Exploring the Performance Limits of Simultaneous Multithreading for Scientific Codes *

Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
e-mail: {valia, anastop, kkourt, nkoziris}@cslab.ece.ntua.gr

Abstract

Simultaneous multithreading (SMT) has been proposed to improve system throughput by overlapping instructions from multiple threads on a single wide-issue processor. The speedup of a single application that is parallelized into multiple threads, is often sensitive to its inherent instruction level parallelism (ILP), as well as the efficiency of synchronization and communication mechanisms between its separate, but possibly dependent, threads.

In this paper, we evaluate and contrast software prefetching and thread-level parallelism (TLP) techniques for a series of scientific codes executed on an SMT processor. We explore the performance limits by evaluating the tradeoffs between ILP and TLP for various kinds of instructions streams. Obtaining knowledge on how such streams interact when executed simultaneously on the processor, and quantifying their presence within each application's threads, we try to interpret the observed performance for each application when parallelized according to the aforementioned techniques. In order to amplify this evaluation process, we also present results gathered from the performance monitoring hardware of the processor.

1 Introduction

Simultaneous Multithreading (SMT) is a hardware technique that allows a superscalar processor to issue instructions from multiple independent threads to its functional units, in the same cycle. Through this increased concurrency, SMT decreases wasted issue slots and increases flexibility [16].

Along with multithreading, prefetching is one of the most popular techniques for tolerating the ever-increasing memory wall problem. In contrast to multithreading, where instructions from different threads are simultaneously executed, prefetching tolerates latency by anticipating what data is needed and moving it to the cache ahead of time, when the running thread encounters a cache miss.

Regarding SMT, two main techniques were proposed in literature to utilize the multiple hardware contexts of the processors for improving performance of a single program: thread-level parallelism (TLP) and speculative precomputation (SPR). With TLP, sequential codes are parallelized so that the total amount of work is decomposed into independent parts which are assigned to a number of software threads for execution. In SPR, the execution of programs is facilitated with the introduction of additional threads, which speculatively prefetch data that are going to be used by the sibling computation threads in the near future, thus hiding memory latencies and reducing cache misses [18], [5], [14].

In this paper we demonstrate that significant performance improvements are hard to be achieved for already optimized parallel applications when running on processors equipped with hyper-threading(HT) technology([9]), Intel's implementation of simultaneous multithreading. Reference applications are loop-based scientific codes, both with regular and irregular or random memory access patterns. We tested the following configurations: First, we balanced the computational workload of a given benchmark on two threads, statically partitioning the iteration space. Then, we ran a main computation thread in parallel with a helper-prefetching thread. The latter was spawned to speculatively precompute addresses that trigger L2 cache misses and fetch the corresponding data. Synchronization of the two threads is essential, in order to avoid the helper thread from running too far ahead, evicting useful data from the cache. Furthermore, where applicable, we experimented with combinations of the above two configurations. We evaluated performance in two ways: First, we gathered re-

*This research is supported by the Pythagoras II Project (EPEAEK II), co-funded by the European Social Fund (75%) and National Resources (25%).

sults from specific performance metrics counted by the processor's monitoring registers. Second, we analyzed the dynamic instruction mix of each application's threads, and recognized the most dominant instructions. Having investigated the way that synthetic streams composed of these instructions interact on SMT processors, for different levels of TLP and ILP, we were able to give further explanations on the observed performance.

The main contributions of this paper are threefold: First, we investigate the experimental CPI and interaction for a number of synthetic instruction streams, common in scientific codes, when executed on an actual SMT processor. Second, we exhaustively attempt to achieve the best possible speedup on this processor, by applying two known techniques proposed in literature for multithreaded execution, i.e., TLP and SPR. In contrast to simulations so far presented, with real measurements on actual hardware, significant performance improvements are hard to be achieved for already optimized parallel applications. Finally, a careful analysis of both real and simulation measurements, identifies resource conflicts which constitute bottlenecks in achieving high performance.

The rest of the paper is organized as follows. Section 2 describes related prior work. Section 3 deals with implementation aspects of software techniques to exploit hardware multithreading. Section 4 explores the performance limits and TLP-ILP tradeoffs, by considering a representative set of instruction streams. Section 5 describes the experimental framework, presents performance measurements obtained from each application, and discusses their evaluation. Finally, we conclude with section 6.

2 Related Work

SMT [16], [17] is said to outperform previous execution models because it combines the multiple-instruction-issue features of modern superscalar architectures with the latency-hiding ability of multithreaded ones. However, the flexibility of SMT comes at a cost. When multiple threads are active, the static partitioning of resources (e.g., instruction queue, reorder buffer, store queue) affects codes with relative high instruction throughput. Static partitioning, in the case of identical thread-level instruction streams, limits its performance, but mitigates significant slowdowns when non-similar streams of microinstructions are executed [15].

Cache prefetching [8], [10] is a technique that reduces the observed latency of memory accesses by bringing data into cache before it is accessed by the CPU. Numerous thread-based prefetching schemes, either static or dynamic, have recently been proposed, including Roth and Sohi's Data Driven Multithreading [13], Luk's Software Controlled Pre-Execution [6], Collins et al., Speculative Precomputation [3], and Kim et al., Helper-Threads [5]. The key idea is to utilize otherwise idle hardware thread contexts to execute speculative threads on behalf of the main thread. These speculative threads attempt to trigger future cache-miss events far enough in advance of access by the non-speculative (main) thread, so that the memory miss latency can be masked. A common implementation pattern 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. Such load instructions, known as delinquent loads, may also be identified dynamically in hardware triggering speculative-helper threads [18]. SPR targets load instructions that exhibit irregular, data-dependent or pointer chasing access patterns. Traditionally, these loads have been difficult to handle via either hardware or software prefetchers.

3 Implementation

From a resource utilization perspective, threads performing software prefetching in SPR usually require less resources than the sibling computation threads, since they do not perform any meaningful computations that could affect program state or data. Furthermore, SPR can improve the performance of program codes that are not easily parallelizable. However, it targets only at reducing memory latencies and cannot always exploit the multiple units and superscalar execution capabilities of the SMT processor, especially in codes which exhibit low ILP. TLP, on the other hand, gives the opportunity to programs for a better utilization of the processor's resources. Since most of these resources, however, are shared between the threads, contention issues often arise, introducing performance penalties.

From an implementation point of view, sequential codes usually can be transformed into thread-level parallel ones in rather a straightforward manner, provided that they lend themselves for efficient parallelization. SPR mechanisms, on the other side, cannot always be incorporated that clearly. Since precomputation via multithreading must be as effective as any other software prefetching approach, applications must be subjected under fine tuning in order to deal with many synchronization and resource utilization matters that emerge. The co-existence of precomputation threads must introduce minimal interference and at the same time contribute beneficially to the progress of computation.

3.1 Synchronization Issues

We have implemented lightweight spin-wait loops as the core of our synchronization primitives. They are written using x86 assembly instructions, and operate entirely at user space on shared synchronization variables. When such loops are executed on processors supporting HT technology, they can induce additional performance penalty due to

memory order violations and consequent pipeline flushes caused upon their exit. Furthermore, they consume significant resources since they spin much faster than the time needed by the memory bus to perform a single update of the synchronization variable. These resources could be otherwise used to make progress on the other logical processor. In order to overcome these issues, we have embedded the `pause` instruction in the spin loop, as recommended by Intel [4]. This instruction introduces a slight delay in the loop and de-pipelines its execution, preventing it from aggressively consuming valuable processor resources. These are resources that are shared dynamically between the two threads on a hyper-threaded processor; execution units, branch predictors and caches are some examples.

However, some other units, such as micro-ops queues, load/store queues and re-order buffers were designed to be statically partitioned, such that each logical processor can use at most half of their entries. When a thread executes a `pause`, it does not releases the entries reserved for it. It continues to occupy them, while they could be entirely allocated to the sibling thread to help it execute at a greater efficiency. By using the privileged `halt` instruction, a logical processor can relinquish all of its statically partitioned resources, make them fully available to the other logical processor, and stop its execution going into a sleeping state. When it later receives an inter-processor interrupt (IPI) from the active processor, it resumes its execution and the resources are partitioned again.

The `halt` instruction is primarily intended for use by the operating system scheduler. Multithreaded applications with threads intended to remain idle for a long period, could take advantage of this instruction to boost their execution. This was the case for some of the multithreaded codes we developed throughout our study. We implemented kernel extensions that allow from user space the execution of `halt` on a particular logical processor, and the wake-up of this processor by sending IPIs to it. By integrating these extensions in the spin-wait loops, we are able to construct long duration wait loops that do not consume significant processor resources. Excessive use of these primitives, however, in conjunction with the resultant multiple transitions into and out of the halt state of the processor, incur extra overhead in terms of processor cycles. This is a performance tradeoff that we took into consideration throughout our experiments. Using the spin-wait loops, we implemented also barrier synchronization mechanisms with sense-reversing, as described in [12].

3.2 Implementing speculative precomputation

There are two main issues that must be taken into account in order to effectively perform software prefetching using the multiple execution contexts of a hyper-threaded processor. First of all, the distance at which the precomputation thread runs ahead of the main computation thread, has to be sufficiently large so that the data is prefetched into the caches before the computation thread makes use of it. At the same time it must be kept small enough, so that cache lines prefetched do not evict useful data in the cache that have not yet been consumed by the computation thread. Secondly, we must guarantee that the co-execution of the precomputation thread does not result in excessive consumption of shared resources that could be critical for the sibling computation thread.

The first requirement can be satisfied by imposing a specific upper bound on the amount of data to be prefetched. Whenever this upper bound is reached but the computation thread has not yet started using the prefetched data, the precomputation thread must stop its forward progress in order to prevent potential evictions of useful data from cache. It can only continue when it is signaled that the computation thread starts consuming the prefetched data. In our program codes, this scenario is implemented using synchronization barriers which enclose program regions (precomputation spans) whose memory footprint is equal to the upper bound we have imposed. The barriers for the precomputation thread are placed at the exit points of the spans, and at the entry points of the spans for the sibling computation thread. In this way, precomputation thread always runs ahead of the sibling thread, maintaining a regulated distance. In the general case, and considering their relatively lightweight workload, precomputation threads reach always first the barriers. As a result, computation threads spend negligible portion of the execution time waiting on these barriers. In general, the upper bound we enforced in our codes ranges from $\frac{1}{A}$ to $\frac{1}{2}$ of the L2 cache size, where A is the associativity of the cache (8 in our case). The fraction $\frac{1}{A}$ is proposed in [14] as a means to eliminate potential conflict misses.

In order to identify precomputation spans with memory footprints of a specific size, we followed two different approaches. For programs with regular or predictable memory access patterns, spans were identified by simple inspection. For codes whose access patterns were difficult to determine a-priori, we had to conduct memory profiling using the Valgrind simulator[11]. From the profiling results we were able to determine and isolate the instructions that caused the majority(92% to 96%) of L2 misses. In either case, precomputation threads were constructed manually from the original code of the main computation threads, preserving only the memory loads that triggered the majority of L2 misses; all other instructions were eliminated.

Despite the lightweight nature of the precomputation threads, as we mentioned in section 3.1 significant processor resources can be consumed even when they are simply

spinning on synchronization barriers. In order to avoid such performance degradation issues, we constructed a version of synchronization barriers with spin-loops that make use of the extensions we developed for halting and waking up the logical processors. When a precomputation thread enters the barrier, it puts its logical processor into halted state and goes itself into sleeping mode, offering all of its resources for exclusive use by the sibling thread. Similarly, when the computation thread is going to enter the barrier, it signals the sleeping thread to wake up. As we have mentioned, such transitions are expensive in terms of processor cycles. For this reason, in our multithreaded programs we followed a selective approach: we measured the times that precomputation threads spend on every barrier in the program; we then identified barriers in which these threads spin for a considerable portion of their total execution time; finally, we embedded our mechanisms for processor halting only in such "long duration" barriers - in the corresponding barriers of the computation threads we incorporated the counterpart mechanisms for processor wake-up.

4 Quantitative analysis on the TLP and ILP limits of the processor

In order to gain some notion about the ability and the limits of hyper-threading technology on interleaving and executing efficiently instructions from two independent threads, we have constructed a series of homogeneous instruction streams. These streams include basic arithmetic operations (add,sub,mul,div), as well as memory operations (load, store), on integer and floating-point 32-bit scalars. For each of them, we tested different levels of instruction level parallelism. Each stream is constructed by repeatedly inlining in our program the corresponding assembly instruction. All arithmetic operations are register-to-register instructions. The memory operations involve data transfers from memory locations to the processor registers and vice versa. In this case, each thread operates on a private vector, whose elements are traversed sequentially.

Let S and T be the sets of architectural registers that can be used within a window of consecutive instructions of a particular stream as source and target operands, respectively. In our experiments, we artificially increase(decrease) the ILP of the stream by keeping S and T always disjoint, and at the same time expanding(shrinking) T, so that the potential for all kinds of data hazards (i.e. WAW,RAW,WAR) is delimited(grown). These hazards are responsible for pipeline stalls. In our tests, we have considered three degrees of ILP: minimum ($|T|$=1), medium ($|T|$=3), maximum ($|T|$=6). To give an example of how we tune ILP in a stream of an instruction A, in the case of medium ILP, we repeat A so that exactly three registers are used exclusively as target registers, and furthermore, a specific target register is reused every three As.

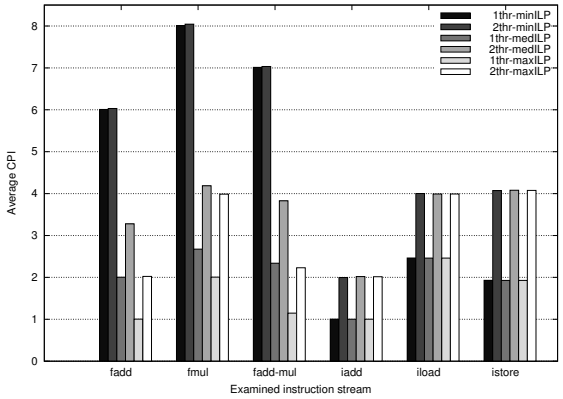


Figure 1. Average CPI for different TLP and ILP execution modes of some common instruction streams

As a first step, we execute each instruction stream alone on a single logical processor, for all degrees of ILP. In this way, all the execution resources of the physical package are fully available to the thread executing that stream, since the peer logical processor sits idle. We execute each stream for about 10 seconds, and for this interval we record the number of instructions that were executed and the total number of clock cycles that elapsed. By dividing these two quantities, we obtain an approximation for the CPI of a specific instruction in the context of a particular ILP level. As a second step, we co-execute within the same physical processor two independent instruction streams of the same ILP, each of which gets bound to a specific logical processor. We experiment with all possible combinations of the available instructions streams. For each combination, we perform as before a similar measurement for the CPI, and we compute then the factor by which the execution of a specific instruction was slowed down compared to its standalone execution. This factor gives us an indication on how various kinds of simultaneously executing streams of a specific ILP level, contend with each other for shared resources.

There is some additional information that we extract from the above experiments. For a particular instruction stream, we can estimate whether the transition from single-threaded mode of a specific ILP level to dual-threaded mode of a lower ILP level, can hinder or boost performance. For example, let's consider a scenario where, in single-threaded and maximum ILP mode, instruction A gives an average CPI of $C_{1thr-maxILP}$, while in dual-threaded and medium ILP mode the same instruction gives an average CPI of $C_{2thr-medILP} > 2 \times C_{1thr-maxILP}$. Because the second case involves half of the ILP of the first case, the above scenario prompts that we must probably not anticipate any

speedup by parallelizing into multiple threads a program that uses extensively this instruction in the context of high ILP (e.g. unrolling).

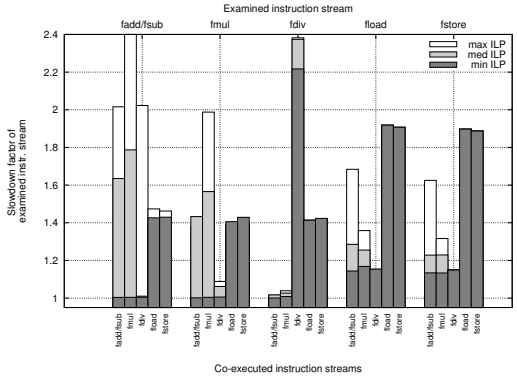
4.1 Co-executing streams of the same type

Figure 1 provides results regarding the average CPI for a number of synthetic streams. It demonstrates how the different combinations of TLP and ILP modes for a given stream can affect its execution time. The streams presented in the diagram are some of the most common instruction streams we encountered in real programs. Let's consider the $fadd$ instruction stream. In the case of minimum ILP, the cycles of the instruction do not alter when moving from 1 to 2 threads, which results practically in overall speedup. This reveals that the benefit from the strength of the processor to interleave instructions from the two threads, overlaps the cost of pipeline stalls due to the frequent data hazards from both threads. However, this scenario does not yield the best performance. The best instruction throughput is obtained in the single-threaded mode of maximum ILP, as depicted in the same diagram. The measurements show indirectly that an instruction window W_{fadd6} of 6 consecutive independent fp-add's executed by a single thread (*1thr-maxILP* case) can complete in less time than splitting the window in two and assigning each half to two different threads (*2thr-medILP* case). Furthermore, as implied by the results for the *2thr-maxILP* case, even if we distribute evenly a bunch of W_{fadd6} windows to two threads for execution, there is no performance gain compared to assigning all of them only to one thread (*1thr-maxILP* case, again). It seems that, when the available ILP increases in a program, pipeline stall problems diminish, leaving space for resource contention issues to arise and affect performance negatively.

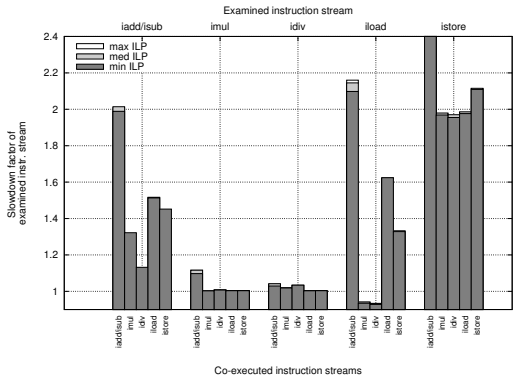
As Figure 1 shows, $fmul$ stream exhibits a similar variation in its CPI. It is interesting to see that mixing in a circular fashion in the same thread fp-add and fp-mul instructions, results in a stream ($fadd - mul$) whose final behavior is averaged over those of its constituent streams. For other instruction streams, such as $iadd$, it is not clear which mode of execution gives the best execution times, since the throughput remains the same in all cases. Hyper-threading achieved to favor TLP over ILP only for $iload$, because the cumulative throughput in all dual-threaded cases is larger compared to the single-threaded cases.

4.2 Co-executing streams of different types

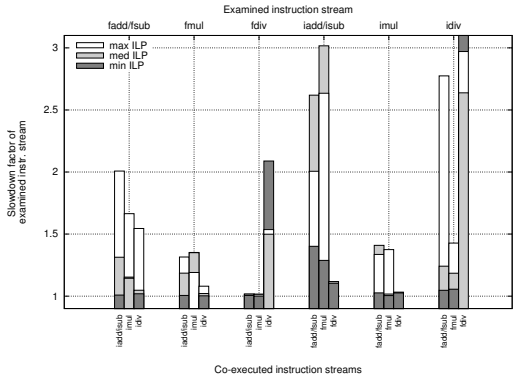
Figures 2(a) and 2(b) present the results from the co-execution of different pairs of streams (for the sake of completeness, results from the co-execution of a given stream with itself, are also presented). We examine pairs whose streams have the same ILP level, because we believe that this is the common case in most parallel applications. The slowdown factor represents the ratio of the CPI when two threads are running concurrently, to the CPI when the benchmark indicated on the top x-axis is being executed in single-threaded mode. What is clear at first glance, is that the throughput of integer streams is not affected by variations of ILP, as happens in the case of floating point streams.



(a) floating-point streams



(b) integer streams



(c) floating-point with integer streams

Figure 2. Slowdown factors from the co-execution of various instruction streams

The execution of $fdiv$ is mostly affected by streams of the same type (slowdown $120\% - 140\%$), but remains unaffected from variations of ILP. $fmul$ also experiences its major slowdown when co-executed with itself. $fadd/fsub$ streams, on the other hand, are affected by streams of the same type (slowdown up to 100%), as well as streams of different fp operations (e.g. slowdown of 180% with $fmul$). In lowest ILP mode, all different pairs of $fadd$, $fmul$ and $fdiv$ streams, can co-exist perfectly (except for the case of $fdiv$-$fdiv$). $fload$ or $fstore$ instructions (with a miss rate of 3%) can slowdown floating-point arithmetic operations by about 40%.

When both threads execute $iadd/isub$, a 100% slowdown arises, which is equivalent to serial execution. Other types of arithmetic or memory operations affect $iadd/isub$ less, by a factor of $10\% - 45\%$. $imul$ and $idiv$ instruction streams are almost unaffected by co-existing threads. $iadd/isub$ induce a slowdown of about 115% and 320% to $iload$ and $istore$ instruction streams, respectively (with 3% miss rate).

Finally, we mixed integer and floating-point instruction streams. Such mixes are more frequent in multiprogrammed workloads, rather than multithreaded scientific codes. The results are depicted in Figure 2(c), and regard pairs of floating-point and integer arithmetic streams of the same ILP. Due to space limitations, a more detailed discussion on these results is omitted.

5 Experimental Framework and Results

We experimented on Intel Xeon processor enabled with HT technology, running at 2.8GHz. This processor is based on Netburst microarchitecture, characterized by the deep pipeline and out-of-order execution capabilities. The core can retrieve three microoperations (μops) per cycle from the trace cache, execute up to six per cycle and graduate up to three per cycle. HT technology makes a single physical processor appear as two logical processors by applying a two-threaded SMT approach. The OS identifies two different logical processors, each maintaining a separate run queue.

It is worth noting that with the introduction of HT technology on Intel processors, the performance monitoring capabilities were extended, so that the performance counters could be programmed to select events that are qualified by logical processor IDs, whenever that was possible. To use these performance monitoring capabilities, a simple custom library was developed. For each of the multithreaded execution modes presented in section 3 we present measurements taken for three events:

• **L2 Misses**: The number of 2nd level read misses as seen by the bus unit. For the TLP methods, including the prefetch hybrid method the L2 misses presented are the sum of the misses for both threads. For the pure software prefetch method, only the misses of the working thread are presented.

• **Resource stall cycles**: The number of clock cycles that a thread stalls in the processor allocator, waiting until store buffer entries are available. This performance metric is indicative of the contention that exists between hardware threads. For all cases, the results presented correspond to the sum of stall cycles on behalf of both logical processors.

• **μops retired**: The number of μops that were retired during the execution of the program. For all cases the μops number is the number of those retired for both threads.

Both in TLP and SPR versions of our codes, we create two threads each of one we bind to a specific logical processor within a single physical package. We have used the NPTL library for the creation and manipulation of threads. Our operating system was Linux version 2.6.9. To force the threads to be scheduled on a particular processor, we have used the `sched_setaffinity` system call. All user codes were compiled with gcc 3.3.5 compiler using the O2 optimization level, and linked against glibc 2.3.2.

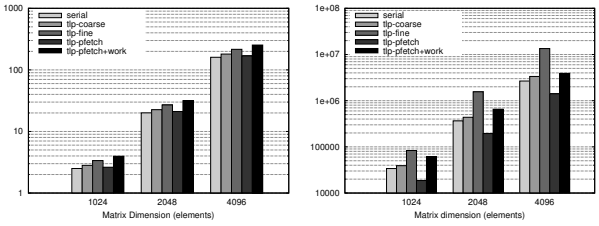
5.1 Microkernels

This section presents the experimental results using two popular computational kernels, Matrix Multiplication and LU decomposition. Experiments were conducted for three matrix sizes: 1024×1024, 2048×2048 and 4096×4096.

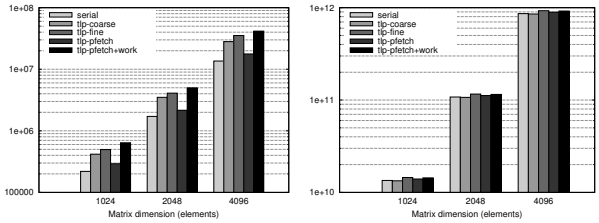
i) Matrix Multiplication: We developed a number of tiled multithreaded versions, where blocked array layouts were applied. We implemented two schemes: work partitioning (TLP) and SPR.

The work partitioning schemes divide the total amount of work into equal parts, each of which gets assigned statically to a specific thread. In our case, each thread takes over different parts of matrix C to compute. We further developed two versions of such schemes, a fine-grained and a coarse-grained one. In the fine-grained version (**tlp-fine**), consecutive elements within a single tile of C are assigned to different threads in a circular fashion. Due to blocked array layouts, these elements are stored in contiguous memory locations, so that traversed elements are mapped in nearby but not identical cache locations. We have chosen tile sizes that completely fit in L1 cache, as they had the best performance, in terms of execution time. In the coarse-grained version (**tlp-coarse**), consecutive tiles of C are assigned to different threads circularly. This way, the two threads work on different cache areas, without interfering in one's another cache lines.

The SPR techniques use prefetching to tolerate cache misses, following the scheme we described in section 3.2. In the pure prefetching version (**tlp-pfetch**), the whole workload is executed by just one thread, while the second

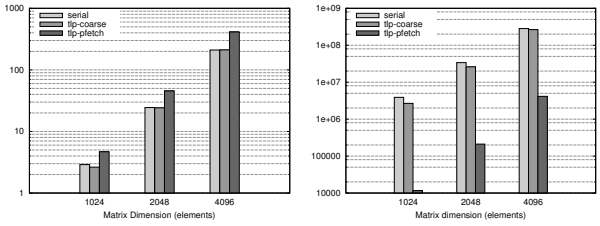


(a) Execution Time (sec) (b) L2 misses

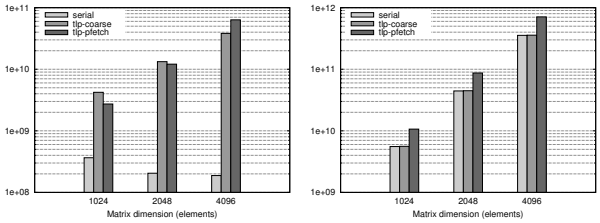


(c) Resource stall cycles (d) μops retired

Figure 3. Results for the Matrix Multiplication kernel



(a) Execution Time (sec) (b) L2 misses



(c) Resource stall cycles (d) μops retired

Figure 4. Results for the LU decomposition kernel

just prefetches the next tile in issue. In the hybrid prefetching version (**tlp-pfetch+work**), the workload is partitioned in a fine-grained fashion, while one of the two threads takes on the prefetching of the next tile in issue.

Figure 3 presents the above 4 versions of the matrix multiplication benchmark, as well as the serial version (**serial**), optimized with all possible loop transformation techniques, including loop unrolling. As illustrated in Figure 3(a), HT technology did not provide any speedup. The fastest dual-threaded method was the pure prefetch method, which had almost identical performance with the serial method, for all matrix sizes. Figure 3(b) shows that for the working thread, it can be achieved a great improvement in the number of L2 misses (around 82% less) when using a different thread as a prefetcher. However, this is not followed by overall speedup, due to the ineffective static resource partitioning in the processor (see section 5.3), which results in resource contention (Figure 3(c)). The other methods are also outperformed by the serial method. The *tlp-coarse* method is slower by a factor of 1.12, the *tlp-fine* by a factor of 1.34 and the *tlp-pfetch+work* method by a factor of 1.58 on all matrix sizes. In this case, also, execution time slowdown is consistent with the increase of stall cycles.

ii) LU decomposition: In the work partitioning implementation, the original LU kernel was parallelized by assigning different tiles to different threads for in-tile factorization (*tlp-coarse*). This is a coarse-grained scheme. It consists of three computation phases, which are determined by the inter-tile data dependences of the algorithm.

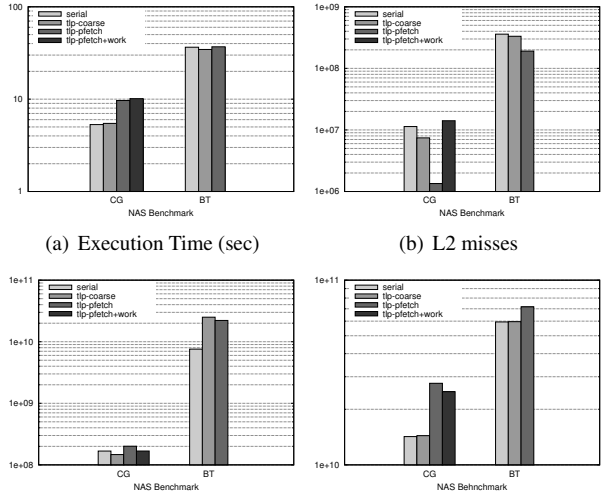
The SPR technique was implemented by means of a pure prefetching scheme (*tlp-pfetch*), where the prefetcher thread fills part of the L1 cache with the next tile to be factorized by the main worker thread. A hybrid precomputation scheme was not implemented for this kernel, since it would require a more fine-grained work partitioning strategy.

As Figure 4(a) depicts, the *tlp-coarse* method proved to be the fastest by offering a slight speedup in the range of $0.5\% - 8.9\%$. It is noteworthy that, despite the fact that the threads work on disjoint data, they contribute mutually to a reduction of the total L2 misses compared to the serial case. The relatively small tile size explains somehow this behavior, since the access of boundary tile elements by one thread seems to trigger more often the prefetch of cache lines which contain elements of neighbouring tiles. Despite the speedup gained in the *tlp-coarse* version, stall cycles grow up to one or even two orders of magnitude. Stalling one of the two threads in the allocator seems to leave enough room in the execution subunits for the other thread to perform better. As in the MM kernel, the L2 misses of the worker thread were also decreased significantly (around 98%) using a second prefetcher thread. However, given that the dual-threaded prefetch method needs more than double the amount of μops to complete compared to the serial method, as Figure 4(d) designates, it is expected that the program will not be able to benefit from the better memory locality.For this method, and for increasing matrix dimension, there was a slowdown by a factor from 1.61 to 1.96.

5.2 NAS benchmarks

In this section, the experimental results using CG and BT benchmarks from NPB suite version 2.3 are presented. CG

solves an unstructured sparse linear system by the conjugate gradient method. The benchmark is characterized by random memory access patterns. BT solves block-tridiagonal systems of 5×5 blocks using the finite differences method, and exhibits somewhat better data locality. The performance of both benchmarks was evaluated for Class A data sizes. The implementations of the benchmarks were based on the OpenMP C versions of NPB 2.3 provided by the Omni OpenMP Compiler Project [1]. We transformed these versions so that appropriate threading functions are used for work decomposition and synchronization, instead of OpenMP constructs.



(a) Execution Time (sec) (b) L2 misses

(c) Resource stall cycles (d) μops retired

Figure 5. Results for CG and BT NAS benchmarks

i) CG: In this case, also, the single-threaded version outperforms all other dual-threaded methods which use the HT technology of the Xeon processor(Figure 5(a)).

The *tlp-coarse* method is slower only by a factor of 1.03, when compared to the single-threaded approach. The methods which implement software prefetching are outperformed by a factor 1.82 for the pure method and 1.91 for the hybrid method. In Figure 5(b) we can see that both the *tlp-coarse* and *tlp-pfetch* methods have better locality than the serial method. Also in Figure 5(d) it is evident that the great performance decrease in the prefetch method is due to the increase in the total number of μops for the program. On the other hand, the increase of total μops for the *tlp-coarse* method is rather small and so is the effect in the total execution time. Figure 5(c) shows that there is no significant variation in the number of stall cycles, indicating that contention in the store buffers is not the reason for performance slowdown.

ii) BT: For this benchmark, we were able to achieve a performance gain by exploiting the HT technology (Figure 5(a)). There was a relative performance gain by a 1.06 factor using the *tlp-coarse* method and a relative performance loss of a 1.01 factor when using the *tlp-pfetch* method. Although, again, *tlp-pfetch* was able to significantly decrease the L2 misses of the worker thread (Figure 5(b)), there was not observed any performance gain because of the increased μops required to implement the prefetching. On the contrary, in the case of *tlp-coarse* there was a drop in L2 misses and the μops remained the same, which lead to better performance. Stall cycles (similarly to LU case) increase considerably, too.

5.3 Further Analysis

Table 1 presents the utilization of the busiest processor execution subunits, when running the reference applications. The first column (serial) contains results of the single-threaded versions. The second column (tlp) presents the behavior of one of two threads for the TLP implementation. In this case, each of the two threads execute an almost equivalent load (at about a half of the total instructions of the serial case, if we ignore the extra parallelization overhead), and consequently, results are identical. The third column (spr) presents statistics of the prefetching thread in the SPR versions of our codes. All percentages in the table refer to the portion of the total instructions of each thread that used a specific subunit of the processor. The statistics were generated by profiling the original application executables using the Pin binary instrumentation tool [7], and analyzing for each case the breakdown of the dynamic instruction mix, as recorded by the tool. Figure 6([4]) presents the main execution units of the processor, together with the issue ports that drive instructions into them. Our analysis examines the major bottlenecks that prevent multithreaded implementations from achieving some speedup.

Compared to the serial versions, TLP implementations do not generally change the mix for various instructions. Of course, this is not the case for SPR implementations. For the prefetcher thread, not only the dynamic mix, but also the total instruction count, cannot be compared with those of the worker thread. Additionally, different memory access patterns require incomparable effort for address calculations and data prefetching, and subsequently, different number of instructions.

In the MM benchmark the most specific characteristic is the large number of logical instructions used: at about 25% of total instructions in both the serial and the TLP versions. This is due to the implementation of Blocked Array Layouts with binary masks [2] that were employed for this benchmark. Although the out-of-order core of the Xeon processor possesses two ALU units (double speed), among them only ALU0 can handle logical operations. As a result,

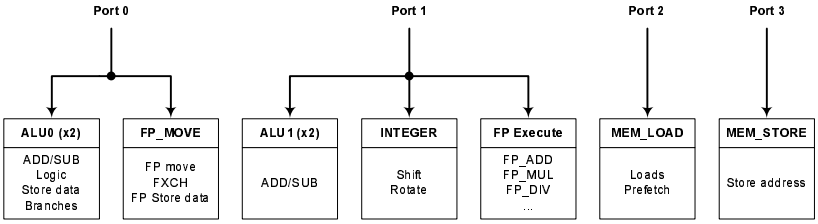


Figure 6. Instruction issue ports and main execution units of the Xeon processor

		Instrumented thread		
	EX. UNIT	***serial***	***tlp***	***spr***
MM	ALUs:	27.06%	26.26%	37.56%
	FP_ADD:	11.70%	11.82%	0.00%
	FP_MUL:	11.70%	11.82%	4.13%
	LOAD:	38.76%	27.00%	58.30%
	STORE:	12.07%	12.02%	20.75%
	Total instr.($\times 10^9$):	4.60	2.27	0.20
LU	ALUs:	38.84%	38.84%	38.16%
	FP_ADD:	11.15%	11.15%	0.00%
	FP_MUL:	11.15%	11.15%	0.00%
	LOAD:	49.24%	49.24%	38.40%
	STORE:	11.24%	11.24%	22.78%
	Total instr. ($\times 10^9$):	3.21	1.62	3.26
CG	ALUs:	28.04%	23.95%	49.93%
	FP_ADD:	8.83%	7.49%	0.00%
	FP_MUL:	8.86%	7.53%	0.00%
	FP_MOVE:	17.05%	14.05%	0.00%
	LOAD:	36.51%	45.71%	19.09%
	STORE:	9.50%	8.51%	9.54%
	Total instr.($\times 10^9$):	11.93	7.07	0.17
BT	ALUs:	8.06%	8.06%	12.06%
	FP_ADD:	17.67%	17.67%	0.00%
	FP_MUL:	22.04%	22.04%	0.00%
	FP_MOVE:	10.51%	10.51%	0.00%
	LOAD:	42.70%	42.70%	44.70%
	STORE:	16.01%	16.01%	42.94%
	Total instr. ($\times 10^9$):	44.97	22.49	8.40

Table 1. Processor subunits utilization from the viewpoint of a specific thread

concurrent requests for this unit in the TLP case, will lead to serialization of corresponding instructions, without offering any speedup. In the SPR version, ALU0 utilization on behalf of prefetcher thread is significant, as well. But in this case, the contention, and consequently the slowdown, is not as high as in TLP case, since the prefetcher executes only a small fraction of the worker's total instructions.

With respect to MM, LU exhibits higher ALUs usage. In this case, however, instructions can be executed by both ALUs and thus are distributed equally on them. Furthermore, as we can see from Figure 1, raw $fadd-mul$ streams can interact without incurring slowdowns. These observations, in conjunction with the fact that each thread in the TLP case executes almost half the instructions of the serial version, explain in a way the marginal speedup achieved. On the contrary, in the SPR case, the prefetcher executes at least the same number of instructions as the worker, and also puts the same pressure on ALUs. This is due to the non-optimal data locality, which leads prefetcher to execute a large number of instructions to compute the addresses of data to be brought in cache. These facts translate into major slowdowns for the SPR version of LU, despite any significant L2 misses reduction.

In CG benchmark, the utilization percentages of each thread in TLP version are not indicative of performance degradation. As Figure 5(a) demonstrates, though, there is a minor slowdown. This is because each thread executes more than the half of the instructions compared to the serial case, due to parallelization overhead. A noteworthy remark regards the SPR implementation of CG: although the prefetcher executes a small number of instructions, execution time is decelerated significantly. We believe that the frequent invocations of synchronization primitives in this benchmark (not included in the profiling process) are responsible for this behavior.

As can be seen in Figure 5(a), TLP mode of BT benchmark was one of few cases that gave us some speedup (around 6%). The relatively low usage and thus contention on ALUs, in conjunction with non-harmful co-existence of $fadd-mul$ streams (which dominated, again, other instructions) and the perfect workload partitioning, are among the main reasons for this speedup. Following a similar rationale, low ALUs utilization together with small number of total instructions, led to minimal performance drop (about 1%) when the SPR scheme was applied.

6 Conclusions

This paper presents performance results for a simultaneous multithreaded architecture, the hyper-threaded Intel microarchitecture. We examined single-programmed workloads, where both work partitioning schemes to exploit TLP, and SPR techniques, were applied. Our evaluation was based on actual program execution, as well as simulation. The results gathered demonstrated the limits in achieving

high performance for multi-threaded applications.

SPR can achieve a fairly good reduction in L2 cache misses. However, in order to fine tune data prefetching, a considerable number of additional instructions have to be inserted into the pipeline. This increase in the number of μops, in combination with some kind of resource contention, harms performance in terms of execution time. Besides, optimized applications with a relatively high IPC (such as the tested microkernels), are really difficult to achieve even better performance without reducing the μops executed. Thus, embodying SPR in the working thread, seems to be the solution that combines low number of μops with reduced cache misses and achieves best performance.

Coarse-grained work partitioning does not have a significant impact on the number of μops executed (usually brings a slight increase). Total execution performance would be expected to be improved, especially in cases of L2 cache miss decrease. However, the two working threads, due to their symmetric profiles, compete for the same hardware resources. This contention constitutes a bottleneck to high performance. A noteworthy performance speedup was achieved only for one of the NAS benchmarks (BT). In this case, irregular memory access patterns (which impose a significant latency), in combination with assorted compute instructions (which do not put pressure on just one type of hardware resources), hide memory latency by interleaving it with computation and minimize overhead due to resource contention.

References

[1] Omni OpenMP Compiler Project. Released in the International Conference for High Performance Computing, Networking and Storage (SC'03), Nov 2003.

[2] E. Athanasaki and N. Koziris. Fast Indexing for Blocked Array Layouts to Improve Multi-Level Cache Locality. In *Proc. of INTERACT'04*, Madrid, Spain.

[3] J. D. Collins, H. Wang, D. Tullsen, C. Hughes, Y-F. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proc. of ISCA '01*, Göteborg, Sweden.

[4] Intel Corporation. *IA-32 Intel Architecture Optimization*. Order Number: 248966-011.

[5] D. Kim, S-W. Liao, P. H. Wang, J. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. P. Shen. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *Proc. of IEEE/ACM CGO 2004*, San Jose, CA.

[6] C-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proc. of ISCA '01*, Göteborg, Sweden.

[7] C-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 40(6):190–200, 2005.

[8] C-K. Luk and T. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proc. of ASPLOS-VII*, Boston, MA.

[9] D. T. Marr, F. B. Desktop, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, Feb 2002.

[10] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[11] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Proc. of RV'03*, Boulder, CO.

[12] D. Patterson and J. Hennessy. *Computer Architecture. A Quantitative Approach*, chapter 6.7. pages 597–598. Morgan Kaufmann, 3rd edition, 2003.

[13] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. In *Proc. of HPCA '01*, Nuevo Leone, Mexico.

[14] F. Blagojevic T. Wang and D. S. Nikolopoulos. Runtime Support for Integrating Precomputation and Thread-Level Parallelism on Simultaneous Multithreaded Processors. In *Proc. of LCR'2004*, Houston, TX.

[15] N. Tuck and D. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proc. of PACT '03*, New Orleans, LA.

[16] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of ISCA '96*, Philadelphia, PA.

[17] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. of ISCA '95*, Santa Margherita Ligure, Italy.

[18] H. Wang, P. Wang, R. D. Weldon, S. M. Ettinger, H. Saito, M. Girkar, S. S-W. Liao, and J. P. Shen. Speculative Precomputation: Exploring the Use of Multithreading for Latency. *Intel Technology Journal*, 6(1):22–35, Feb 2002.