J Supercomput (2008) 44: 64–97
DOI 10.1007/s11227-007-0149-x

Exploring the performance limits of simultaneous multithreading for memory intensive applications

Evangelia Athanasaki · Nikos Anastopoulos · Kornilios Kourtis · Nectarios Koziris

Published online: 6 October 2007

Abstract Simultaneous multithreading (SMT) has been proposed to improve system throughput by overlapping instructions from multiple threads on a single wide-issue processor. Recent studies have demonstrated that diversity of simultaneously executed applications can bring up significant performance gains due to SMT. However, 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. Moreover, as these separate threads tend to put pressure on the same architectural resources, no significant speedup can be observed.

In this paper, we evaluate and contrast thread-level parallelism (TLP) and speculative precomputation (SPR) techniques for a series of memory intensive codes executed on a specific SMT processor implementation. We explore the performance limits by evaluating the tradeoffs between ILP and TLP for various kinds of instruction streams. By 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.

E. Athanasaki (✉) · N. Anastopoulos · K. Kourtis · N. Koziris
School of Electrical and Computer Engineering, Computing Systems Laboratory, National Technical University of Athens, Zografou Campus, Zografou 15773, Greece
e-mail: valia@cslab.ece.ntua.gr

N. Anastopoulos
e-mail: anastop@cslab.ece.ntua.gr

K. Kourtis
e-mail: kkourt@cslab.ece.ntua.gr

N. Koziris
e-mail: nkoziris@cslab.ece.ntua.gr

Keywords Simultaneous multithreading · Thread-level parallelism · Instruction-level parallelism · Software prefetching · Speculative precomputation · Performance analysis

1 Introduction

Simultaneous Multithreading (SMT) [30] allows a superscalar processor to issue instructions from multiple independent threads to its functional units, in a single cycle. It targets at maximum utilization of processor resources by simultaneously processing independent operations. The key motivation behind this technique is that many applications are characterized by insufficient inherent ILP that leaves the multiple units of a superscalar processor underutilized during each cycle, or by long latency operations, such as cache misses or branch mispredictions that stall the entire pipeline for multiple cycles.

Applications exhibiting either kind of behavior during their execution can benefit from SMT, when they are parallelized into multiple threads. In the former case, maximum utilization can be achieved when instructions from additional application threads (which may also exhibit low parallelism) fill up empty issue slots on the processor. In this way, thread level parallelism is effectively converted into instruction level parallelism. In the latter case, whenever a thread stalls on a long latency event, instructions from other threads can immediately issue, and therefore, utilize the otherwise idle pipeline.

Along with multithreading, prefetching is one of the most popular techniques for tolerating the ever-increasing memory wall problem. In contrast to multithreading, where memory stalls of a thread are overlapped by executing useful instructions from other threads, prefetching usually targets memory latency tolerance within a single thread of execution. Prefetch instructions move data to some level of the cache hierarchy ahead of time, anticipating that they will arrive early enough before they are actually needed. As long as prefetched data are not evicted prior to its use, memory access latency can be completely hidden.

Regarding SMT, two main alternatives have been investigated in literature to utilize the multiple hardware contexts of the processor in order to accelerate a sequential application: thread-level parallelism (TLP) and speculative precomputation (SPR). With TLP, sequential codes are parallelized so that the total amount of work is distributed evenly among threads for execution, as in traditional shared memory multiprocessors. TLP is a commonplace option for most regular codes. In SPR, the execution of an application is facilitated by additional helper threads, which speculatively prefetch data that are going to be used by the main computation thread in the near future, thus hiding memory latency and reducing cache misses [5]. It targets performance improvement of applications that are not easily parallelizable or exhibit hardly predictable access patterns that render prefetching ineffective.

In either scenario, executing multiple application threads on SMT architectures is a subtle issue. Most processor resources (e.g., caches, instruction queues, functional units, fetch/decode units, etc.) are either shared or partitioned between logical processors, and only a few of them are replicated. Therefore, significant resource conflicts

can arise, e.g., due to cross-thread cache line evictions or when threads tend to compete for the same units at the same time. Furthermore, the benefit of multithreading on SMT architectures is sensitive to the specific application and its level of tuning [19]. For example, a highly optimized (e.g., unrolled and blocked) version of matrix multiplication is usually tuned to fully exploit registers, cache and multiple functional units of the processor. As a result, adding extra threads on the same physical package may harm this good usage, and therefore, performance. On the other hand, a nonoptimized matrix multiply has poor locality and exploitation of resources, and hence may benefit from SMT.

Regarding single application performance, most previous work demonstrated results on simulated SMT machines [5, 14, 19, 31]. Experiments on real SMT-based machines did not report significant speedups when sequential stand alone applications were threaded according to either TLP or SPR [7, 11, 28, 31, 32]. In this work, we explore the potential for performance improvement of memory intensive applications when they execute on hyper-threaded processors using the aforementioned multithreading techniques. Hyper-threading technology [18] is Intel's implementation of SMT. Reference applications for our study are array-based numerical programs, both with regular and irregular or random memory access patterns, as well as array-based and pointer-intensive codes found in graph and database applications, mostly characterized by irregular accesses. Although a particular technique may seem more appropriate than any other for a specific class of applications, as discussed above (e.g., TLP for regular codes, SPR for more irregular ones), our work aims at a comprehensive evaluation of all options for all codes we consider.

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 along 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 a key factor for the effectiveness of SPR, since it must guarantee accuracy and timeliness of data prefetches by properly regulating the run ahead distance of the helper thread, and at the same time introduce minimal overhead.

We evaluated performance in two ways: First, we gathered results 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 the codes we consider when executed on an actual SMT processor. Second, we exhaustively attempt to achieve the best possible speedup on this processor, by applying the two alternatives for multithreaded execution, i.e., TLP and SPR. In contrast to simulations so far presented, with real measurements on actual hardware, we find that significant performance improvements are hard to be achieved for most of the applications. Finally, a careful analysis of both real and simulation measurements, reveals weaknesses of the specific SMT implementation under test and identifies resource conflicts that constitute performance bottlenecks.

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 presents details on the processor and the experimental setup. Section 6 demonstrates experimental results obtained for each application, and discusses their evaluation. Finally, we conclude with Sect. 7.

2 Related work

Simultaneous multithreading [30] 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: All hardware contexts are simultaneously active, competing in each cycle for the available shared resources. This dynamic sharing of the functional units allows SMT to substantially increase throughput, attacking the two major impediments to processor utilization—long latencies and limited per-thread ILP. 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 performance, but mitigates significant slowdowns when nonsimilar streams of microinstructions are executed [28].

A number of works have considered software-controlled prefetching techniques for improving single-thread application performance. These works focus on codes with regular access patterns [20, 21], as well as pointer-based codes with recursive data structures that exhibit irregular access patterns [16, 17]. Cache prefetching reduces the observed latency of memory accesses by bringing data into the cache before they are accessed by the CPU. However, as processor throughput improves due to various other memory latency tolerance techniques, prefetching suffers from certain disadvantages. First, use of memory bandwidth is increased since prefetching increases memory traffic. Memory requirements are also increased as much more cache accesses are generated and thus, more data are being brought in the caches for future use. Finally, the overhead due to the extra prefetch instructions inserted can be notable, outweighing any performance gains of prefetching.

Numerous thread-based prefetching schemes have been proposed in literature. The key idea is to utilize one or more spare execution contexts of a multithreaded or multicore processor to speculatively precompute future memory accesses and prefetch data for the nonspeculative (main) context, under the assumption that all contexts share some level of cache. By having separate threads to execute the entire path down to the computation of specific addresses, we are able to handle more efficiently access patterns that are otherwise difficult to prefetch, either by hardware or by software (i.e., by simply inserting prefetch instructions in the main context). Studies of particular importance dealing with thread-based prefetching are Collins et al. Speculative Precomputation [5], Roth's and Sohi's Speculative Data Driven Multithreading [24],

Sundaramoorthy et al. Slipstream Processors [26], Luk's Software Controlled Pre-Execution [14] and Kim et al. Helper-Threads [11]. Precomputation schemes considered in these studies target mainly loads with unpredictable, irregular, data-dependent or pointer-chasing patterns. Usually, a small number of static loads, known as *delinquent loads*, are responsible for the majority of memory stalls in the nonspeculative thread of each application.

As for thread-level parallel codes, most studies with simulated SMTs have demonstrated limited performance gains mainly because of homogeneity of threads, as opposed to multiprogrammed workloads where resource requirements are less conflicting [12, 13, 30]. This trend is persistent in evaluations with real HT-based machines as well [4, 7, 28, 32]. Most of these works consider OpenMP as the main programming paradigm for thread-level parallelization. In our work, we use explicit threading, as an attempt to minimize thread management overheads of runtime mechanisms and isolate each application's inherent ability to scale on SMTs.

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 take advantage of 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. There are cases, however, where application threads can benefit from shared resources, and especially shared caches. This happens, for example, when threads work on adjacent data, so that cache lines fetched by one thread can be used by others, as well.

From an implementation point of view, sequential codes usually can be transformed into thread-level parallel ones in a rather straightforward manner, provided that they have sufficient inherent parallelism. Parallelization is usually performed by statically partitioning the iteration space at innermost (*fine-grained*) or outermost (*coarse-grained*) loop levels, as happens in traditional SMP environments. Due to its relative simplicity, there are no generic TLP implementation issues we could discuss in this section. We will rather go through a brief discussion about such issues on a per-application basis. On the other hand, SPR mechanisms 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 issues that emerge. The coexistence and coordination of precomputation threads must introduce minimal interference and at the same time contribute beneficially to the progress of main computation threads. In the sections that follow, we examine many key aspects in the process of implementing SPR.

3.1 Synchronization issues

Many of our multithreaded workloads make extended use of synchronization mechanisms to guarantee atomicity on shared data accesses and to control and coordinate thread execution. These mechanisms have to be as lightweight as possible, so that their frequent invocation does not introduce significant overhead. For this purpose, we have implemented spin-wait loops as a building block of our synchronization primitives. Spin-wait loops are written using 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 [10]. 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; as Table 1 presents, the execution units, branch predictors and caches are some examples.

Not all resources, however, are released by a thread when it executes a `pause` instruction. Some processor units, such as the 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 continues to occupy the entries reserved for it in these units. The thread, however, does not contribute to any useful work, so its entries could be entirely allocated to the sibling thread to help it execute faster. 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. Later, as soon as it 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, as described in Sect. 5.2. 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. The Linux OS we used does not provide services to be used

Table 1 Hardware management in Intel hyper-threaded processors

Shared	Execution units, trace cache, L1 D-cache, L2 cache, DTLB, global history array, microcode ROM, uop retirement logic, IA-32 instruction decode, instruction scheduler, instruction fetch logic
Replicated	Processor architecture state, instruction pointers, rename Logic, ITLB, streaming buffers, return stack buffer, branch history buffer
Partitioned	uop queue, memory instruction queue, reorder buffer, general instruction queue

explicitly by applications for similar purposes. For this reason, we implemented kernel extensions that allow the execution of `halt` from user space 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.

3.2 Implementing speculative precomputation

Most previous work on SPR has relied on schemes that continuously spawn new speculative threads during execution, either at the context of the main thread or at the context of other speculative threads, at designated points in the program. These schemes assume ideal hardware support, such as multiple contexts to host multiple speculative threads, efficient hardware mechanisms that permit a thread to spawn another instantly and with minimal overhead (e.g., flash-copy of live-in variables), or special hardware for lightweight synchronization and low overhead suspension/resumption of threads.

None of these facilities, however, are supported in current hyper-threaded processors. We have to employ therefore software techniques for efficient management of speculative threads. In our work, we create at the beginning of the program a single speculative thread that persistently occupies a specific hardware context throughout execution. The main computation thread executes on the peer context in the same physical package. Whenever the speculative thread does not perform prefetching, it is periodically throttled, so that it is prevented from running too far ahead of the main computation thread and contending with it for execution resources. In brief, implementing SPR consists of the following steps: identifying delinquent loads, generating code for prefetcher threads, inserting synchronization points between prefetcher and computation threads, and optimizing the final code for better performance.

There are two main issues that need to be addressed in order SPR to be effective. First, prefetcher threads must be timely and accurate, i.e., they must bring the right data at the right time, by keeping always a regulated distance with the computation threads. Second, since most of the processor resources are shared, they must execute with minimal possible interference to the computation threads. In the next sections, we discuss the steps in the SPR implementation process, along with these two requirements and the tradeoffs they raise.

3.2.1 Constructing prefetcher threads

In order to identify top cache-missing memory references, we conducted memory profiling to the original programs using the cachegrind tool from Valgrind simulator [22]. From the profile feedback, we were able to determine and isolate the loads that caused the majority of L2 misses. In most programs, only a few loads were responsible for 92% to 96% of the total misses.

The prefetcher threads were constructed by replicating the original program code and preserving only the backward slices of the target delinquent loads, i.e., the chains of instructions that affect the address computation of these loads. All other instructions were eliminated. To prefetch a load, we use the native Intel prefetch instructions rather than preloading (i.e., "dummy" loads into temporal variables). This is because native prefetch instructions are expected to introduce less interference to the execution pipeline. In fact, it was verified that native prefetch instructions provided better performance over preloading, in the vast majority of delinquent loads.

3.2.2 Synchronizing worker and prefetcher threads

The most important part in SPR implementation is synchronization between prefetcher and computation thread. It affects both the miss coverage ability of the prefetcher thread and the extra overhead it poses to the computation thread. In our programs, almost all delinquent loads reside within loops. Loop nests, in general, provide a convenient structure to apply a producer-consumer execution model between the prefetcher and main computation threads: prefetcher thread runs ahead, brings a certain amount of data in cache, and waits until worker thread starts consuming it.

In order to regulate the runahead distance of the prefetcher thread, we start from posing an upper bound on the amount of data it prefetches each time. This distance has to be sufficiently large, so that the data is prefetched into cache before the computation thread makes use of it, and at the same time it must be kept small enough, so that prefetched lines do not evict data from cache that have not yet been consumed by the computation thread. In our programs, we enforced this upper bound for prefetched data to be about half or less the L2 cache size.

To identify code regions that cover this amount of data, we start from the innermost loop containing a delinquent load and keep traversing to the next outer loop, until the memory references that have been executed so far have memory footprint equal to the imposed bound. The entry and exit points of such *precomputation spans*, imply in fact synchronization points between the worker and prefetcher thread. The shorter these spans are, the more frequent the synchronization becomes between worker and prefetcher and vice versa. Frequent synchronization enables fine-grained control over prefetched data, but entails increased overhead as well. This is the primary reason why we choose to prefetch for the L2 cache rather than the much smaller L1 cache.

We enforce synchronization by enclosing precomputation spans with synchronization barriers. The barrier for the precomputation thread is placed at the exit point of a span, while for the computation thread at the entry point. In this way, the prefetcher thread always runs ahead of the main computation thread. Whenever it has prefetched the expected amount of data but the computation thread has not yet started using it, the prefetcher thread enters its barrier and stops its further progress. It can only continue prefetching the next chunk of data, when it is signaled that the computation thread has started consuming the data, i.e., it has reached its barrier as well. In the general case, and considering their lightweight workload, precomputation threads reach almost always first a barrier, after having prefetched the data within a span. As a result, computation threads need almost never to wait when they reach this barrier. In practice, it was measured that on average only 1.45% of the total execution time of computation threads was spent synchronizing on their barriers. An example of how synchronization is implemented according the above scenario is shown in Fig. 1.

Listing 1. Worker thread

```
1 for (k=0; k<n; k++) {
2     for (ib=0; ib<n; ib+=bs) {
3
4         barrier_wait(&barrier);
5         for (jb=0; jb<n; jb+=bs) {
6             for(i=ib; i<MIN(ib+bs,n); i++) {
7                 for(j=jb; j<MIN(jb+bs,n); j++) {
8                     tc[i][j] = MIN(g[i][k]+g[k][j], g[i][j]);
9                 }
10            }
11        }
12
13    }
14    dtemp = tc;
15    tc = g;
16    g = dtemp;
17 }
```

Listing 2. Prefetcher thread

```
1 g_local=g;
2 tc_local=tc;
3
4 for (k=0; k<n; k++) {
5     for (ib=0; ib<n; ib+=bs) {
6
7         for (jb=0; jb<n; jb+=bs) {
8             for(i=ib; i<MIN(ib+bs,n); i++) {
9                 for(j=jb; j<MIN(jb+bs,n); j++) {
10                    prefetch(&g_local[i][j+64]);
11                }
12            }
13        }
14        barrier_wait(&barrier);
15
16    }
17    dtemp = tc_local;
18    tc_local = g_local;
19    g_local = dtemp;
20 }
```

Fig. 1 Enforcing synchronization between worker and prefetcher thread in TC benchmark. The three innermost loops correspond to a precomputation span. All variables except for those in italics are thread local

3.2.3 Further optimizations

A key factor for the efficiency of SPR, is the efficiency of barriers themselves. The prefetcher thread, which runs ahead and enters first its barrier, must wait on it without disturbing significantly the peer computation thread by consuming shared resources that may be critical for it to make progress efficiently. When the computation thread enters its barrier, the prefetcher thread must resume as fast as possible, again with minimal possible overhead. Consequently, two important requirements regarding barriers are low resource consumption and high responsiveness. In cases where precomputation spans demand frequent synchronization, responsiveness might be more important for SPR to operate efficiently. On the other hand, resource consumption might be an issue for spans that involve few work and long periods of sleep by the prefetcher thread. In our study, we have considered three different barrier implementations in order to evaluate the extent to which each of them meets the aforementioned requirements.

The first implementation is barriers with sense-reversing [23] using spin-wait loops with the `pause` instruction, as described in Sect. 3.1. The thread that waits on the barrier spins repeatedly on a user-level variable until the last thread enters the barrier and changes its value. An advantage of this approach is fast notification and resumption of the waiter thread, due to fast propagation of value changes.

However, as we mentioned in Sect. 3.1, significant execution bandwidth of the processor can be consumed even when the waiter thread simply busy waits on a variable. In order to attack potential performance degradation, we constructed a version of synchronization barriers with spin-loops that make use of the kernel extensions we developed for halting and waking up the logical processors. When a waiter thread (i.e., anyone but the last) reaches the barrier, it enters the spin-wait loop, wherein it executes a `halt` instruction to put its logical processor into halted state. In this way, it goes itself into sleeping mode, offering all of its resources for exclusive use by the sibling thread. The waiter may be woken up periodically by IPIs sent by the OS (e.g., timer interrupts), but will exit the loop and the halted state only when it is notified by the last thread to do so. To encode this notification, the last thread first updates the spin-wait variable and then sends an IPI to the logical processor of the waiter.

Our third approach was the native implementation of barriers in the threading library. The version of Pthreads library we used, makes use of futexes, a mechanism provided by the Linux kernel as a building block for fast userspace locking. Further details on the internals of the mechanism are out of the scope of this paper, however, a good discussion about its concepts and its implications for implementing synchronization primitives is done in [8]. In this implementation of barriers, a waiter thread that enters the barrier makes a `futex` system call with a `FUTEX_WAIT` argument, which causes the thread to be suspended in the kernel. The thread is actually descheduled, and assuming that there are not other runnable processes, all its resources are released and made available to the other thread (i.e., the processor switches from multi-threading to single-threading mode). When the last thread enters the barrier, it calls `futex` with a `FUTEX_WAKE` argument, to wake up and reschedule all waiters on the barrier.

In order to gauge the efficiency of each implementation in terms of both resource consumption and responsiveness, we conducted a simple experiment. On the two con-

Table 2 Performance of various barrier implementations for the purposes of SPR. The second column presents the time required for a worker thread to complete a certain amount of work, while a second thread is waiting on the barrier. The third column shows the time required for the waiter to resume execution after being woken up by the worker when it enters its barrier

Barrier implementation	Average work time in seconds (std. deviation %)	Average wakeup delay in cycles (std. deviation %)
spin-loops	4.274 (±6.86%)	1 395 (±22.03%)
spin-loops w/ halt	3.549 (±6.47%)	3 766 427 (±88.85%)
futex	3.535 (±8.14%)	39 374 (±6.28%)

texts of a Xeon hyper-threaded processor, we executed in parallel two threads, one heavyweight ("worker") and one lightweight ("waiter"). Both perform simple computations on matrices, but the second thread has a workload almost 100 times lighter than the first. At the end of their computations, the threads are synchronized with a barrier. The lightweight thread finishes very quickly its job and enters the barrier, waiting the peer thread for a large portion of its execution time. At first, we measured the time within which the worker thread performs its computations. The larger this time is, the more disturbing is the co-existence of the waiter while it waits on the barrier. Then, we measured the machine cycles required for the waiter to resume its execution after being woken up by the worker. This time is a direct indication of the responsiveness of each implementation. We repeated the above experiment multiple times for each barrier implementation, and present the results in Table 2.

As expected, the first implementation provides best response times, since it does not invoke any OS intervention, but it is the most aggressive in term of resource consumption. The other two approaches introduce much less interference, with the implementation with futexes providing best wakeup times and optimal stability. It seems that the kernel control path for waking up the waiter thread is much shorter for this implementation.

We tested each of the above implementations in our programs, to synchronize computation and prefetcher threads. As we suspected from our previous analysis, the version of barriers with futexes outperformed all other approaches. This is because most delinquent loads in our programs involve little computation and long wait periods, which renders efficient resource management more important than responsiveness. As a future work, we will study the possibility of hybrid implementations, e.g., using barriers with spin-wait loops together with barriers with futexes, as an attempt to attack more effectively loads that entail trivial sleep times and frequent synchronization.

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 the Xeon hyper-threaded processor 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 experimented with different levels of instruction level parallelism, in order to establish lower and upper bounds on the capabilities of the processor for simultaneous execution. 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 of numbers and the vector 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 for each instruction stream: 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 according to the previous discussion, 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. We notice here that for integer mul's and div's, the potential ILP is limited to 1, since for each such instruction the combination of `edx` and `eax` is always implied by the machine ISA as the output register.

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 stand alone 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 processor resources. For example, when the stream of an instruction A is co-executed with the stream of an instruction B, but the CPI of A doesn't increase considerably, that indicates a low rate of contention for resources between the two streams.

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. This can tell us whether it is better to implement a certain degree of parallelism as ILP or TLP, given that both forms of parallelism are operationally equivalent. For example, let's consider a scenario where, in single-threaded and maximum ILP mode, instruction A gives an average CPI of

$C_{1thr\text{-}maxILP}$, while in dual-threaded and medium ILP mode the same instruction gives an average CPI of $C_{2thr\text{-}medILP} > 2 \times C_{1thr\text{-}maxILP}$. 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 2 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. As we will see in the following sections, *fadd-mul* is one such representative instruction mix in our benchmarks. Its corresponding stream is constructed by interleaving within the same thread fp-add's and mul's in an alternating fashion.

Let us 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 ability of the processor to overlap instructions from the two threads, outweighs 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

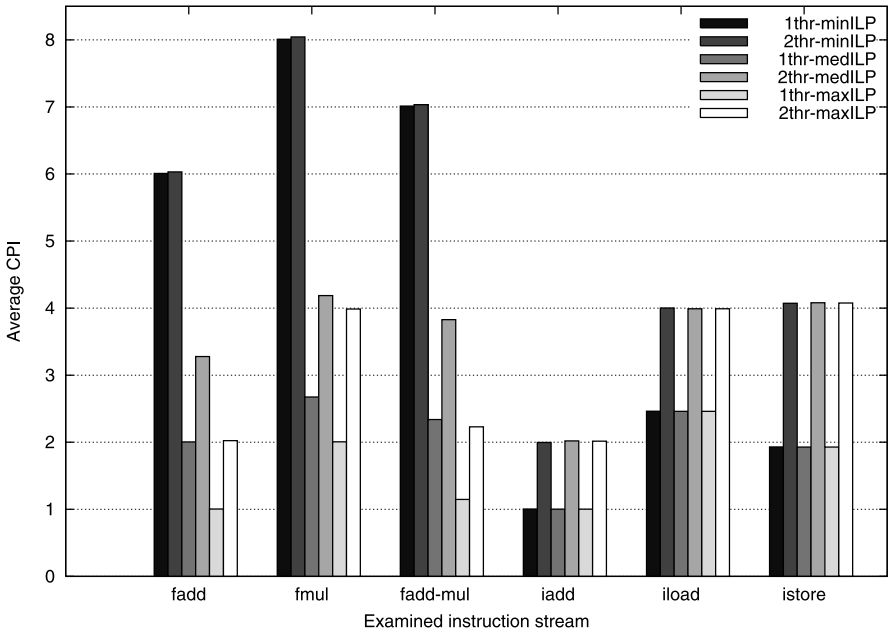


Fig. 2 Average CPI for different TLP and ILP execution modes of some common instruction streams

time than splitting the window in two and assigning each half to two different threads (*2thr-medILP* case). In other words, implementing parallelism as ILP is better than implementing it as TLP for this 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 Fig. 2 shows, *fmul* stream exhibits a similar behavior, as regards the way in which CPI fluctuates between different execution modes. In addition, it is interesting to see that mixing 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 of the instruction remains the same in all cases. Hyper-threading achieved to favor TLP over ILP only in the case of *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 3 and 4 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

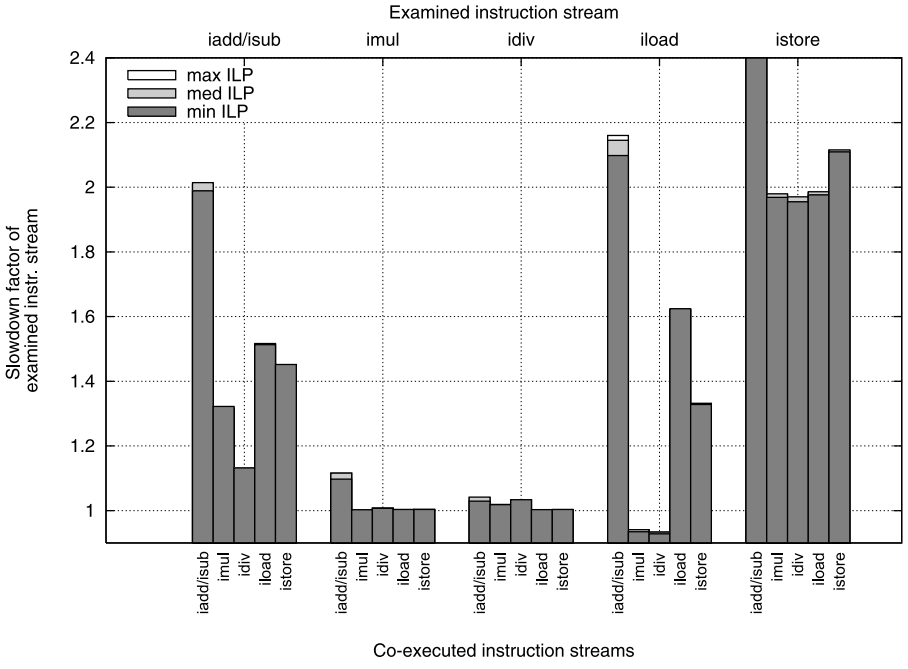


Fig. 3 Slowdown factors from the co-execution of various integer instruction streams

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.

- The execution of *fdiv* instruction streams is mostly affected by streams of the same type rather than streams of different operations. When both threads execute fp-div's, their instructions take each about 120%–140% more time to complete compared to their stand alone execution. Other floating-point operations, however, interact perfectly, causing insignificant slowdown to each other. It seems that there are enough hardware resources to avert any bottleneck for this type of operation. *fdiv* is the only stream that is not affected seriously by 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 (*fmul* causes a significant slowdown of 180%). 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%. This can be a significant delay in the case of parallel benchmarks, especially if we take into account the accessory synchronization overhead added to the hardware bottleneck. We note that if the miss rate of

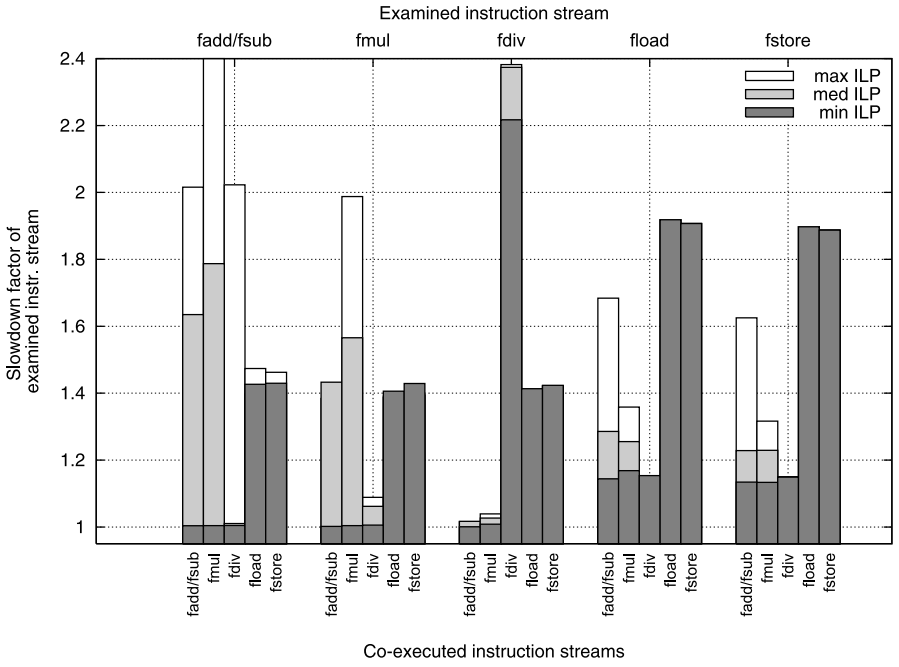


Fig. 4 Slowdown factors from the co-execution of various floating-point instruction streams

load/store instructions increases to 100%, the delay caused to other instruction streams is really minor. However, this is not the case of optimized codes. (Both Figs. 3 and 4 present the results of 3% miss rate for load and store instruction streams.)

- 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.

  *iload*s and especially *istore*s (with 3% miss rate) appear to have a slowdown of 32% and 98%, respectively, when they co-execute with each other. *iadd*/*isub* induce a slowdown of about 115% and 320% to *iload* and *istore* instruction streams, respectively (the latter factor exceeds the boundaries of Fig. 3). If the miss rate increases to 100%, the slowdown is moderated.

Finally, we mixed integer and floating-point instruction streams. Such mixes are more frequent in multiprogrammed workloads or irregular parallel codes, rather than regular ones. We present them here in order to show how various kinds of instructions interact with each other on hyper-threaded processors. The results are depicted in Fig. 5, and concern pairs of floating-point and integer arithmetic streams of the same ILP degree. First, we observe that floating-point streams co-exist with integer streams somehow better than do the latter with the former. Second, each floating-point stream sustains its largest slowdown when executed with its integer counterpart. This phenomenon is somewhat milder in the opposite case. Summarizing Fig. 5, *fadd*/*fsub* is

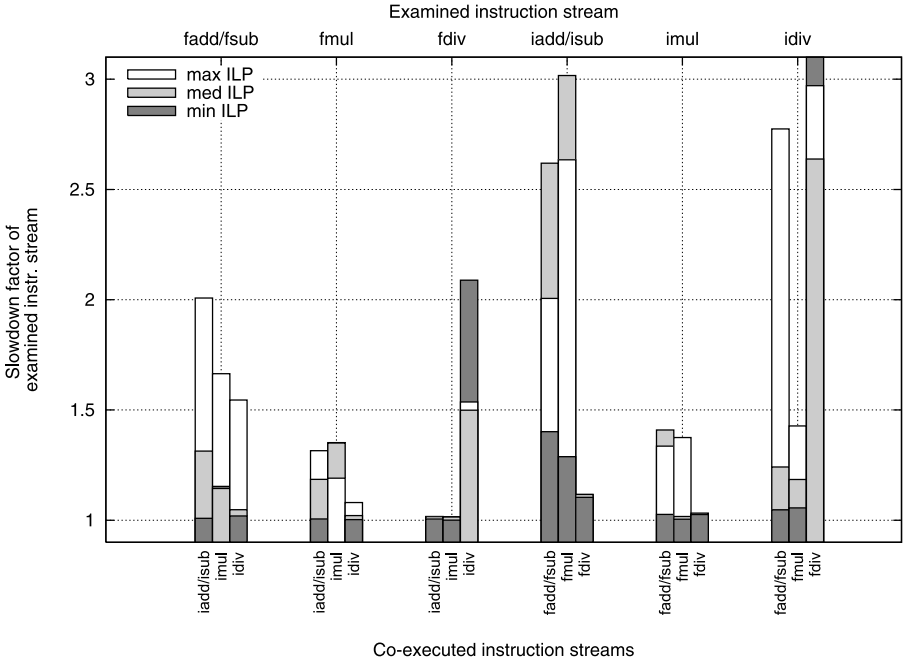


Fig. 5 Slowdown factors from the co-execution of various pairs of floating-point with integer instruction streams

decelerated in the worst case by 100%, *fmul* by 35%, *fdiv* by 108%, while *iadd*/*isub* by 201%, *imul* by 40% and *idiv* by 358%. The smallest slowdowns are generally remarked for minimum ILP, and can be as low as 0%.

5 Experimental framework

5.1 The Xeon hyper-threading architecture

We experimented on an Intel Xeon processor, running at 2.8 GHz. This processor is based on Netburst microarchitecture, and is one of the first mainstream chips to encompass low-end simultaneous multithreading capabilities.

It has an out-of-order, superscalar, speculative core, characterized by its deep pipeline. The core has instruction fetch bandwidth of up to three microoperations (uops) per cycle, dispatch and execution bandwidth of up to six per cycle and retirement bandwidth of up to three per cycle. The branch misprediction penalty is about 20 cycles. There is a 128-entry renamed register file and up to 126 instructions can be in flight at a time. Of those, 48 can be loads and 24 can be stores. It has a 16 KB 8-way set associative L1 data cache with a 2 cycle load-use latency and 64 bytes cache line. The L2 unified cache is 1 MB 8-way set associative, with cache lines of 64 bytes, as well. The equivalent of a typical L1 instruction cache in our system is the execution trace cache. Trace caches do not store instructions but traces of decoded uops. Xeon's trace cache has a capacity of 12 K uops, and is 8-way set associative.

The Xeon processor provides capabilities for software prefetching by special instructions. The programmer can use these instructions to provide hints about the memory locations that are going to be accessed later. The processor also implements a hardware prefetching mechanism by which memory data are transparently prefetched to the unified L2 cache based on prior reference patterns. The hardware prefetcher supports multiple streams and can recognize regular patterns such as forward or backward linear accesses.

Hyper-threading technology [18] makes a single physical processor appear as two logical processors by applying a two-threaded SMT approach. The operating system identifies two different logical processors, each maintaining a separate run queue. In a HT-enabled processor, almost all execution resources are shared: caches of all levels, execution units, instruction fetch, decode, schedule and retirement logic, global history array. On simultaneous requests for shared resources, access is alternated between threads, usually in a fine-grained fashion (cycle-by-cycle). Thus, when both threads are active, the maximum instruction fetch, decode, issue, execution and retirement bandwidth is essentially halved for each of them.

The architectural state in a hyper-threaded processor is replicated for each thread. Replicated are also the instruction pointers, ITLBs, rename logic, return stack and branch history buffers. Buffering queues between pipeline stages, as well as load/store queues, are statically partitioned, so that each thread can use at most half of their entries. In this way, a thread can make forward progress independent and unaffected from the progress of the other thread. The reorder buffer is also partitioned.

As quoted in [28], this static partitioning of hardware resources instead of their dynamic sharing is the primary difference between hyper-threading and the theoretically optimal architecture proposed in [29]. SMT research has argued that dynamic sharing of structures is more effective than static partitioning. However, as we noted, statically partitioned resources deter unoptimized threads from affecting the performance of co-executing threads. Anyway, the disadvantages of static partitioning are minimized with only two hardware contexts, so that the actual behavior is not far from the behavior of the virtual dynamically shared structures.

5.2 Operating system

The operating system that was used for the experiments was Linux version 2.6.13. A decisive factor for the performance of multithreaded applications in a HT-enabled processor is the scheduling algorithm that the operating system uses. Obviously, even if both logical processors of a physical package are identified as different processing units, they must be distinguished from logical processors of different physical packages.

Linux as of 2.6 version has implemented a unified algorithm to handle scheduling of multi-CPU systems in which the processing units have unequal relationships with each other. Such systems are for example SMP machines with HT-enabled processors or NUMA machines. This algorithm is based on what is called scheduling domain. Scheduling domains are sets of processing entities which share properties and scheduling policies. With scheduling domains, the different properties of the logical and physical processing units can be identified and the scheduler can act accordingly.

The Linux scheduler uses the `halt` instruction to boost the performance of the running tasks. When there is only one thread of execution for a physical processor the scheduler uses the `halt` instruction in the idle logical CPU in order to activate the single thread (ST) mode for the processor. In this way, the running thread can take advantage of the full resources of the processor. The `halt` instruction is also used by the scheduler when there are two different tasks with different priorities, each on a logical processor of the same physical package. The task with the highest priority will run in ST mode to fully utilize the resources of the physical package.

5.3 Performance monitoring

Most modern processors provide on-chip hardware for performance monitoring. Intel processors provide mechanisms for selecting, filtering, counting and reading performance events via Model Specific Registers (MSRs). In Xeon processors, one can monitor a rich set of performance events such as cache misses/hits assorted by coherence protocol semantics, TLB misses/hits, branch prediction statistics, instructions/uops statistics and memory bus transactions.

Performance counters help programmers to better understand the issues involved in the execution of a program and identify possible bottlenecks and issues that decrease performance. Performance counters also provide a good insight of the internals of the processor, enabling the programmer to achieve optimal performance.

It is worth noting that with the introduction of the hyper-threading 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. Because the `rdmsr` and `wrmsr` instructions are privileged instructions, this library used a simple device provided by the Linux OS to access the MSR registers. A trivial kernel module was developed to set the PCE flag in CR4 register in order for the library to use the `rdpmc` instruction to read the contents of the performance counters.

6 Performance evaluation

6.1 Benchmarks

The first two benchmarks, BT and CG, are from version 2.3 of NAS Parallel Benchmark suite. **BT** is a simulated CFD application that uses an implicit algorithm to solve 3-D compressible Navier–Stokes equations. It solves block-tridiagonal systems of 5×5 blocks using the finite differences method. The benchmark operates mostly on multi-dimensional arrays and exhibits bad locality. **CG** is a conjugate gradient method, used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix, by solving an unstructured sparse linear system. The benchmark lacks any temporal locality, makes a lot of indirect memory references, but does not suffer from excessive misses probably due to the fact that the matrix is symmetric and the hardware prefetcher can detect the resultant regular accesses.

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 were used for work decomposition and synchronization, instead of OpenMP constructs.

In **HJ** benchmark, the equi-join of two database relations is computed using the *hash-join* algorithm [25], one of the most important operations in database query processing. The algorithm first partitions both relations using a hash function on the join attribute. In this way, the tuples of a partition of one relation need to be checked only against the tuples of the corresponding partition of the other relation, since all tuples in both partitions have the same hash value. Finally, for each such pair of partitions, the algorithm proceeds as follows: it builds a hash table on the partition of the smaller (*build*) relation, and then probes this table using tuples of the larger (*probe*) relation to find matches. Most of the time in the algorithm is spent on hash-lookup operations during this probe phase.

We partitioned a probe relation with 2 M tuples and a build relation with 1 M tuples into 20 partitions each. The relations were created such that every tuple in the build relation matches exactly two tuples in the probe relation. For each build partition, we used a hash table of 500 buckets. Given a uniform and random distribution of join attribute values, this means that each hash bucket hosts a linked list of 100 entries, each pointing to a different build tuple. As a result, the amortized cost for each hash

lookup is 50 pointer jumps. In the TLP implementation, consecutive pairs of build and probe partitions are assigned to threads in a cyclic fashion. Each thread builds a hash table for its build partition, and then probes it using the corresponding probe partition. There are no dependences between different pairs of partitions, so no synchronization between threads is required.

LU is a tiled version of an LU decomposition kernel that operates on a 1024×1024 matrix. In the TLP version, the kernel is parallelized by assigning different tiles to different threads for in-tile factorization. This work partitioning scheme is a coarse-grained one. It is divided in three computation phases, which are determined by inter-tile data dependences of the algorithm. As a result, the use of barrier synchronization is mandated between consecutive phases. Within each phase, the threads can work on different tiles independently, without arising any data hazards.

MM is a tiled and unrolled matrix multiplication between three 1024×1024 matrices, where blocked array layouts have been applied. This means that the order of traversal of the elements in each matrix coincides with the order in which these elements are stored in memory. Tiling along with blocked layouts yield optimal cache performance. In the TLP scheme, consecutive tiles of output matrix C (in block row-major order) are assigned for computation to different threads in a circular fashion, which is a coarse-grained work partitioning scheme. There are not any data dependences, so data consistency is implied and no synchronization mechanisms are required. Furthermore, the two threads work on different cache areas, without interfering in one's another cache lines.

SV is a Sparse matrix-Vector multiplication kernel. We used a $100\,000 \times 100\,000$ matrix, with 150 nonzero elements distributed randomly and uniformly on each row. It is stored using the Compressed Storage Row (CSR) format [3]. According to this format, only the nonzero elements of the matrix are stored in memory (contiguously and in row-major order), and additional arrays are used to index the column of each element in the original matrix, as well as the beginning of each row. As in CG, this yields many indirect memory references. However, SV suffers from many irregular memory accesses in the input vector, because unlike CG, the sparse matrix has completely random structure.

Thread level parallelization of the kernel is performed in terms of data parallel decomposition at the outermost loop level (coarse-grained). In this way, the matrix is partitioned according to its rows, and each thread undertakes a different chunk of rows and computes the corresponding part of the output vector. This scheme does not impose any kind of data dependences. Furthermore, because nonzero elements are distributed uniformly across the matrix, there is no issue of load imbalance between threads. If the matrix had different distribution of nonzeroes, this naive scheme could probably lead to load imbalance, because each part would contain different number of nonzeroes. In this case, a more elaborate scheme that partitions the matrix according to the number of nonzeroes should be applied.

Lastly, **TC** computes the transitive closure of a directed graph with 1600 vertices and 25 000 edges. The graph is represented with an adjacency matrix (dense representation). This problem reduces essentially to an all-pairs shortest-paths problem, which can be solved using the Floyd–Warshall algorithm [6]. The structure of the algorithm is very similar to that of regular matrix multiplication, but one cannot permute the loops in any order, due to dependences imposed by the outermost loop. To

Table 3 Benchmarks under test

Application	Data set
NAS BT	Class A
NAS CG	Class A
Hash-join	2 M probe tuples (232 MB), 1 M build tuples (116 MB), 20 partitions per relation, 500 entry hash-table
LU decomposition	1024×1024, 16×16 blocks
Matrix multiplication	1024×1024, 64×64 blocks
Sparse matrix-Vector multiplication	$100\,000 \times 100\,000$, 150 non-zeroes per row
Transitive closure	1600 vertices, 25 000 edges, 20×20 blocks

improve data reuse and facilitate parallelization, we have implemented a tiled version of the algorithm, by tiling the two innermost loops. In TLP, in each iteration of the outer loop, consecutive tiles of the matrix (in block row-major order) are processed by different threads in a circular fashion. Before threads proceed to the next iteration, they are synchronized with a barrier to satisfy the outermost loop dependence.

Table 3 summarizes the benchmarks and their data sets. Of those benchmarks, BT, HJ and TC had the worst cache performance in their original, serial versions, with local L2 miss rates between 23% and 35%. LU, MM and CG had the best performance, with rates below 0.6%, while SV suffered a miss rate of about 7%.

6.2 Experimental results

In the following sections, we evaluate the performance of the above applications, when applying each of the multithreaded execution schemes. At first, we report measurements from specific events counted by the performance monitoring mechanisms of the processor. Then, we present attained performance for each application and try to explain it through a combined analysis of these performance metrics. We present measurements taken for four events:

- *L2 misses*: The number of second level cache read misses (including load and Request for Ownership—RFO) as seen by the bus unit. For TLP, we show the cumulative number of misses on behalf of both threads. This gives us an indication on whether the threads of a particular application benefit from shared caches or experience conflicts. For SPR, we present only the misses of the worker thread, in order to capture the ability of the prefetcher to cover worker's misses.
- *L1 misses*: The number of first level data cache read misses. Again, for the various execution modes, we present the cumulative number of misses only for processors where worker threads are executed.
- *Resource stall cycles*: The number of clock cycles that a thread stalls in the allocator, waiting until store buffer entries are available. The allocator is the part of the processor which takes uops from the uop queue and allocates to them many of the key machine buffers (e.g., register file entries, ROB entries, load/store buffer entries, instruction queue/memory queue entries). It stalls a logical processor when it tries to use more than the half of partitioned buffers. This performance metric is indicative of the contention that exists between the threads for shared resources.

For the serial execution of each application, we present the ratio of stall cycles over the total number of cycles. For TLP schemes, we present the ratio of stall cycles on behalf of both worker threads, over the total number of cycles in the scheme. When this ratio is significantly increased compared to the serial execution, this designates contention between workers which has as a result both threads to stall for a larger portion of their execution time. For SPR schemes, we present the ratio of stall cycles of the worker thread over the total number of cycles in the scheme. When this ratio is increased compared to serial execution, this means that the worker thread is disturbed by the co-execution of the prefetcher thread.

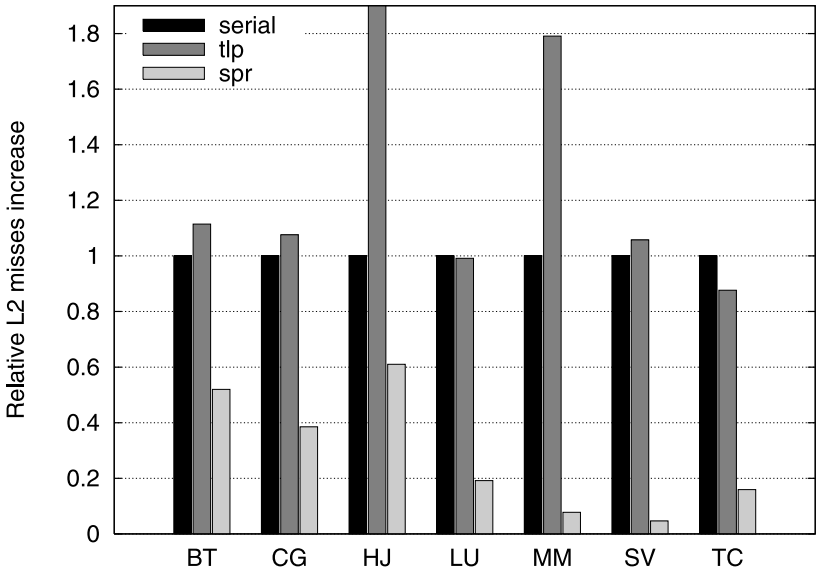
- *Nonbogus uops retired*: The number of nonbogus uops that were retired during the execution of the program. Bogus uops are the ones that were canceled because of a misprediction. For all cases, the uops numbers presented are the numbers of those retired for both threads. For TLP, therefore, this sum expresses the extra instructions due to parallelization and synchronization overhead. For SPR, it additionally expresses the amount of workload of the prefetcher thread.

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. 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 the GNU C compiler version 4.1.2 using the O2 optimization level, and linked against version 2.5 of the GNU C library. We note also that the hardware prefetcher of the processor was enabled throughout all experiments.

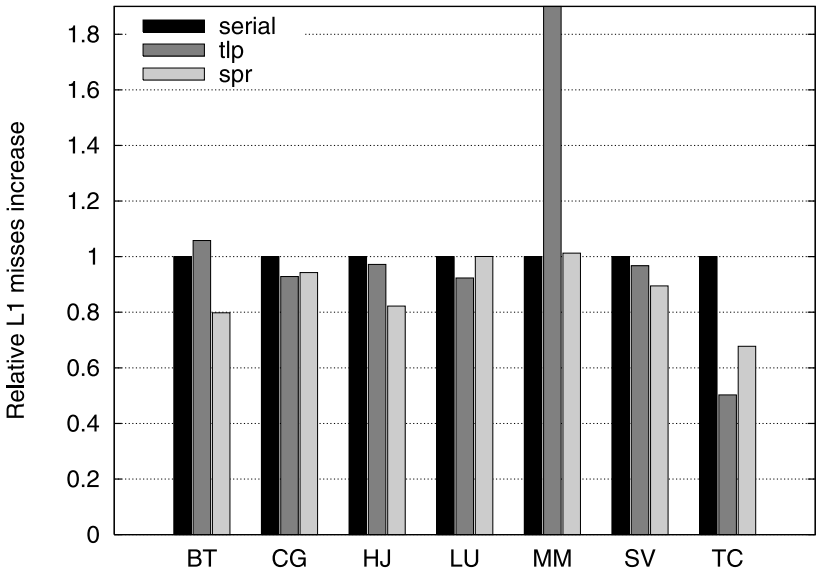
6.2.1 Cache performance

Figure 6a shows the numbers of L2 misses normalized with respect to the serial execution. In TLP schemes, L2 misses on behalf of both threads increase for most applications. In particular, HJ and MM suffer a significant increase by a factor of 2.44 and 1.79, respectively. For HJ, this is because the working sets[1] of the threads are much larger than the L2 cache size, which results in excessive number of cross-thread cache line evictions. This is not the case for MM, however. This benchmark has almost perfect locality, with access patterns easily recognizable by the hardware prefetcher, suffering very few L2 misses (less than 30 000), and the working sets of both threads fit together in L2 cache. Thus, we believe that the increase of L2 misses in this case is due to conflict misses, which could be attacked with a more careful placement of data used by each thread or by applying related techniques such as copying [27]. In the other benchmarks, the increase of L2 misses was below 11%. Of those, TC saw actually a decrease in the cumulative number of L2 misses by 13%, which shows that this benchmark benefits from shared caches and the fact that the worker threads act mutually as prefetchers to some extent.

[1] By *working set* here, we mean the part of overall data on which each thread is working each time. For many of the applications, a thread exhibits some kind of reuse on this data chunk before proceeding to process the next chunk.



(a) L2 misses



(b) L1 misses

Fig. 6 Normalized numbers of L2 and L1 misses with respect to single-threaded execution

In SPR versions, prefetcher threads achieved coverage of L2 misses of worker threads in all benchmarks, by 72% on average. This percentage renders the selection of delinquent loads and the synchronization periods between workers and prefetch-

ers successful. SV enjoyed the largest miss reduction (by almost 96%), and HJ the smallest (by 39%). In general, best coverage rates were achieved for codes that have a few delinquent loads that reside usually within a heavily traversed loop nest (e.g., SV, MM, LU, TC). In addition, these loads have really short backward slices, compared to codes such as HJ where multiple pointer jumps are required to compute the address of a single load.

Figure 6b presents the normalized numbers of L1 cache misses. In TLP implementations, the relative variance of L1 misses with respect to sequential execution was below 7% for five out of seven benchmarks. The notable exceptions were again MM, which suffered an increase of 97%, and TC, which experienced a decrease of 50%. In SPR implementations, the L1 misses of the worker thread for most applications were reduced by less than 20%. In our experience, such small L1 miss divergences hardly can be safe indicators for performance. In general, L1 miss rates are not the dominant performance factor, especially when codes are not highly tuned for locality and are not optimized to take advantage of L1 cache. This is the case for most of our benchmarks, as well, since neither TLP schemes partition the data into chunks that fit in L1, nor SPR schemes target L1 misses.

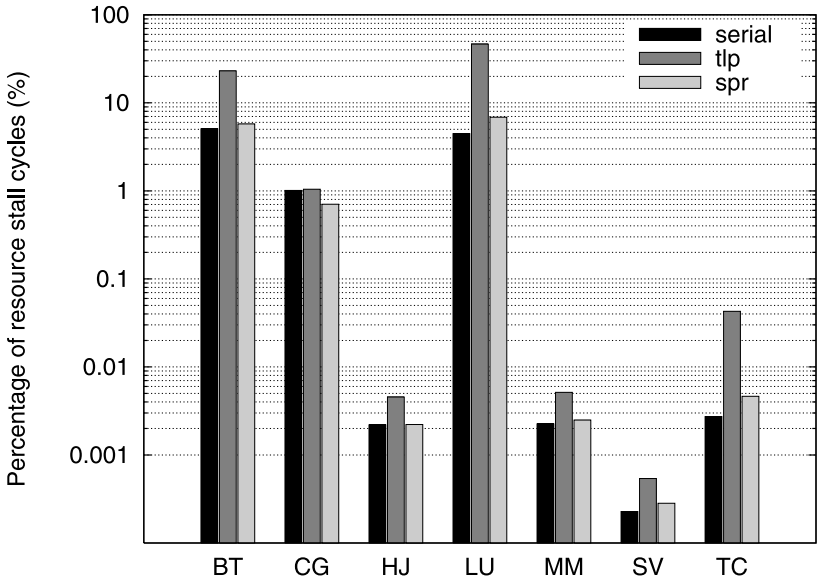
6.2.2 Resource stall cycles

Figure 7a shows for each scheme the fraction of total cycles that corresponds to cycles during which worker threads are stalled by the allocator. For BT and LU benchmarks, stall cycles constitute notable portion of their total cycles (about 5%). In TLP schemes, this portion sees an increase by a factor of 4.5 for BT and 10.4 for LU, which indicates potential contention between worker threads. For the rest benchmarks, the fraction of stall cycles is rather negligible (less than 0.0027% for four of them). On average, TLP schemes experience an increase by a factor of 5.5 in their percentage of stall cycles compared to their single-threaded counterparts. TC sees the largest increase (by a factor of 15.7) and CG the smallest (by a factor of 1.03). On the other hand, the co-execution of prefetcher threads in SPR schemes does not seem to increase significantly the stalls of the worker threads. On average, worker threads suffer an increase by a factor of 1.2. These results verify once again that threads with diverse profiles may co-exist better than threads with symmetric profiles on the contexts of a hyper-threaded processor.

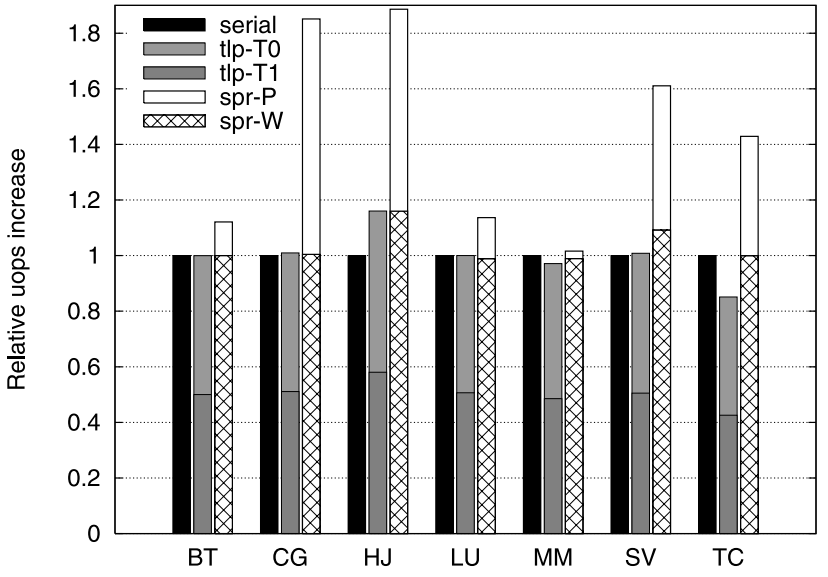
6.2.3 Uops retired

Figure 7b depicts the relative increase of retired uops from both threads. Furthermore, each bar gives a breakdown of uops for all threads participating in a specific execution scheme. *tlp-T0* and *tlp-T1* designate the uops on behalf of each worker thread in TLP. *spr-W* and *spr-P* denote uops attributed to worker and prefetcher threads in SPR, respectively.

In TLP schemes, the total number of retired uops did not increase compared to the sequential execution for most benchmarks. Exceptions were HJ (increase by 16%) and TC (decrease by 15%). In all cases, the initial workload was distributed evenly on both worker threads, as is evident by the almost equal fractions *tlp-T0* and *tlp-T1*.



(a) Resource stall cycles



(b) Non-bogus uops retired

Fig. 7 Ratio of resource stall cycles over total number of cycles, and normalized number of uops with respect to single-threaded execution

In SPR implementations, the cumulative number of uops goes up by 43% on average. As can be seen in Fig. 7b, this increase is attributed mainly to the instructions executed by the precomputation threads. The instructions of worker threads remain the same compared to the serial execution, except for HJ and SV, where they are increased by 16% and 9%, respectively. The largest number of instructions by prefetcher threads were inserted in CG, HJ, TC and SV benchmarks. These instructions account for 42% to 84% of the instructions in the serial execution. The smallest extra overhead was introduced in BT, LU and MM benchmarks, where prefetchers executed a fraction of only 3%–15% of the total instructions in the single-threaded case. In the first group of codes, there are usually many instances of delinquent loads that should be preexecuted by the prefetcher, which has to issue a prefetch instruction for every such instance, or it must execute a rather long chain of instructions until the computation of the delinquent load address. In the other set of codes, there are few delinquent load instances, or a single prefetch instruction suffices to cover multiple instances (e.g., those that occur in adjacent cache lines) as a result of the regularity in the access patterns of these codes.

In order to gain a better understanding on the execution profile of the threads in SPR versions, we measured for each application the fraction of cycles during which the worker and prefetcher threads perform useful computations or sleep on synchronization barriers. This cycle breakdown is presented in Fig. 8. In this diagram, periods of work and wait are denoted as *work* and *synch*, respectively. As expected, small number of instructions executed by the prefetcher threads translate into large wait periods and vice versa. On average, prefetcher threads sleep on barriers for 53% of

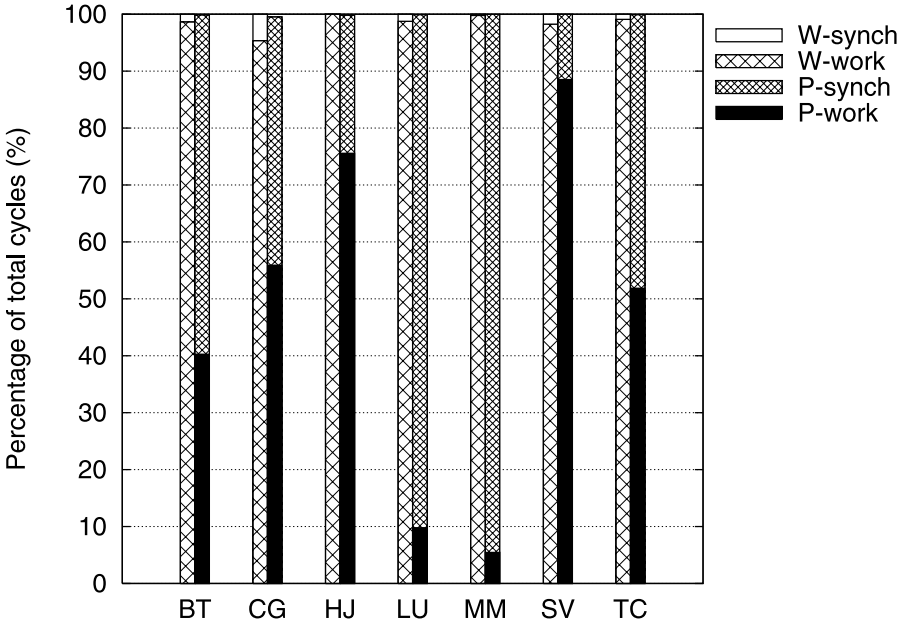


Fig. 8 Cycles breakdown for worker(W) and prefetcher(P) threads in SPR

the total SPR execution time. Of those, prefetcher for MM sleeps the most (94%), and prefetcher for SV the least (12%).

6.2.4 Multithreaded speedups

Figure 9 presents the attained speedups of the multithreaded execution schemes with respect to sequential execution. At a first glance, TLP schemes outperformed SPR implementations in almost all cases. Six out of seven benchmarks enjoyed performance gains with thread-level parallelization. The maximum speedup was 1.36 (TC), and the minimum 1.03 (MM). HJ suffered a slowdown of 1.22. The average speedup for all applications was 1.14. SPR achieved performance improvements in three cases. The maximum speedup attained with this method was 1.34 (TC) and the minimum 1.04 (LU). SPR had the worst performance in CG, where a slowdown of 1.45 was observed.

Combining our previous results and observations on performance metrics and the timing results of Fig. 9, we can deduce that SPR is a promising technique and can accelerate the execution of the main computation thread, in cases where the extra instructions introduced by the prefetcher thread are relatively few, the coverage of L2 misses is good, and the percentage of time during which the prefetcher sleeps, having relinquished its share of resources, is rather large. This is the case of TC, LU and BT benchmarks. MM falls into this category, as well, but its already good locality and the trivial number of L2 misses it suffers do not leave much room for improvement.

On the other hand, even under very good L2 miss coverage, if the extra overhead that the prefetcher introduces with its co-execution is noteworthy, and the time for

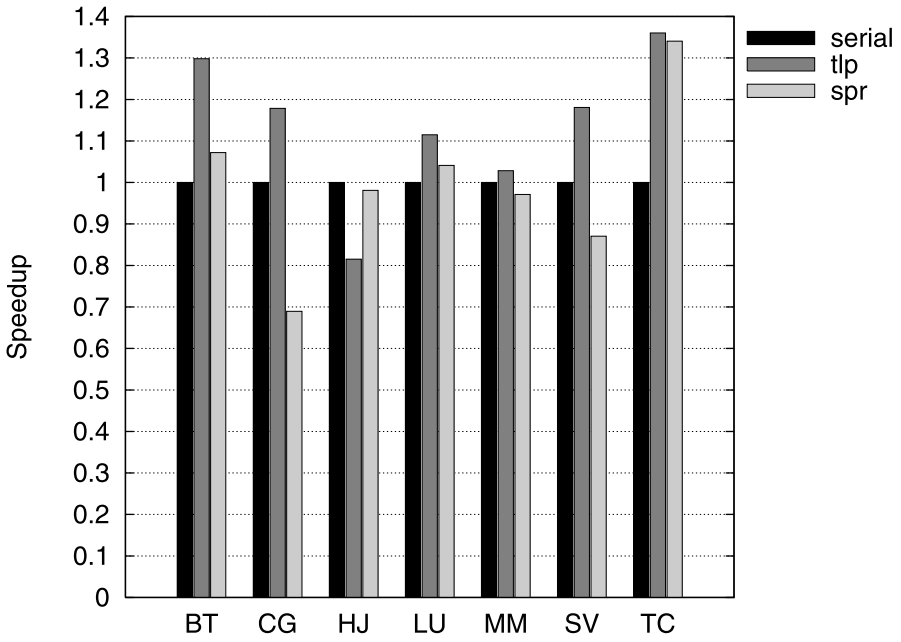


Fig. 9 Speedup of TLP and SPR methods

which it must share processor resources with the worker thread is considerable, SPR can affect performance negatively. This happens for example in SV and CG benchmarks. HJ is an interesting exception. In this code, SPR achieves the smallest coverage of L2 misses, in comparison to all other benchmarks. Furthermore, 90% more instructions are executed with respect to the serial version, and the prefetcher thread does work for at least 75% of its total execution time. Nevertheless, performance degradation is negligible. This motivates us to examine less aggressive SPR schemes, where we could probably be more eclectic in choosing delinquent loads, or even willing to sacrifice some of the successfully covered misses, in order to favor overall performance. In any way, we argue that miss coverage and prefetcher thread overhead constitute a tricky tradeoff which is worthwhile to investigate.

Regarding TLP, our measurements are on par with most previous results in literature which argue that SMT favors mostly codes which are not highly tuned for optimal utilization of functional units or locality. MM is one benchmark that falls in both categories and confirms this assertion. TC, BT and SV, on the other hand, exhibit bad locality and large miss rates, and thus seem to benefit more from SMT. Other significant factors for the efficiency of TLP methods on SMTs are the degree of inter-thread cache interference (prominent in HJ and MM) and the amount of extra instructions due to parallelization or synchronization (HJ). Cache contention in those cases prompts that, even though most programs can be converted into thread-level parallel ones in a straightforward manner; they should be subjected under fine tuning when they are intended to run on SMT systems in order to execute efficiently. Cache blocking techniques or other methods for reducing conflicts could be employed for this purpose.

6.3 Further experimentation

Table 4 presents the utilization of the busiest processor execution subunits when running each of the reference applications. The first column (*serial*) contains results of the initial, single-threaded versions. The second column (*tlp*) presents the behavior of one of the two workers in the TLP implementations. In this case, each of the two threads execute an almost equivalent workload (at about a half of the total instructions of the serial case, ignoring the extra parallelization overhead), and consequently, percentages are nearly identical. The third column (*spr*) presents statistics of the prefetching thread in the SPR versions of our codes (the peer working thread behaves like the single thread of the serial version). 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 [15], and analyzing for each case the breakdown of the dynamic instruction mix, as recorded by the tool. Figure 10 [9] presents the main execution units of the Xeon 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.

As a general observation, all benchmarks are characterized by high percentages of instructions that read from, or write to memory, hence proving their memory intensive nature. Compared to the serial versions, TLP implementations do not generally

Table 4 Processor subunits utilization from the viewpoint of a specific thread

	Execution unit	Instrumented thread		
		serial	tlp	spr
BT	ALU0+ALU1:	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%
	MEM_LOAD:	42.70%	42.70%	44.70%
	MEM_STORE:	16.01%	16.01%	42.94%
CG	ALU0+ALU1:	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%
	MEM_LOAD:	36.51%	45.71%	19.09%
	MEM_STORE:	9.50%	8.51%	9.54%
HJ	ALU0+ALU1:	78.61%	78.65%	79.81%
	MEM_LOAD:	40.13%	40.09%	40.07%
	MEM_STORE:	0.91%	0.91%	0.06%
LU	ALU0+ALU1:	38.84%	38.84%	38.16%
	FP_ADD:	11.15%	11.15%	0.00%
	FP_MUL:	11.15%	11.15%	0.00%
	MEM_LOAD:	49.24%	49.24%	38.40%
	MEM_STORE:	11.24%	11.24%	22.78%
MM	ALU0+ALU1:	27.06%	26.26%	37.56%
	FP_ADD:	11.70%	11.82%	0.00%
	FP_MUL:	11.70%	11.82%	4.13%
	MEM_LOAD:	38.76%	27.00%	58.30%
	MEM_STORE:	12.07%	12.02%	20.75%
SV	ALU0+ALU1:	25.05%	27.58%	24.03%
	FP_ADD:	13.32%	12.90%	0.00%
	FP_MUL:	13.32%	12.90%	0.00%
	MEM_LOAD:	51.15%	46.15%	53.77%
	MEM_STORE:	17.23%	13.27%	0.34%
TC	ALU0+ALU1:	67.14%	67.21%	79.62%
	MEM_LOAD:	40.72%	41.47%	21.93%
	MEM_STORE:	8.55%	8.52%	0.19%

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,

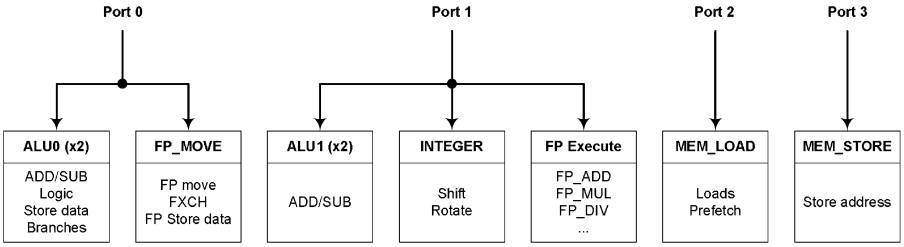


Fig. 10 Instruction issue ports and main execution units of the Xeon processor

different memory access patterns require incomparable effort for address calculations and data prefetching, and subsequently, different number of instructions.

In TLP implementations, BT, SV, MM, LU and CG benchmarks exhibit high usage of floating-point units. The utilization of ALUs is also significant. As seen in Sect. 4.1, *fadd-mul* instruction mixes can co-execute efficiently on a hyper-threaded processor, yielding performance gains in most cases. On the other hand, streams with instructions that utilize ALUs, such as *iadd*'s, do not experience slowdowns, but do not seem to benefit from hyper-threading, as well. Of the above benchmarks, BT is dominated the most by fp-add's and mul's, and the least by instructions that execute on ALUs, and we believe that this mixture is among the main reasons for the noteworthy speedup that was achieved for this benchmark.

Other benchmarks such as MM or LU, where this mixture of instructions is different, enjoyed lower performance gains. For example, in MM benchmark, the most prominent characteristic is the large number of logical instructions used: almost 26% 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, concurrent requests for this unit in the TLP case will lead to serialization of corresponding instructions, without offering any speedup. With respect to MM, LU exhibits higher ALUs usage. In this case, however, instructions can be executed by both ALUs and are distributed equally on them. This explains in a way the better performance of LU. Exception to the above trend is the TC benchmark. It is dominated exclusively by integer operations, but this does not seem to constitute a bottleneck for its performance, since it enjoys a large speedup. However, we believe that hyper-threading exploits the benchmark's bad locality rather its instruction mix, to boost its performance.

As expected in SPR schemes, prefetcher threads execute instructions that apart from memory units, utilize solely ALUs. They execute such instructions at least for the same portion as the worker threads do. This means that when the number of their instructions is comparable to the total instructions of the worker threads, significant pressure is put on ALUs. And as we have discussed, it is difficult to overlap instructions executed on ALUs of a hyper-threaded processor. These facts explain somehow the inability of SPR to accelerate applications such as CG, HJ and SV. TC is an exception again, and the performance improvement that SPR offers can be attributed to its already bad locality and the good miss coverage by the prefetcher.

7 Conclusion and future work

This paper presented performance results for a simultaneous multithreaded architecture, the hyper-threaded Intel microarchitecture, when running single-programmed workloads comprised of memory-intensive applications. In these applications, both work partitioning schemes to exploit thread-level parallelism and speculative precomputation techniques to attack memory latency were considered. Our evaluation was based on timing results and performance measurements obtained from actual program execution, as well as simulation. The results gathered demonstrated the limits in achieving high performance for applications threaded according to TLP or SPR schemes.

Implementing speculative precomputation on a real SMT machine was rather challenging, because it is not a natural choice for the applications we considered, and furthermore, it is not supported in hyper-threaded processors by special hardware facilities assumed in most previous related studies. We had to consider, therefore, various tradeoffs and evaluate different options, in order to make SPR as efficient as possible. In three out of seven benchmarks, SPR achieved speedups between 4% and 34%, and performed comparably to single-threaded execution in most of the rest. Prefetcher threads managed to cover 39%–96% of the L2 misses of the workers, while their execution introduced 43% more instructions on average. We can argue that SPR technique can be effective, when the coverage of L2 misses is rather large, the extra instructions that the prefetcher introduces are few, and the prefetcher is suspended having released its resources for a large portion of its total execution time. In order to fine tune data prefetching in some codes, a considerable number of additional instructions have to be inserted into the pipeline. However, even under such increase in the number of uops, SPR seems promising since it harms performance only by a marginal factor. This motivates the exploration of less aggressive SPR schemes.

With thread-level parallelization, six out of seven applications enjoyed speedups between 3% and 36%. On average, TLP accelerated applications by almost 14%. Our results confirm what most previous studies have shown so far, i.e., that SMT favors mostly programs that are moderately or nonoptimized for locality or resource utilization. Additionally, inter-thread cache conflicts can be a major factor for performance degradation. For this reason, porting thread-level parallel codes to SMT systems should not be effortless, if we opt for performance.

As a future work, we intend to explore the possibility of integrating TLP and SPR schemes in the same application, in order to achieve even better performance gains. Furthermore, we will evaluate the scalability of TLP and SPR in multi-SMT systems (e.g., SMPs of SMTs). With the advent of hybrid multicore and manycore processors, the investigation of hybrid multithreaded techniques to exploit multiple grains of parallelism within a single application seems to be worthwhile, as well.

Of particular interest is the exploration of alternative parallelization techniques that try to avoid resource contention on SMTs. Detecting and co-scheduling phases within an application's lifetime that are complementary in terms of their resource requirements, could be a possible direction. In any case, assigning different types of computations to different threads seems to be a challenge for multithreaded parallel applications running on SMTs.

References

1. Omni OpenMP compiler project (2003) Released in the international conference for high performance computing, networking and storage (SC'03), November 2003
2. Athanasaki E, Koziris N (2004) Fast indexing for blocked array layouts to improve multi-level cache locality. In: Proceedings of the 8th workshop on interaction between compilers and computer architectures (INTERACT'04), held in conjunction with HPCA-10, Madrid, Spain, February 2004, pp 109–119
3. Barrett R, Berry M, Chan T, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, van der Vorst H (1994) Templates for the solution of linear systems: building blocks for iterative methods. SIAM, Philadelphia
4. Bulpin J, Pratt I (2004) Multiprogramming performance of the Pentium 4 with hyper-threading. In: Proceedings of the third annual workshop on duplicating, deconstructing and debunking (WDDD 2004) held in conjunction with ISCA 04, Munich, Germany, June 2004, p 5362
5. Collins J, Wang H, Tullsen D, Hughes C, Lee Y-F, Lavery D, Shen J (2001) Speculative precomputation: long-range prefetching of delinquent loads. In Proceedings of the 28th annual international symposium on computer architecture (ISCA '01), Göteborg, Sweden, July 2001, pp 14–25
6. Cormen T, Leiserson C, Rivest R (2001) Introduction to algorithms. MIT Press, Cambridge
7. Curtis-Maury M, Wang T, Antonopoulos C, Nikolopoulos D (2005) Integrating multiple forms of multithreaded execution on multi-SMT systems: a study with scientific applications. In: ICQES
8. Drepper U (2005) Futexes are tricky. December 2005
9. Intel Corporation. IA-32 Intel architecture optimization. Order Number: 248966-011
10. Intel Corporation (2001) Using spin-loops on Intel Pentium 4 processor and Intel Xeon processor. Order Number: 248674-002, May 2001
11. Kim D, Liao S-W, Wang P, del Cuvillo J, Tian X, Zou X, Wang H, Yeung D, Girkar M, Shen J (2004) Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors. In: Proceedings of the 2nd IEEE/ACM international symposium on code generation and optimization (CGO 2004), San Jose, CA, March 2004, pp 27–38
12. Lo J, Eggers S, Emer J, Levy H, Stamm R, Tullsen D (1997) Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. ACM Trans Comput Syst 15(3):322–354
13. Lo J, Eggers S, Levy H, Parekh S, Tullsen D (1997) Tuning compiler optimizations for simultaneous multithreading. In: Proceedings of the 30th annual ACM/IEEE international symposium on microarchitecture (MICRO-30), Research Triangle Park, NC, December 1997, pp 114–124
14. Luk C-K (2001) Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In: Proceedings of the 28th annual international symposium on computer architecture (ISCA '01), Göteborg, Sweden, July 2001, pp 40–51
15. Luk C-K, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) In: Building customized program analysis tools with dynamic instrumentation. SIGPLAN Not 40(6):190–200
16. Luk C-K, Mowry T (1996) Compiler-based prefetching for recursive data structures. In: Proceedings of the 7th international conference on architectural support for programming languages and operating systems (ASPLOS-VII), Boston, MA, October 1996, pp 222–233
17. Luk C-K, Mowry T (1999) Automatic compiler-inserted prefetching for pointer-based applications. IEEE Trans Comput 48(2):134–141
18. Marr D, Binns F, Hill D, Hinton G, Koufaty D, Miller JA, Upton M (2002) Hyper-threading technology architecture and microarchitecture. Intel Technol J 6:4–15
19. Mitchell N, Carter L, Ferrante J, Tullsen D (1999) ILP versus TLP on SMT. In: Proceedings of the 1999 ACM/IEEE conference on supercomputing (CDROM), November 1999
20. Mowry T (1998) Tolerating latency in multiprocessors through compiler-inserted prefetching. ACM Trans Comput Syst 16(1):55–92
21. Mowry T, Lam M, Gupta A (1992) Design and evaluation of a compiler algorithm for prefetching. In: ASPLOS-V: proceedings of the fifth international conference on architectural support for programming languages and operating systems, New York, NY, USA. ACM Press, New York, pp 62–73
22. Nethercote N, Seward J (2003) Valgrind: a program supervision framework. In: Proceedings of the 3rd workshop on runtime verification (RV'03), Boulder, CO, July 2003
23. Patterson D, Hennessy J (2003) Computer architecture. A quantitative approach, 3rd edn. Kaufmann, Los Altos

24. Roth A, Sohi G (2001) Speculative data-driven Multithreading. In: Proceedings of the 7th international symposium on high performance computer architecture (HPCA '01), Nuevo Leone, Mexico, January 2001, pp 37–48

25. Silberschatz A, Korth H, Sudarshan S (2001) Database systems concepts, 4th edn. McGraw–Hill/ Higher Education, New York

26. Sundaramoorthy K, Purser Z, Rotenberg E (2000) Slipstream processors: improving both performance and fault tolerance. In: Proceddings of the 9th international conference on architectural support for programming languages and operating systems (ASPLOS IX), Cambridge, MA, November 2000, pp 257–268

27. Temam O, Granston E, Jalby W (1993) To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In: Proceedings of the 1993 ACM/IEEE conference on supercomputing (SC'93), Portland, OR, November 1993, pp 410–419

28. Tuck N, Tullsen D (2003) Initial observations of the simultaneous multithreading Pentium 4 processor. In: Proceedings of the 12th international conference on parallel architectures and compilation techniques (PACT '03), New Orleans, LA, September 2003

29. Tullsen D, Eggers S, Emer J, Levy H, Lo J, Stamm R (1996) Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In: Proceedings of the 23rd annual international symposium on computer architecture (ISCA '96), Philadelphia, PA, May 1996, pp 191–202

30. Tullsen D, Eggers S, Levy H (1995) Simultaneous multithreading: maximizing on-chip parallelism. In: Proceedings of the 22nd annual international symposium on computer architecture (ISCA '95), Santa Margherita Ligure, Italy, June 1995, pp 392–403

31. Wang H, Wang P, Weldon RD, Ettinger S, Saito H, Girkar M, Shih S, Liao W, Shen J (2002) Speculative precomputation: exploring the use of multithreading for latency. Intel Technol J 6(1):22–35

32. Wang T, Blagojevic F, Nikolopoulos D (2004) Runtime support for integrating precomputation and thread-level parallelism on simultaneous multithreaded processors. In: Proceddings of the 7th ACM SIGPLAN workshop on languages, compilers, and runtime support for scalable systems (LCR'2004), Houston, TX, October 2004

Evangelia Athanasaki received her Diploma in Electrical and Computer Engineering (2002) and her PhD in Computer Engineering (2006) from the National Technical University of Athens. She is currently a Grid Infrastructure Engineer in the Greek Research and Technology Network (GRNET). Her research interests include Computer Architecture, High Performance Computing and Grid Computing.

Nikos Anastopoulos received his Diploma in Computer Engineering & Informatics from the Computer Engineering & Informatics Department of the University of Patras, Greece. He is currently a PhD candidate in the School of Electrical and Computer Engineering, National Technical University of Athens. His research interests include parallel and high performance computer architectures, multithreaded and multicore processors, programming models and software techniques for the efficient interaction of scientific applications with emerging multiprocessor architectures. He is a student member of the IEEE.

Kornilios Kourtis received his Diploma from the School of Electrical and Computer Engineering of the National Technical University of Athens. He is currently a PhD candidate in the same institution. His research interests include high performance computing, parallel computing and distributed systems. He focuses on software optimization techniques for modern and emerging computer architectures. He is a student member of the IEEE.

Nectarios Koziris received his Diploma in Electrical Engineering from the National Technical University of Athens (NTUA) and his Ph.D. in Computer Engineering from NTUA (1997). He joined the Computer Science Department, School of Electrical and Computer Engineering at the National Technical University of Athens in 1998, where he currently serves as an Assistant Professor. His research interests include Computer Architecture, Parallel Processing, Parallel Architectures (OS and Compiler Support, Loop Compilation Techniques, Automatic Algorithm Mapping and Partitioning) and Communication Architectures for Clusters. He has published more than 60 research papers in international refereed journals and in the proceedings of international conferences and workshops. Nectarios Koziris is a recipient of the IPDPS 2001 best paper award for the paper "Minimising Completion Time for Loop Tiling with Computation and Communication Overlapping". He serves as a reviewer in International Journals (TPDS, JPDC, JSC etc) and as a Program Committee member in various parallel computing conferences (IPDPS, HiPC, ICPP, IPDPS, CAC, PDSEC, SAC etc). He is a member of IEEE Computer Society, member of IEEE TCPP and TCCA, member of ACM and chairs the Greek IEEE Computer Society Chapter. He also serves as Vice-Chairman for the Greek Research and Education Network (GRNET-Greek NREN, www.grnet.gr).