span. As such, it achieves more efficient control of precomputation.

3.1.3 Implementation Details

In our current prototype we detect precomputation spans using an execution-driven cache simulator for x86 executables derived from Valgrind [12]. We use complete instruction traces from the simulator to: identify dynamic instruction sequences that bring as much data as the target thresh-

old for precomputation; identify the data fetched and map them back to memory accesses in the source code; isolate these memory accesses and issue them in a single batch in precomputation threads; identify trigger points for precomputation threads and insert wake-ups. In this implementation, precomputation threads release the processor once they are done so that all held resources are made available to sibling computation threads. This optimization is vital for SMT processors. We are currently using alarm signals and a shared flag for controlling the precomputation threads. Alternatively, we could use the privileged `halt` instruction and an interrupt inside the operating system. Both mechanisms are very expensive in processor cycles. Unfortunately, they are mandated by limitations of the operating system and lack of more efficient synchronization instructions on the Intel Hyperthreading processors.

3.2 Combining SPR and TLP

A task queue model is used to combine SPR with TLP in a single precomputation span. In this scheme, both the computation and the precomputation thread execute loops by chunking. Chunks are obtained from two thread-shared work queues, one used for computation and the other used for precomputation, as shown in Figure 2(c). The term *queue* is used only for illustrative purposes. Since the runtime system schedules loops, a *queue* is actually a synchronized trip counter rather than an actual linked data structure.

Any thread running on an execution context of the processor can pick chunks of computation or precomputation from any of the two queues. The precomputation queue can be temporarily deactivated to allow for TLP execution without interference. A thread completing a precomputation chunk

can obtain a chunk of computation for the same span, or a chunk of precomputation for a subsequent span.

3.2.1 Execution Mechanism

The code is setup to use worker threads bound to different execution contexts on the same processor. Precomputation and computation chunk sizes are selected offline using profiling. Chunk size selection is discussed in more detail in Section 3.2.2. Chunks are inserted in the queues by any of the worker threads. Computation chunks are inserted upon initiation of loops by the runtime system. Precomputation chunks are inserted either upon initiation of loops, or from within a loop in an earlier precomputation span, if precomputation is set to be triggered so that it has a minimum runahead distance from its span. The runtime system maintains that precomputation chunks are inserted before or simultaneously with the first chunk of the targeted computation span.

All computation and precomputation chunks are timestamped. There is a one to many correspondence between a precomputation chunk and computation chunks in its span. A precomputation chunk is timestamped with a unique integer ID τ and is described as a tuple $(\tau, S'_1, S_2, S_3, \dots, S'_k)$, where S_i corresponds to the iteration space of loop nest i and $S'_i \subseteq S_i$. Note that there might be just one nest in the tuple, meaning that the precomputation chunk spans only a subset of a loop iteration space or one full loop iteration space. The iterations of loops in the span of a precomputation chunk with timestamp τ are tagged with the same timestamp and are represented with a tuple (τ, i, C) , where i identifies a loop in the precomputation span and C is the chunk size used in the execution of the loop.

If no runahead distance is desired between a precomputation chunk and its span, the runtime system inserts one descriptor with the tuple $(\tau, S'_1, S_2, S_3, \dots, S'_k)$ in the precomputation queue and one descriptor with the tuple $(\tau, S_j | S'_j)$ for each distinct loop j covered by the precomputation span, in the computation queue. When a precomputation tuple is seen by a thread visiting the precomputation queue, the entire iteration space of it is executed and all delinquent loads in the span are issued. The tuple remains in the queue until the precomputation thread finishes executing the loads. When the computation queue is visited, only a chunk of size C from one of the loops in the span is dequeued and executed, if the precomputation span with timestamp τ is still in the precomputation queue. Otherwise, the thread aggressively dequeues and executes half of the remaining iterations of the loop. Similarly, if a precomputation thread finishes its task and sees the precomputation queue empty, it will proceed to steal half of the iterations remaining from the next loop in the computation queue. In other words, for each loop, the runtime system switches the chunk size and coarsens it, when the corresponding precomputation phase has finished.

A similar mechanism is used if a precomputation span covers multiple loops. The difference is that the runtime system inserts one tuple per partially, or fully covered loop in the precomputation span. Each tuple has the same timestamp τ and, initially, the same chunk size C . When the precom-

putation thread finishes fetching data for the span denoted by τ , it starts stealing work from the computation queue and the chunk size is set equal to half of the remainder iterations for the loop fragments pending execution. The same adjustment is done for all computation threads.

If a runahead distance needs to be established between precomputation and sibling computation, the precomputation thread is triggered as described in Section 3.1.2. To this end, we fix a distance d between precomputation and sibling computation, measured in cache misses, and “truncate,” this distance to an integer number of iterations in the last loop of the preceding precomputation span. This means that we find the minimum number of the latest loop iterations that incurs d cache misses in the preceding span. If S_r is the remainder of iterations in the preceding precomputation span, then $S_r - d$ iterations are scheduled with the workstealing strategy mentioned earlier. The runtime system inserts a tuple for the following precomputation phase in the precomputation queue and the remainder d iterations from the previous precomputation span are executed.

The dequeuing strategy is summarized as follows. A thread looks for work in both the precomputation queue and the computation queue. If τ_c and τ_p are the computation and precomputation timestamps of the tuples at the head of the two queues respectively, the thread makes the following checks:

- If there is only precomputation or only computation work scheduled, the thread proceeds with executing it. A precomputation chunk is scheduled as a unit. A computation chunk is bisected, unless the subsequent precomputation needs to run ahead of its span, in which case a chunk of d iterations is set aside as a remainder.
- If $\tau_1 = \tau_2$, the precomputation chunk is selected and executed.
- If $\tau_1 < \tau_2$, the computation chunk is selected and executed, unless the remaining iterations are less than or equal to the desired runahead distance (d) from the following precomputation span.

3.2.2 Selecting Chunk Sizes

The previous discussion suggested a dynamic adjustment of the chunk size based on whether there is precomputation work to be executed concurrently with sibling or following computation. The initial selection of a chunk size C is important since too small a C will increase overhead, whereas too large a C may limit the ability of the runtime system to re-balance the computation load between threads. For example, for a loop with 100 iterations, a chunk size of 50 will allow the second thread to steal work only if precomputation finishes before the execution of the first 50 iterations, whereas with a chunk size of 10, work stealing can start as late as 90 iterations into the loop.

We devised a simple model for selecting the computation loop chunk sizes in our runtime system. Assume that a loop nest (or a subset of it) which is in the span of a precomputation chunk has a trip count of N and the estimated time

to execute it is T_c . Let T_p be the time it takes a thread to complete precomputation for the specific computation span and during which precomputation overlaps with computation. It is expected that $T_p < T_c$. Assume that when the computation is multithreaded, it obtains a speedup of S_c . We assume that S_c is uniform across the computation, i.e. if two threads execute a subset of the N iterations they will still get a speedup of S_c . This assumption is simplifying but appears to work reasonably well in practice. Assume also that if a part of the computation finds its data prefetched in memory, it obtains a uniform speedup of S_p .

If the loop is executed in TLP mode without SPR, its expected execution time is $\frac{T_c}{S_c}$. If the loop is executed initially with a single computation thread overlapping with precomputation and subsequently with two threads in parallel, the expected execution time will be:

$$\frac{T_c - T_p}{S_c S_p} + \frac{T_p}{S_p} + \left\lceil \frac{N}{C} \right\rceil o$$

In other words, the part executed in parallel will benefit from both TLP and precomputation by obtaining a multiplicative speedup, whereas the sequential part will benefit only from precomputation. The last component in the equation accounts for the synchronization overhead for executing chunks. Figure 3 illustrates the impact of the chunk size on the performance of the code that uses precomputation and TLP simultaneously. The figure plots the normalized execution time of a hypothetical loop with a trip count of 100 iterations and normalized execution time $T_c = 1$. The hypothetical loop is assumed to be executed sequentially. The time is plotted against the ratio T_p/T_c , namely the percentage of time precomputation overlaps with computation and computation is executed sequentially. We assume two overhead values equal to 10% of the mean iteration time and 50% of the mean iteration time and four chunk sizes equal to 1, 2, 5 and 10% of the loop trip count. We assume speedups of 1.3 and 1.2 for TLP and SPR respectively. These values are obtained empirically from our own experiments and after studying the experimental results in several papers using the Intel’s Hyperthreading processors as their testbed [3, 5, 18, 19]. On a 1.4 GHz Xeon Hyperthreading processor, we measured the synchronization overhead to dispatch a chunk to approximately 200 cycles. The assumed overheads correspond to loops with run lengths between 400 and 2000 cycles per iteration. These are fine-grain loops for the specific platform, therefore we consider our estimates as conservative.

The curve labeled $o = 0$ in the charts corresponds to the ideal case in which chunking has zero overhead. This simulation shows that a chunk size equal to 10–20% of the trip count of the targeted loop is sufficient to yield higher performance than plain thread-level parallelization, when TLP and precomputation overlap for as much as 50% of the time, and synchronization costs 10% of the mean run length of one loop iteration. If the synchronization overhead is equal to half the mean execution time of one iteration (bottom chart in Figure 3), a chunk size equal to 20% of the trip count can still outperform thread-level parallel execution when precomputation and TLP overlap up to 30% of the time.

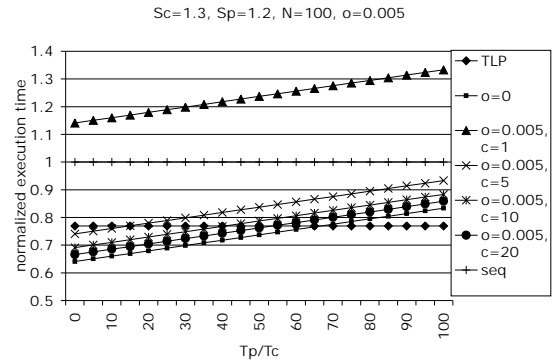
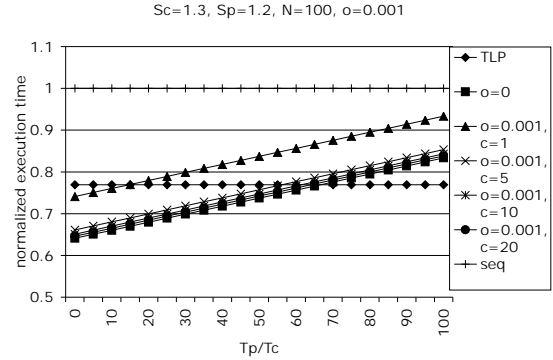


Figure 3: Impact of chunk size selection, as a function of the percentage of time precomputation overlaps with sequential computation, for some representative synchronization overheads and loop chunk sizes. The rest of the computation is executed using TLP.

In our implementation, we obtain the information for estimating computation run lengths using profiling. We apply precomputation and parallelization together only if the target computation’s run length is sufficiently long. We use a lower bound of 20000 cycles, following a suggestion in [19]. When the precomputation span exceeds this threshold, we select chunk sizes equal to 10% of the loop trip counts in the span. Though not a product of a formal analysis, our empirical model yields good results in our working prototype.

4. EXPERIMENTAL RESULTS

We experimented on a 4-processor Dell 6650 server, with Intel Xeon MP processors built with Hyperthreading technology and running at 1.4 GHz. The Xeon MP processor has two execution contexts with private register files. The contexts share most of the processor resources, including caches, TLB, and execution units. The processor has an 8 KB, 4-way associative L1 data cache, a 12 KB, 8-way associative instruction trace cache, a 256-KB, 8-way associative

Benchmark	Type	Data Set
3-D Jacobi	kernel, 7-point stencil	$100 \times 100 \times 100$ using 10×10 tiles
Regular m-x-m	kernel	1024×1024 10×10 tiles
NAS BT	application, PDE solver	Class A
NAS SP	application, PDE solver	Class A

Table 3: Benchmarks used in our experiments.

unified L2 cache and a 512 KB, 8-way associative external L3 cache. The system we used has 1 GB of RAM and runs Linux RedHat 9.0, with the 2.4.25 kernel. All experiments were conducted on an idle system. We report results obtained from several scientific kernels and application codes, using our runtime system. In all cases we constructed the precomputation threads manually after profiling our kernels with Valgrind and a simulated cache which was replicated accurately from our host system. We used a precomputation span target data set size of 32 KB, i.e. 1/8th of the size of the L2 cache of the processor. We used four codes, summarized in Table 3.

We report the execution times and L2 caches misses (plotted over time) for Jacobi, BT and SP in Figures 4, 5 and 6 respectively. Figure 7 reports execution times and L2 data cache misses for entire executions of the mxm kernel. All codes were implemented in C, using our own threads library, which is customized for Intel’s Hyperthreading processors. The implementations of the NAS Benchmarks were based on C implementations originally developed by the Omni OpenMP compiler group [17]. The 3-D Jacobi kernel was tiled for better temporal locality. The two loops used in the kernel were tiled in their two outermost levels. The tile sizes were selected after exhaustive search among square and rectangular tile sizes that fit in the L1 cache. The mxm kernel is a pencil and paper implementation parallelized along the outermost dimension. Both in the 3-D Jacobi kernel and in mxm, precomputation was targeted to eliminate L2 cache misses and was applied in multiple chunks for each loop, since the memory footprints of the loops exceeded by far the L2 cache size. A runahead distance equivalent to four cache misses was used in the 3-D Jacobi kernel. Multiple precomputation chunks per loop were used in NAS BT and SP in all cases, except from two of the three equation solvers (along the x and y dimensions) in which a single precomputation chunk was selected to span all the enclosed loops. Results for the NAS benchmarks were collected using the class A problem sizes, which fit the memory of our system. All benchmarks were compiled with the Intel C/C++ compiler (version 7.1). The loop chunk size was set equal to 10% of the trip count of the targeted loops.

The legends in the charts are read as follows: ST represents execution with one thread per processor and hyperthreading deactivated (meaning that one thread uses all the resources in the processor); TLP represents thread-level parallel execution with two threads per processor and hyperthreading activated; SPR represents speculative precomputation on one of the two threads using the schemes described in

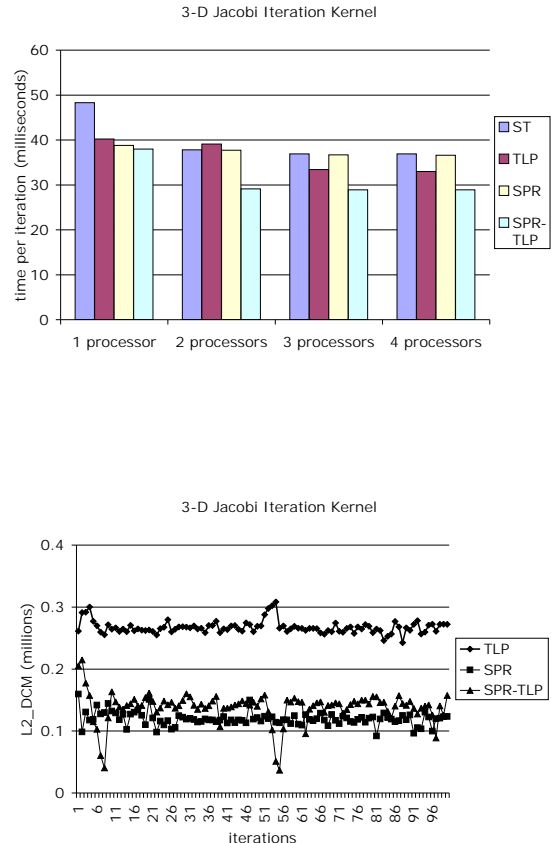


Figure 4: Execution times and L2 data cache misses per iteration of the tiled 3-D Jacobi iteration kernel, using plain TLP, plain SPR and the integrated scheme proposed in this paper.

Section 3.1; SPR-TLP represents the execution scheme that integrates precomputation and thread-level parallelism using the task queue model outlined in Section 3.2. Execution times and cache misses are reported per iteration for 3-D Jacobi, NAS BT and NAS SP, all of which are strictly iterative codes. Total execution time and total number of cache misses are reported for the mxm kernel. Cache misses were measured during executions on four processors (using four threads in ST versions and 8 threads in the other versions). The `perfctr` driver from Mikael Pettersson [14] was used for obtaining measurements from hardware performance monitoring counters. The driver was modified to collect reliably event counts from both threads on the same Hyperthreading processor.

The results indicate that neither precomputation alone nor TLP alone can always yield the best performance. In the 3-D Jacobi kernel, TLP is superior to SPR on three processors and four processors due to the drastic reduction of cache conflicts between threads. However, TLP is inferior to precomputation on one and two processors, where the

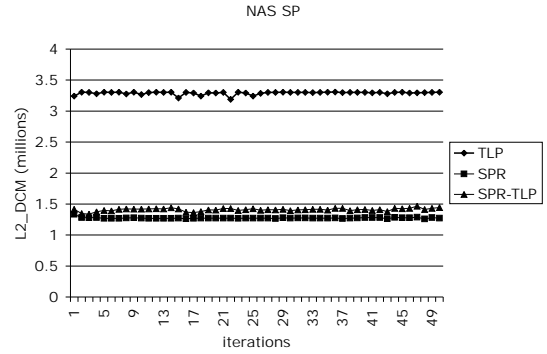
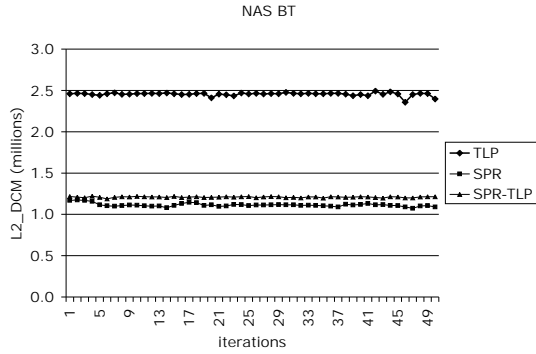
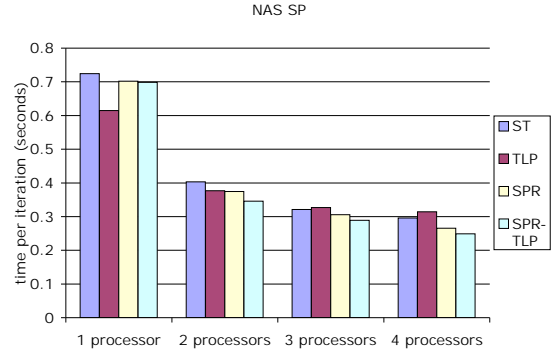
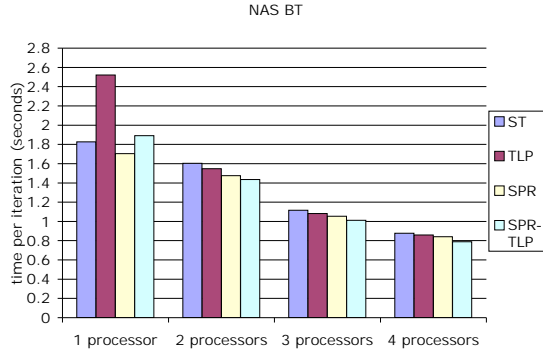


Figure 5: Execution times and L2 data cache misses per iteration of NAS BT, using plain TLP, plain SPR and the integrated scheme proposed in this paper.

Figure 6: Execution times and L2 data cache misses per iteration of NAS SP, using plain TLP, plain SPR and the integrated scheme proposed in this paper.

threads have larger working sets and conflicts dominate. In NAS BT, TLP is always inferior to SPR and the behavior of the parallelized version is pathological on one processor, due to cache thrashing. In NAS SP, TLP is superior to SPR on one processor, but inferior to SPR on two or more processors. In the mxm kernel, TLP is always a better option than SPR. Overall, precomputation can hide a significant fraction of memory latency in all cases, however its benefit is sometimes overwhelmed by the benefits of TLP.

The results show advantage in integrating precomputation and thread-level parallelism using our runtime mechanisms in three out of four codes, 3-D Jacobi, NAS BT and NAS SP. In the mxm kernel, the hybrid version and the version that uses only TLP perform almost the same.

In the parallel executions of the 3-D Jacobi kernel, TLP alone gives speedups ranging from 0.97 (a slowdown) to 1.16, over the implementation that uses a single thread per processor. Precomputation gives little speedup (1.01) on two, three and four processors and a noticeable speedup (1.19) on one processor. The combination of precomputation and

TLP achieves the best performance (speedup ranging between 1.21 and 1.23 on one to four processors, measured against the ST version). Plain TLP reduces the number of L2 data cache misses of the ST version by 43% per processor. Plain SPR reduces the L2 data cache misses of the ST version by 74% per processor, whereas the integrated SPR-TLP code reduces the L2 data cache misses of the ST code by 70% per processor. We observe that although the hybrid version pays an additional cache performance penalty due to conflicts between threads, it still achieves the best overall performance.

In the mxm kernel, thread-level parallelism appears to yield the best speedup over multiprocessor execution with a single thread per processor. This speedup ranges between 1.35 and 1.41 (the arithmetic mean is 1.37). The hybrid SPR-TLP version achieves speedups ranging between 1.33 and 1.40 (arithmetic mean is also 1.37). Plain precomputation yields only marginal speedups ranging from 0.97 (a slowdown on one processor) to 1.05. The explanation for the good performance of the TLP version is that the mxm kernel poses minimal conflicts in the functional units of the

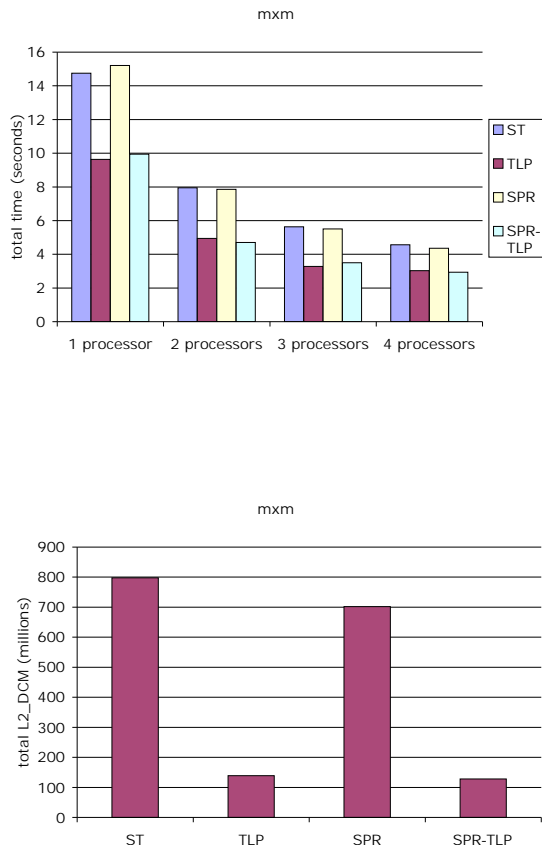


Figure 7: Execution times and L2 data cache misses of a pencil and paper implementation of mxm, using plain TLP, plain SPR and the integrated scheme proposed in this paper.

processor. On the other hand, precomputation has limited coverage, being able to mask only 18% of the cache misses incurred in the ST execution. Parallelization at the thread level does not incur any additional cache misses compared to the ST version (i.e. each thread incurs about half the L2 cache misses of the ST version). The hybrid SPR-TLP version performs competitively compared to the TLP version. On one and three processors, the TLP version outperforms the SPR-TLP version by 3% and 7% respectively. On two and four processors, the SPR-TLP version outperforms the TLP version by 5% and 3% respectively. The integrated SPR-TLP scheme eliminates about as many cache misses as the SPR version.

In NAS BT, thread-level parallelism within a single processor causes thrashing of the shared caches, yielding a performance penalty of almost 40%. The two versions of the code that use precomputation avoid this problem. On two, three and four processors, precomputation outperforms TLP by margins ranging between 2% and 5%. Precomputation combined with TLP yields the best performance, outperforming

plain TLP by 6.4% to 8%. The NAS SP benchmark does not suffer from the cache thrashing problem encountered in BT and the TLP implementation performs the best on one processor. However, both the SPR version and the integrated SPR-TLP version perform better on two to four processors. The hybrid version obtains the best speedup over the version that uses one thread per processor in both NAS BT and NAS SP (1.11 and 1.16 respectively).

Although in absolute terms the speedups we obtain appear to be modest, we must note that what limits the performance improvements to a large extent is the complex implications of sharing resources between threads on each processor. We have verified experimentally that even in the ideal, non-realistic case in which the two threads perform register-only operations without memory accesses, the speedup can not exceed 1.5 because of conflicts in instruction execution resources. On average, combinations of precomputation and TLP outperform plain TLP by 11%, plain precomputation by 8% and single-threaded execution per processor by 14%. The results show also that the precomputation mechanisms we present reach their primary target, by reducing L2 data cache misses by factors of two to three in three codes (3-D Jacobi, BT and SP) and by lower margins in one code (mxm). The aggressive miss overlap characteristics of the processor, as well as the relatively modest cache miss rates of the applications do not enable precomputation to yield even higher speedups. One interesting result that we observe from the cache miss rates is that the combination of precomputation and multithreading does not nullify the benefits of prefetching.

In order to investigate the possibility of achieving further improvements in codes with higher data cache miss rates, we conducted some preliminary experiments with an irregular data transposition kernel taken from the European Center for Medium-Range Weather Forecasts (ECMWF) IFS code. The IFS code is part of an integrated weather forecasting system which uses a spectral model of the atmosphere. We used a $102 \times 102 \times 102$ grid and measured approximately 12 million L2 cache misses per iteration, which is at least one order of magnitude more than the L2 cache misses in any of the other codes we used. We found that SPR was able to eliminate 40% of these cache misses, a rather modest number which could be increased if our precomputation regulation schemes could split precomputation in smaller chunks without significant overhead. Running the same code through the Valgrind cache simulator revealed a 75% proportion of conflict misses. Only 30% of these misses could be masked with our precomputation mechanisms, without creating excessive thread triggering overhead. Precomputation improved performance by 6%, whereas thread-level parallelism yielded a smaller improvement of 4.9%. This experiment revealed that a large number of L2 cache misses is not by itself a sufficient indication of the potential for speeding up code with precomputation, because precomputation has difficulties in resolving conflict misses, particularly when these misses happen in small time frames.

5. CONCLUSION

We presented runtime mechanisms that coordinate and combine speculative precomputation with thread-level parallelism to achieve more efficient execution of scientific

programs on real simultaneous multithreaded processors. The primary contributions of this paper are: a) methods to improve the effectiveness of speculative precomputation by regulating and coordinating precomputation with sibling computation; b) methods to combine precomputation and thread-level parallelism in the same program, to obtain the best of both techniques. We developed these mechanisms for scientific applications and we are currently engineering them so that they can be used in an experimental OpenMP compiler.

Our results prove the viability of the concept of hybrid multithreaded execution and we plan to explore this technique further in both shared-memory and distributed-memory codes, by factoring communication into the thread execution mechanism. A number of problems we encountered due to limitations of the Intel processors merit further investigation with simulation. These problems include the high synchronization and thread activation overhead, as well as unexpectedly high numbers of conflict misses that some codes had despite the high cache associativity. We anticipate that in future multithreaded processor cores, there will not be a 0-1 decision between precomputation and thread-level parallel execution, but precomputation will be used as an additional optimization tool which can be combined with other forms of multithreaded execution.

Acknowledgments

This work is supported by an NSF ITR grant (ACI-0312980), an NSF CAREER award (CCF-0346867) and the College of William and Mary.

6. REFERENCES

- [1] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *Proc. of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA'2002)*, pages 99–108, Winnipeg, Canada, August 2002.
- [2] J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic Speculative Precomputation. In *Proc. of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*, pages 306–317, Austin, TX, December 2001.
- [3] R. Cross. Intel Hyper-Threading Technology. *Intel Technology Journal*, 6(1), February 2002.
- [4] IBM. POWER5: IBM's Next Generation Power Microprocessor. In *Proc. of the 15th Symposium of High-Performance Chips (HotChips'15)*, Stanford, CA, August 2003.
- [5] D. Kim, S. Liao, P. Wang, J. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. Shen. Physical Experimentation with Prefetching Helper Threads on Intel's Hyper-Threaded Processors. In *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'2004)*, San Jose, CA, March 2004.
- [6] D. Kim and D. Yeung. A Study of Source-Level Compiler Algorithms for Automatic Construction of Pre-Execution Code. *ACM Transactions on Computer Systems*, 2004. to appear.
- [7] S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *Proc. of the 2002 ACM Conference on Programming Languages Design and Implementation (PLDI'02)*, pages 117–128, Berlin, Germany, June 2002.
- [8] B. Lim and R. Bianchini. Limits on the Performance Benefits of Multithreading and Prefetching. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):37–46, May 1996.
- [9] C. Luk. Tolerating Memory Latency through Software Controlled Preexecution on Simultaneous Multithreading Processors. In *Proc. of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*, pages 40–51, Göteborg, Sweden, July 2001.
- [10] T. Mowry. *Tolerating Latency through Software Controlled Data Prefetching*. PhD thesis, Department of Computer Science, March 1994.
- [11] T. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *ACM Transactions on Computer Systems*, 16(1):55–92, February 1998.
- [12] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Proc. of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, July 2003.
- [13] K. Olukotun, L. Hammond, and M. Willey. Improving the Performance of Speculatively Parallel Applications on the Hydra CMP. In *Proc. of the 13th ACM International Conference on Supercomputing (ICS'99)*, pages 21–30, Rhodes, Greece, June 1999.
- [14] M. Pettersson. Perfctr: Linux Performance Monitoring Counters Driver. Technical report, Computing Science Department, Uppsala University, January 2005. <http://user.it.uu.se/~mikpe/linux/perfctr>.
- [15] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *Proc. of the 7th International Symposium on High-Performance Computer Architecture (HPCA-7)*, pages 37–49, Moterrey, Mexico, January 2001.
- [16] A. Roth and G. Sohi. A Quantitative Framework for Automated Pre-Execution Thread Selection. In *Proc. of the 35th International Symposium on Microarchitecture (MICRO-35)*, Istanbul, Turkey, November 2002.
- [17] M. Sato, H. Harada, and A. Hasegawa. Cluster-Enabled OpenMP: An OpenMP Compiler for the SCASH Software Distributed Shared Memory System. *Scientific Programming*, 9(2-3):120–131, 2001.
- [18] X. Tian, Y. Chen, M. Girkar, S. Ge, R. Lienhart, and S. Shah. Exploring the Use of Hyper-Threading Technology for Multimedia Applications with Intel OpenMP Compiler. In *Proc. of the 17th International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.
- [19] N. Tuck and D. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proc. of the 2003 International Conference on Parallel Architectures and Compilation Techniques (PACT'2003)*, New Orleans, LA, September 2003.
- [20] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95)*, pages 392–403, St. Margherita Ligure, Italy, June 1995.
- [21] H. Wang, P. Wang, R. Weldon, S. Ettinger, H. Saito, M. Girkar, S. Liao, and J. Shen. Speculative Precomputation: Exploring the Use of Multithreading for Latency. *Intel Technology Journal*, 6(1), February 2002.