Improving Latency Tolerance of Multithreading through Decoupling

Joan-Manuel Parcerisa and Antonio González, *Member*, *IEEE Computer Society*

Abstract—The increasing hardware complexity of dynamically scheduled superscalar processors may compromise the scalability of this organization to make an efficient use of future increases in transistor budget. SMT processors, designed over a superscalar core, are therefore directly concerned by this problem. This work presents and evaluates a novel processor microarchitecture which combines two paradigms: simultaneous multithreading and access/execute decoupling. Since its decoupled units issue instructions in-order, this architecture is significantly less complex, in terms of critical path delays, than a centralized out-of-order design, and it is more effective for future growth in issue-width and clock speed. We investigate how both techniques complement each other. Since decoupling features an excellent memory latency hiding efficiency, the large amount of parallelism exploited by multithreading may be used to hide the latency of functional units and keep them fully utilized. Our study shows that, by adding decoupling to a multithreaded architecture, fewer threads are needed to achieve maximum throughput. Therefore, in addition to the obvious hardware complexity reduction, it places lower demands on the memory system. Since one of the problems of multithreading is the degradation of the memory system performance, both in terms of miss latency and bandwidth requirements, this improvement becomes critical for high miss latencies, where bandwidth might become a bottleneck. Finally, although it may seem rather surprising, our study reveals that multithreading by itself exhibits little memory latency tolerance. Our results suggest that most of the latency hiding effectiveness of SMT architectures comes from the dynamic scheduling. On the other hand, decoupling is very effective at hiding memory latency. An increase in the cache miss penalty from 1 to 32 cycles reduces the performance of a 4-context multithreaded decoupled processor by less than 2 percent. For the nondecoupled multithreaded processor, the loss of performance is about 23 percent.

Index Terms—Access/execute decoupling, simultaneous multithreading, latency hiding, instruction-level parallelism, hardware complexity.

1 Introduction

The gap between the speeds of processors and memories has kept increasing in the past decade and it is expected to sustain the same trend in the near future. This divergence implies, in terms of clock cycles, an increasing latency of those memory operations that cross the chip boundaries. In addition, processors keep on growing their capabilities to exploit parallelism by means of greater issue widths and deeper pipelines, which makes the negative impact of memory latencies on performance even higher. To alleviate this problem, most current processors devote a high fraction of their transistors to on-chip caches in order to reduce the average memory access time.

Some processors, commonly known as out-of-order processors [8], [9], [17], [19], [41], include dynamic scheduling techniques, most of them based on Tomasulo's algorithm [35] or variations of it. These processors tolerate both memory and functional unit latencies by overlapping them with useful computations of other independent instructions which are found by looking ahead in the instruction stream inside a limited instruction window. This is a general scheduling mechanism that dynamically extracts the instruction parallelism available in the instruction window.

• *The authors are with the Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain.*
E-mail: {jmanel, antonio}@ac.upc.es.

Manuscript received 19 Jan. 2001; accepted 16 Mar. 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 113505.

As memory latency continues to grow in the future, out-of-order processors will need larger instruction windows to find independent instructions to fill the increasing number of empty issue slots and this number will grow even faster with greater issue widths. The increase in the instruction window size will have an obvious influence on the chip area, but its major negative impact will strike at the processor clock cycle time. As reported recently [21], the issue and the bypass logic and, also—although to a lesser extent—the renaming circuitry, are in the critical path that determines the clock cycle time. In their analysis, the authors of that study state that the delay function of these networks increases quadratically with the issue width and window length. Furthermore, since wire delays remain constant as feature sizes shrink, these latencies will not scale down in future process technologies. According to these results, some authors have recently proposed several superscalar architectures which address the clock cycle problem by partitioning critical components of the architecture and/or providing less complex scheduling mechanisms [6], [15], [21], [24], [31], [42]. These architectures follow different partitioning strategies and implement different instruction issue schemes, either in-order or out-of-order.

This work focuses on a particular partitioning paradigm called access/execute decoupling. Decoupling was first proposed for early scalar architectures to provide them with dual issue and a limited form of dynamic scheduling that has low complexity and is especially oriented to tolerate memory latency. Typically, a decoupled access/execute architecture [2], [7], [11], [23], [27], [28], [39], [40] splits,

either statically or dynamically, the instruction stream into two. The access stream is composed of those instructions involved in the fetch of data from memory, while the execute stream is formed by the instructions that consume these data and perform the actual computations. These streams execute in different processing units, which are called Access Processor (AP) and Execute Processor (EP), respectively, in this paper. Although each processing unit issues instructions in-order, both units are allowed to run asynchronously, one with respect to the other. As far as the AP manages to go ahead of the EP, data from memory is effectively prefetched into the appropriate buffering storage so that the EP consumes it without getting stalled.

One of the main arguments for the decoupled approach is the reduced issue logic complexity. In this model, several instructions per cycle are issued in-order within each processing unit. Such a decoupled architecture adapts to higher memory latencies by scaling much simpler structures than an out-of-order, i.e., scaling at a lower hardware cost or, conversely, scaling at a higher degree with similar cost. Therefore, we believe that decoupled access/execute architectures can progressively regain interest as issue width and memory latency keep on growing and demanding larger instruction windows because these trends will make it worth trading issue complexity for clock speed.

On the other hand, simultaneous multithreading [37], [38] has been shown to be an effective technique to boost ILP. In this paper, we analyze its potential when implemented on a decoupled processor core. We present a multithreaded architecture where each thread executes in access/execute decoupled mode. That is, after being decoded, each instruction is dynamically steered either to the AP or to the EP processing units and the instructions of a given thread in each processing unit are issued in-order. All the threads are active simultaneously and they compete for the issue slots in each processing unit so that instructions from different contexts can be issued in the same cycle. For the rest of this paper, we will refer to it either as simultaneous multithreading or, simply, multithreading, for short.

We show in this paper that multithreading by itself, i.e., without dynamic scheduling support, has little effect in hiding a high memory latency. Instead, decoupling provides an excellent memory latency tolerance. Therefore, the combination of decoupling and mulithreading takes advantage of their best features: While decoupling is a simple but effective technique for hiding a high memory latency with less issue logic complexity than out-of-order, multithreading provides enough parallelism to hide the latency of the functional units and to keep them busy. In addition, multithreading also contributes to hiding memory latency when a program decouples badly. However, since decoupling hides most memory latency, few threads are needed to keep the functional units busy and achieve a near-peak issue rate. This is an important result since decreasing the number of threads reduces the memory pressure produced by the large combined working sets, which has been reported to be a major bottleneck in multithreading architectures, and reduces the hardware cost and complexity.

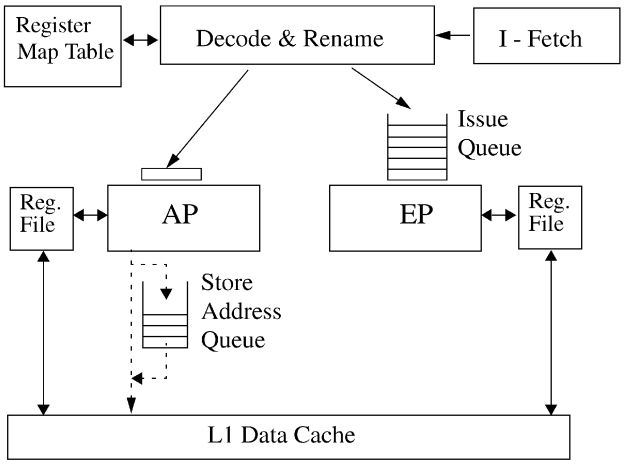

Fig. 1. Scheme of the single-threaded decoupled processor.

The rest of this paper is organized as follows: Section 2 describes a single-threaded decoupled architecture. In Section 3, the latency hiding effectiveness of decoupling is evaluated. Section 4 describes and evaluates the proposed multithreaded decoupled architecture. Finally, in Section 5, we summarize the main conclusions.

2 Overview of a Single-Threaded Decoupled Architecture

In this section, the single-threaded decoupled architecture is described that will be assumed throughout the rest of the paper (Fig. 1). Its main architectural parameters are summarized in Table 1. Later on, in Section 4, the multithreaded decoupled architecture is described which consists of some extensions of this model.

The single-threaded decoupled architecture fetches instructions from a single stream (it has a single PC). Then, the program is dynamically split into two streams which are dispatched to two superscalar, in-order issue, decoupled processing units: the Access Processing unit (AP) and the Execute Processing unit (EP), each having separate physical register files, functional units, and datapaths. Precise exceptions are supported by means of a reorder buffer, a graduation mechanism, and a register renaming map table [12], [29].

The fetch logic fetches four instructions per cycle (or up to the first taken branch) and it includes a bimodal branch predictor with 2K 2-bit counters [26] updated at branch resolution. After being decoded and renamed, up to four instructions of any kind are dispatched to the AP or to the EP, according to their data type, i.e., integer and memory instructions are dispatched to the AP while FP instructions are sent to the EP, which is the same approach as that of other decoupled processors, like the ZS-1 [28] or the MIPS R8000 [11]. Although this rather simplistic dynamic partitioning scheme mostly benefits numerical programs, it still provides a basis for our study, which is mainly focused on the latency hiding potential of decoupling and its synergy with multithreading. Recent studies [3], [22], [25] have proposed other alternative partitioning schemes that address the decoupling of integer codes, but they are

TABLE 1
Default Single-Threaded Architecture Parameters

Parameter	AP	EP
Fetch, Decode/Rename width	up to 4 instructions	
Issue width	2	2
Branch predictor	bimodal, 2K 2-bit counters	
IQ size	4	48
SAQ size	32	
Functional units count	2	2
Functional units latency	1	4
Physical Registers	64	96
L1 I-cache	infinite	
L1 D-cache	64 KB, direct mapped, write-back, 32 byte lines, 1 cycle hit time, 16 primary outstanding misses, 2 R/W ports	
L2 off chip cache	infinite, 16 cycles hit time, 16 bytes/cycle bus bandwidth	

not considered here. The instructions dispatched to the EP are buffered into a long 48-entry FIFO Issue Queue (IQ), while those dispatched to the AP are put in a small 4-entry FIFO queue. If the target queue for a dispatched instruction is full, the dispatch stalls. The 48-entry IQ in the EP provides enough storage to allow the AP to run ahead of the EP without blocking the dispatch.

In each processing unit, two instructions per cycle are issued in-order, to the functional units, that are general purpose and fully pipelined. To better exploit the parallelism between the AP and the EP, the instructions can issue and execute speculatively beyond up to four unresolved branches (like the MIPS R10000 [41] or the PowerPC 620 [19]). This feature may sometimes become a key factor to enable the AP to slip ahead of the EP. After having their address calculated, stores are held in the Store Address Queue (SAQ) for disambiguation until they graduate, allowing nonmatching loads to bypass stores. Whenever a matching pair is found, the data from the pending store is immediately forwarded to the load if it is available. Otherwise, the load is put aside until this data is forwarded to it, without blocking the pipeline.

In many decoupled processors ([2], [20], [28] among others), data fetched from memory is buffered into a load data queue. In these architectures, either the compiler or the dispatch logic must generate a duplicate of the load instruction for the EP to read the operand from the load data queue head and copy it to a register. Since we found in preliminary experiments (not shown here) that such code duplication would significantly reduce performance, it is avoided by implementing dynamic register renaming. That is, data fetched from memory is written into a physical register rather than a data queue, eliminating the need for copying. It is also a convenient way to manage the disordered completion of loads when a lockup-free cache is present. Duplication of conditional branch instructions, also used in [28] to communicate branch outcomes between processing units, is not needed if the processor includes support for control speculation and recovery and it can identify the instructions to squash in case of a misprediction.

The primary data cache is on-chip, with two R/W ports [32], direct-mapped, 64 KB-sized, with a 32 byte line length, write-allocate, and it implements a write-back policy to minimize off-chip bus traffic. It is a lockup-free cache, with its Miss Status Hold Registers [16] modeled similarly to the MAF of the Alpha 21164 [5]. It can hold up to 16 outstanding (primary) misses to different lines, each capable of merging up to four (secondary) misses per pending line. We assume that L1 cache misses always hit in an infinite multibanked off-chip L2 cache and they have a 16-cycle latency plus any penalty due to the contention of the L1-L2 bus, which is modeled in detail. This is a fast 128-bit wide data bus, operating at full processor frequency, thus capable of delivering 16 bytes per cycle, like that of the R10000 (the bus is busy during two cycles for each line that is fetched or copied back).

3 The Latency Hiding Effectiveness of Decoupling

Since the interest in decoupling is closely related to its ability to hide memory latency without resorting to other more complex issue mechanisms, we have first quantified such ability for a wide range of L2 cache latencies, from one to 256 cycles. Other similar studies on decoupled machines have been reported before [1], [7], [13], [18], [27], [28], [30], [39], [40], but they did not incorporate techniques like store-load forwarding, control speculation or lockup-free caches. We have evaluated a single-threaded decoupled architecture, as described in Section 2, with all the architectural parameters shown in Table 1. However, for these experiments, the sizes of all the architectural queues and physical register files are scaled up proportionally to the L2 latency (e.g., when doubling the L2 latency from 16 to 32 cycles, we also doubled the sizes of the IQ in EP, the SAQ, the ROB, and the number of outstanding misses and renaming registers). Of course, such scaling—especially the register

file—may have implications in the cycle time that should be handled by considering other partitioned layouts [21], but the concern of this study is about exploiting access/execute parallelism to tolerate memory latency.

3.1 Experimental Framework

The experiments consisted of a set of trace driven cycle-by-cycle simulations of the SPEC FP95 benchmark suite [34], fed with their *ref* input data sets. The programs were generated with the Compaq f77 compiler, applying full optimizations, for an AlphaStation 600 5/266. The traces were obtained by instrumenting the binaries with the ATOM tool [33]. Since the simulator is very slow, due to the detail of the simulations, we run only a portion of 100 M (million) instructions of each benchmark, after skipping an initial start-up phase. To determine the length of this initial portion to discard, we compared the instruction-type frequencies of such a fragment, starting at different initial points, with the frequencies measured in a full run. We found that the start-up phase does not have the same length for all the benchmarks: about 5,000 M instructions for 101.tomcatv and 103.su2cor, 1,000 M for 104.hydro2d and 146.wave5, and less than 100 M for the rest.

The simulator assumes an infinite I-cache. Notice that I-cache miss ratios for SPEC FP95 are usually very low, so this approximation introduces a small perturbation. Due to the trace-driven nature of the simulator, branch mispredictions are modeled by stalling the fetch until the branch is resolved and, therefore, cache pollution effects are not taken into account. In contrast, handling of load and store misses is accurately modeled cycle by cycle to reflect the MAF-like behavior, the availability of the L1-L2 bus, and the time when line-refill and copy-back transactions gain access to the L2 cache, and when they complete. Besides this, for higher accuracy, the L1 cache tags are not updated immediately at tag probe time, but at the time the line replacement actually takes place.

3.2 Results

Load miss latency is hidden by overlapping it with useful computations and also with the latency of subsequent misses, provided that these computations do not need the missing data. This overlapping is naturally achieved for decoupled loads (FP loads, in our implementation) when the AP runs far enough ahead of the EP. On the other hand, the latency of two integer missing loads (not decoupled by our implementation) may also overlap if the two loads are scheduled prior to their first uses. However, since the AP schedules the issue of integer loads in order, their latency overlapping depends entirely on the static scheduling.

The anticipation of the AP is lost due to the so-called "loss of decoupling" events: The AP stalls caused by integer load misses fall into this category, but it also includes several control and memory data dependences that force the AP and the EP to synchronize [2], [36]. For instance, data dependence synchronization occurs in some particle-in-cell codes like *wave5*, where data generated by the EP is used by the AP to compute array indices. Control dependence synchronization is caused by mispredicted conditional branches, especially those of FP branches, since they completely drain the pipeline. Codes with many such events are said to have a bad decoupling behavior.

The nonhidden memory latency can be measured as the number of stall cycles that a load miss causes. For this purpose, in addition to the IPC, we have measured the average "perceived" latency of integer and FP load misses separately. We define the perceived latency of a load as the number of stall cycles that it causes to the first instruction I_d that uses its data. That is, the number of cycles between the time I_d is first considered for issue (when it reaches the head of the Issue Queue) and the time the memory operand is actually delivered (a similar definition is found in [2]). The simulator measures the perceived latency of every load by identifying and time-stamping its target register. The average perceived latency of hits and misses depends on the miss ratio. In contrast, the average perceived latency of load misses does not depend on the miss ratio, but only depends on whether the anticipation or decoupling of the AP over the EP is large enough to hide the latency and, therefore, this metric characterizes the "decoupling behavior" of each program.

Fig. 2a and Fig. 2b depict the perceived load miss latency for FP and integer loads, respectively. As shown in Fig. 2a, except for *fpppp*, more than 96 percent of the FP load miss latency is always hidden. Fig. 2b shows that *fpppp*, *su2cor*, *turb3d*, and *wave5* are the programs with the highest perceived integer load miss latency. Notice that, as discussed above, integer load misses have quite high perceived latencies because these loads are not decoupled (their dependent instructions are also executed in the AP), so their latency tolerance relies exclusively on the scheduling ability of the compiler.

The most appropriate approach to measure the latency tolerance of an architecture is by analyzing the impact of memory latency on performance and it depends on both the number of load misses and their average penalty. Fig. 2c shows the load miss ratios for the configurations at both ends of the considered latency range: one and 256 cycles. Note that, in some cases, the increase of L2 latency substantially increases the miss ratio due to the late updates of the L1 cache because, after a pending miss, subsequent loads to the same line are more likely to produce new misses. These new misses do not necessarily increase the number of requests to the L2 cache if the hardware can merge them in a single request.

Fig. 2d shows the IPC loss of each configuration, relative to the 1-cycle L2 latency case, and their absolute IPC values are tabulated in Table 2. We can see in Fig. 2d that, for *tomcatv*, *swim*, *mgrid*, *applu*, and *apsi*, in spite of having substantial miss ratios, their performance is hardly degraded due to their good decoupling behavior. In addition, programs like *fpppp* or *turb3d* with quite bad decoupling behavior are also little performance degraded due to their extremely low miss ratios. On the other hand, the most performance degraded programs are those with both high perceived miss latencies and significant miss ratios: *hydro2d*, *wave5*, and *su2cor*.

To summarize, performance is very little affected by the L2 latency when either it can be hidden efficiently (well-decoupled programs like *tomcatv*, *swim*, *mgrid*, *applu*, and

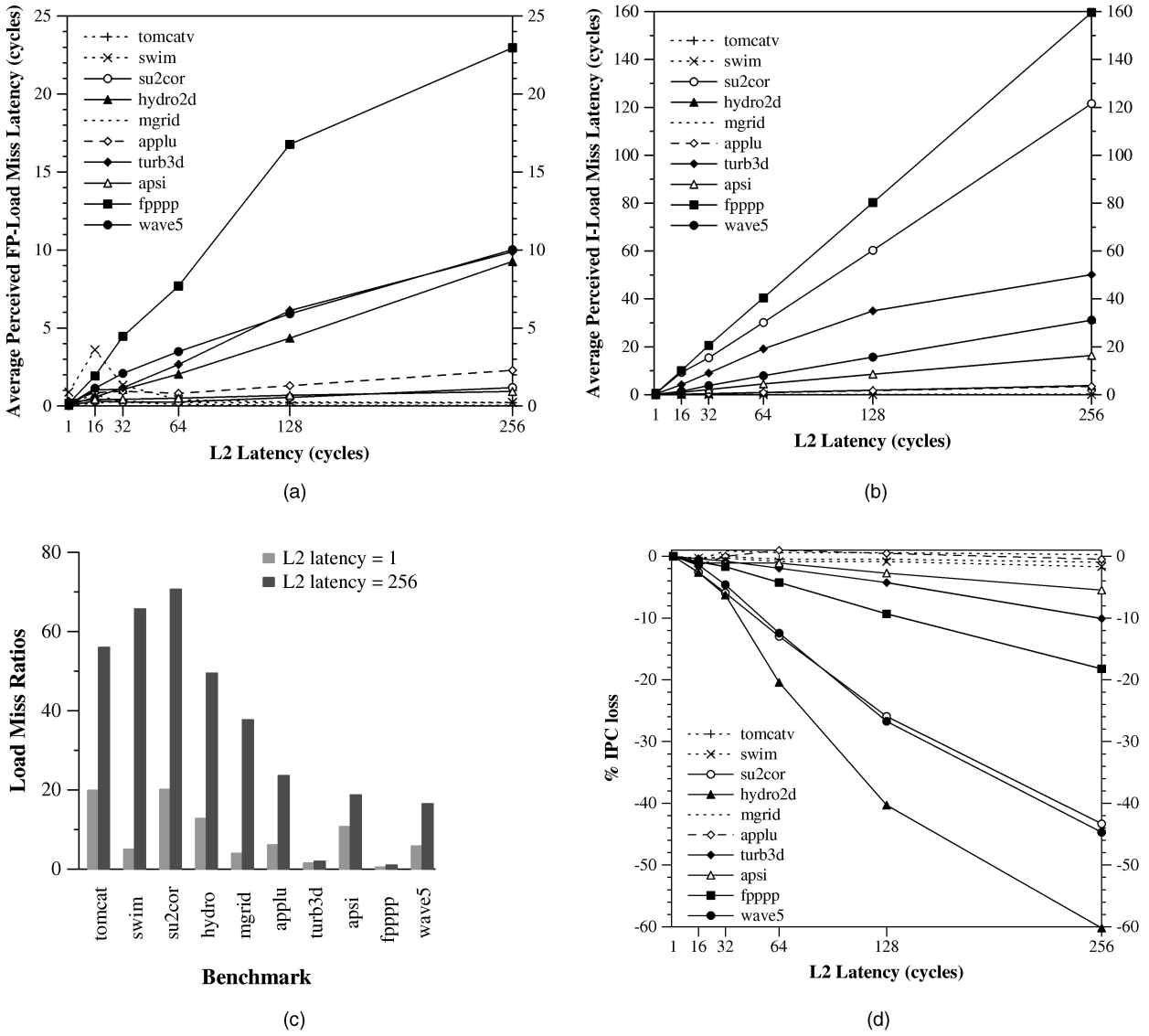

Fig. 2. (a) Perceived miss latency of FP loads. (b) Perceived miss latency of Integer loads. (c) Miss ratios of Loads and Stores when L2 latency is either one or 256 cycles. (d) Impact of latency on performance (loss relative to the 1-cycle L2 latency case).

apsi) or when the miss ratio is low (*fpppp* and *turb3d*), but it is seriously degraded for programs that lack both features (*su2cor*, *wave5*, and *hydro2d*). On our decoupled architecture implementation, the hidden miss latency of FP loads depends on the degree of program decoupling, while that of integer loads relies exclusively on the static instruction scheduling.

TABLE 2
IPCs for Several L2 Latencies

Latency	1	16	32	64	128	256
tomcatv	2.11	2.10	2.11	2.10	2.10	2.09
swim	3.54	3.53	3.53	3.51	3.51	3.48
su2cor	2.70	2.63	2.54	2.35	2.00	1.53
hydro2d	1.91	1.86	1.79	1.52	1.14	0.76
mgrid	3.44	3.42	3.47	3.46	3.46	3.45
applu	2.07	2.04	2.07	2.09	2.08	2.06
turb3d	2.58	2.57	2.56	2.53	2.47	2.32
apsi	1.83	1.81	1.81	1.81	1.78	1.73
fpppp	2.36	2.34	2.32	2.26	2.14	1.93
wave5	2.17	2.14	2.07	1.90	1.59	1.20

4 A Multithreaded Decoupled Architecture

In the experiments of the previous section, we also analyzed the causes that prevent the EP from filling the issue slots and found that the latency of the functional units caused more wasted issue slots (30 percent to 25 percent of the issue slots for L2 latencies of one to 256 cycles) than the memory latency (3 percent to 20 percent) and it was the most important source of wasted issue slots. In other words, this observation suggests that the in-order issue policy imposed on the EP has little tolerance to the multicycle latency of the EP functional units.

Simultaneous multithreading (SMT) is a dynamic scheduling technique that increases processor throughput by exploiting thread level parallelism. Multiple simultaneously

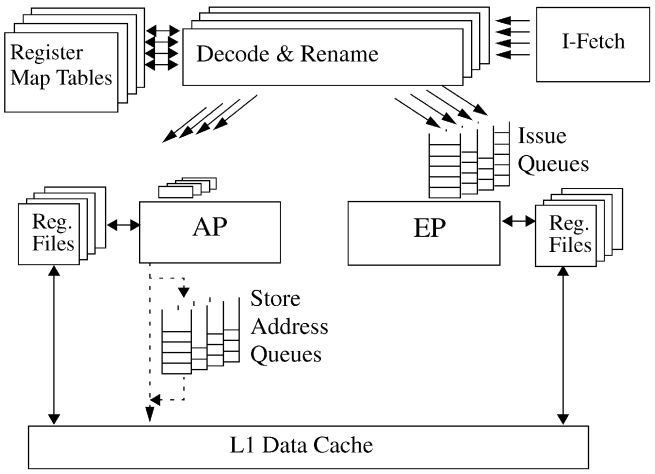

Fig. 3. Scheme of the multithreaded decoupled processor.

active contexts compete for issue slots and functional units. Previous studies of SMT assumed several dynamic instruction scheduling mechanisms ([4], [10], [37], [38], among others) other than decoupling. In this paper, we analyze its potential when implemented on a decoupled processor, i.e., each thread context executes in a decoupled mode, as described in the previous section. We still refer to it as simultaneous, although there are obvious substantial differences from the original SMT, because it retains the key concept of issuing from different threads during a single cycle. Since decoupling provides excellent memory latency tolerance and multithreading supplies enough amounts of parallelism to remove the remaining stalls, we expect important synergistic effects in a microarchitecture which combines these two techniques. In this section, we present and evaluate the performance and memory latency tolerance of the multithreaded decoupled access/execute architecture and we analyze the mutual benefits of both techniques, especially when the miss latency is large.

4.1 Architecture Overview

Our proposal is a multithreaded decoupled architecture (Fig. 3). That is, each thread executes in a decoupled mode, sharing the functional units and caches with other threads. The multithreaded decoupled architecture is based on the decoupled design described in Section 2 and Table 1, with some extensions: It can run up to six independent threads and issue up to eight instructions per cycle (four at the AP and four at the EP) to eight functional units. The L1 lockup-free data cache is augmented to four ports. The fetch and decode stages—including branch prediction and register map tables—and the register files and queues—including the ROB, the issue queues, and the SAQ—are replicated for each context. The issue logic, the functional units, and the data cache are shared by all the threads. There is no thread communication mechanism other than memory since, in this work, we consider only independent threads.

In our model, all the threads are allowed to compete for each of the eight issue slots each cycle and priorities among them are determined in pure round-robin order (similar to the *full simultaneous issue* scheme reported in [38]). Each cycle, only two threads have access to the I- cache and each of them can fetch up to eight consecutive instructions (up to the first taken branch). The chosen threads are those with fewer instructions pending to be dispatched (similar to the RR-2.8 with I-COUNT schemes, reported in [37]).

4.2 Experimental Evaluation

The multithreaded decoupled simulator is fed with t different traces, corresponding to t independent threads. The trace of every thread is built by concatenating the first 10 million instructions of the 10 traces used in the previous section—each thread using a different permutation—thus totaling 100 million instructions per thread. In this way, all threads have different traces but balanced workloads, similar miss-ratios, etc.

4.3 Wasted Issue Slots

Fig. 4 shows the breakdown of wasted issue slots when varying the number of threads from one to six. Each cycle, the number of wasted issue slots is recorded, along with the cause that prevents each individual thread from filling them, obtaining a per thread issue slot breakdown. These results are then averaged among the running threads to obtain the graphs of Fig. 4. The main causes that make a thread lose an issue slot are having the issue queue empty (labeled *empty IQ*) and having any operand unavailable either because it is the result of a previous load (labeled *memory latency*) or an instruction other than a load (labeled *FU's latency*).

The first column in Fig. 4 represents the case with a single thread and it shows that the major bottleneck is caused by the EP functional units latency in accordance with similar results observed on the single threaded decoupled architecture (mentioned at the beginning of Section 4). When two more contexts are added, the multithreading mechanism drastically reduces these stalls in both units and produces a 2.31 speed-up (from 2.68 IPC to 6.19 IPC). Since the AP functional units are nearly saturated (90.7 percent) for three threads, negligible additional speed-ups are obtained by adding more contexts (6.65 IPC is achieved with four threads).

Notice that, although the AP almost achieves its maximum throughput, the EP functional units do not saturate due to the load imbalance between the AP and the EP. Therefore, the effective peak performance is reduced by 17 percent, from eight IPC to 6.65 IPC. This problem could be addressed with a different choice of the number of functional units in each processor unit, but this is beyond the scope of this study.

Another important remark is that, when the number of threads is increased, the combined working set is larger and the miss ratios increase progressively, putting greater demands on the external bus bandwidth. On average, this results in more pending misses, thus increasing the effective load miss latency and increasing the EP stalls caused by *memory latency* (see the rightmost graph of Fig. 4). On the other hand, the AP stalls due to integer load misses, which cannot be reduced by decoupling, as discussed in Section 3, are almost eliminated by multithreading (see *memory latency* in the leftmost graph of Fig. 4).

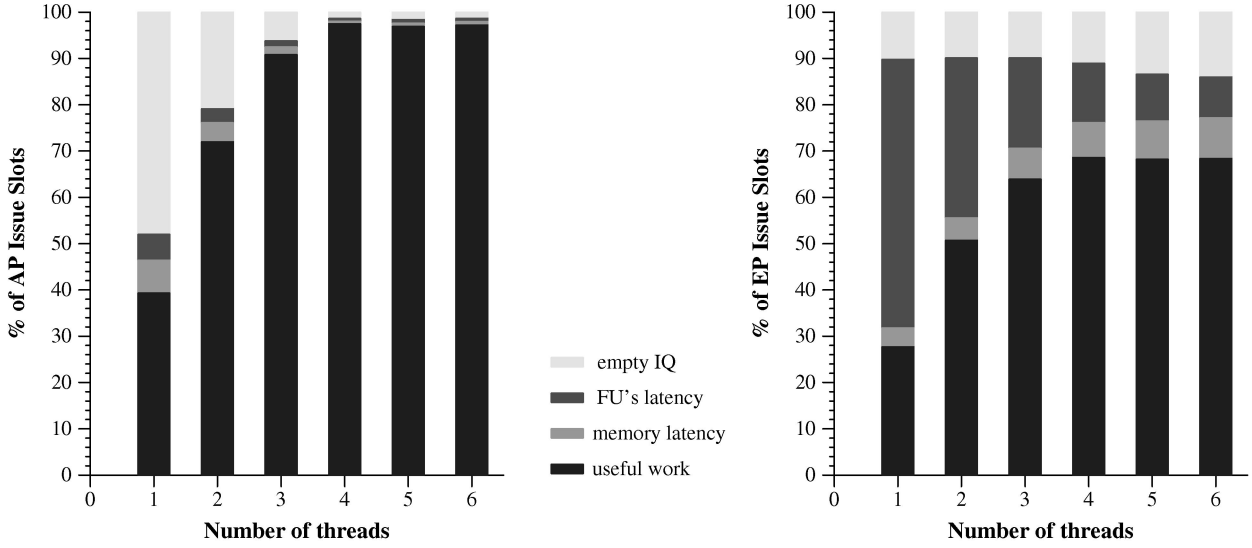

Fig. 4. AP (left) and EP (right) issue slots breakdown for the multithreaded decoupled architecture.

4.4 Latency Hiding Effectiveness

Multithreading and decoupling are two different approaches to tolerating memory latency. We have run some experiments, similar to those of Section 3, to reveal the separate contributions of decoupling and multithreading to the latency hiding effect. We have quantified the latency tolerance for two multithreaded architectures, both having from one to four contexts: a multithreaded decoupled processor and a degenerated version of it where the issue queues are disabled, i.e., similar to a pure in-order multithreaded architecture. Such nondecoupled architecture also has register renaming to support out-of-order completion of nonblocking misses and to provide precise exceptions. Notice that, therefore, it may still hide some memory latency by overlapping the execution of nonblocking misses with subsequent instructions, including other misses.

These two architectures have similar complexity, except for the number of physical registers required per thread. However, this number does not grow—in fact, it decreases—when the number of contexts increases since each has its own register file. Therefore, the register file access time is not expected to determine the processor cycle time. Other structures that have similar complexity in both architectures will more likely stay in the critical path. We have thus considered cycle time implications as a second order factor to compare them and performance is given in terms of IPC. A further complexity analysis, while important, is beyond the scope of this paper.

Fig. 5a shows the average perceived load miss latency when varying L2 latency from one to 256 cycles, for the eight configurations (combinations of one to four threads with/without decoupling). The definition of perceived latency given in Section 3 is slightly modified for a multithreaded architecture to express the same notion of memory latency tolerance: The latency perceived by a load is the number of cycles where an instruction that uses its value cannot issue and it causes an issue slot to be wasted (not filled by any other thread). That is, if a load use is at the head of a thread's issue queue, but all the issue slots in a cycle are successfully filled with instructions from other threads, this cycle does not add to the load perceived latency. Fig. 5b shows the corresponding relative performance loss (with respect to the 1-cycle L2 latency) of each of the eight configurations. Notice that performance loss compares the impact of memory latency on the IPC for each architecture, rather than their absolute performance.

Several conclusions can be drawn from these graphs. First, we can observe in Fig. 5a that the average perceived load miss latency is quite low when decoupling is enabled (less than six cycles for an L2 latency of 256 cycles), but it is much higher when decoupling is disabled, and it may only be hidden by overlapping instructions from other threads. Second, although it may seem rather surprising, multithreading does not significantly improve the average perceived miss latency: There are less than three cycles difference in the perceived latency for one and four threads in the nondecoupled processor and less than 1.5 cycles in the decoupled processor. Although having more threads increases the opportunity to fill some empty issue slots, there are very few additional cases when all of them are filled, which is the condition needed in our definition to consider that the memory latency is not perceived by a stalled load use instruction. Moreover, the little latency hiding provided by the additional threads is almost offset by the increase of the miss ratio (due to the larger combined working set), which produces longer bus contention delays.

Third, when the L2 memory latency is increased from one cycle to 32 cycles, it is shown in Fig. 5b that the decoupled multithreaded architecture experiences performance drops of less than 3.6 percent (less than 1.5 percent with four threads), while the performance degradation observed in all the nondecoupled configurations is greater than 23 percent. Even for a huge memory latency of 256 cycles, the performance loss of all the decoupled configurations is lower than 39 percent, while it is greater than 79 percent for the nondecoupled configurations. Fourth, multithreading provides some additional latency tolerance improvements, especially in the nondecoupled

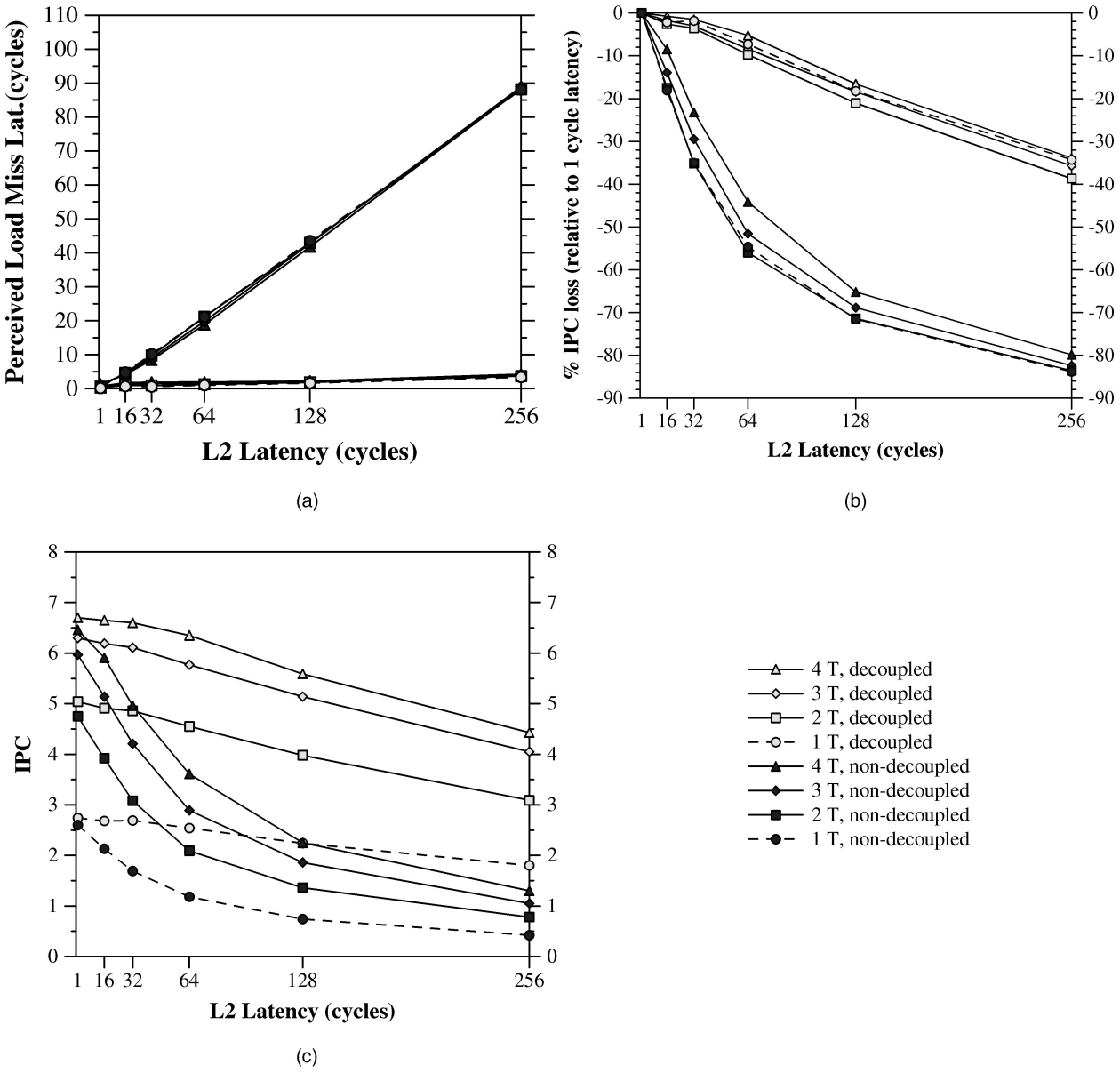

Fig. 5. (a) Average perceived load miss latency of individual threads. (b) Latency tolerance: Performance loss is relative to the 1-cycle L2 latency case. (c) Contribution of decoupling and multithreading to performance.

configurations, but it is much lower than the latency tolerance provided by decoupling.

Some other conclusions can be drawn from Fig. 5c, which shows the IPC for each configuration. While having more threads raises the performance curves, decoupling makes them flatter. In other words, while the main effect of multithreading is to provide more throughput by exploiting thread level parallelism, the major contribution to memory latency tolerance, which is related to the slope of the curves, comes from decoupling and this is precisely the specific role that decoupling plays in this hybrid architecture.

4.5 Hardware Context Reduction and the External Bus Bandwidth Bottleneck

Multithreading is a powerful mechanism that highly improves the processor throughput, but it has a cost: It needs a considerable amount of hardware resources. We have run some experiments that illustrate how decoupling reduces the hardware context requirements. We have measured the performance of several configurations having from one to eight contexts, both with a decoupled multithreaded architecture and a nondecoupled multithreaded architecture (see Fig. 6a). While the decoupled configuration achieves the maximum performance with just three or four threads, the nondecoupled configuration needs six threads to achieve similar IPC ratios.

One of the traditional claims of the multithreading approach is its ability to sustain a high processor throughput, even in systems with a high memory latency. Since hiding a longer latency may require a higher number of contexts and, as is well-known, this has a strong negative impact on the memory performance, the reduction in hardware context requirements obtained by decoupling may become a key factor when L2 memory latency is high.

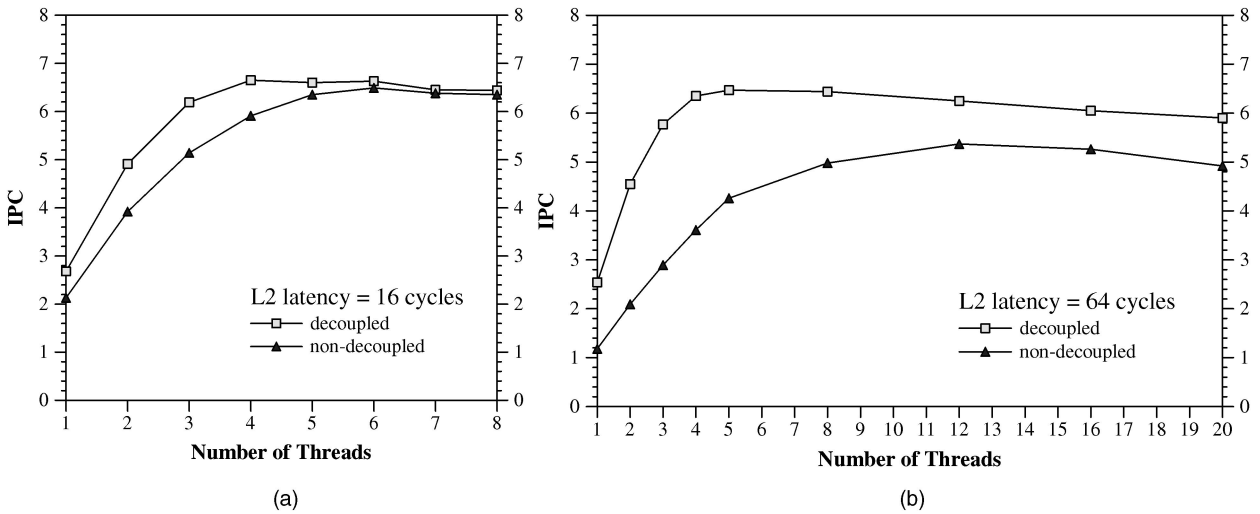

Fig. 6. (a) Decoupling reduces the number of hardware contexts. (b) Maximum performance without decoupling cannot be reached due to external bus saturation.

To illustrate this, we have run the previous experiment for an L2 memory latency of 64 cycles. As shown in Fig. 6b, while the decoupled architecture achieves the maximum performance with just four or five threads, the nondecoupled architecture cannot reach similar performance with any number of threads because it would need so many that they would saturate the external L2 bus: The average bus utilization is 89 percent with 12 threads and 98 percent for 16 threads. Moreover, notice that the decoupled architecture requires just three threads to achieve about the same performance as the nondecoupled architecture with 12 threads. Thus, decoupling significantly reduces the amount of parallelism required to reach a certain level of performance.

The previous result suggests that the external L2 bus bandwidth is a potential bottleneck in this kind of architecture. To further describe its impact, we have measured the performance and bus utilization of several configurations having from one to six hardware contexts, for three different external bus bandwidths of eight, 16, and 32 bytes/cycle. Results are shown in Fig. 7a and Fig. 7b. For an 8 bytes/cycle bandwidth, the bus becomes saturated when more than three threads are running and performance is degraded beyond this point.

To summarize, decoupling and multithreading complement each other to hide memory latency and increase ILP with reduced amounts of thread-level parallelism and low issue logic complexity.

5 Summary and Conclusions

In this paper, we have analyzed how access/execute decoupling improves the latency tolerance of simultaneous multithreading. A multithreaded decoupled architecture aims at taking advantage of the latency hiding effectiveness

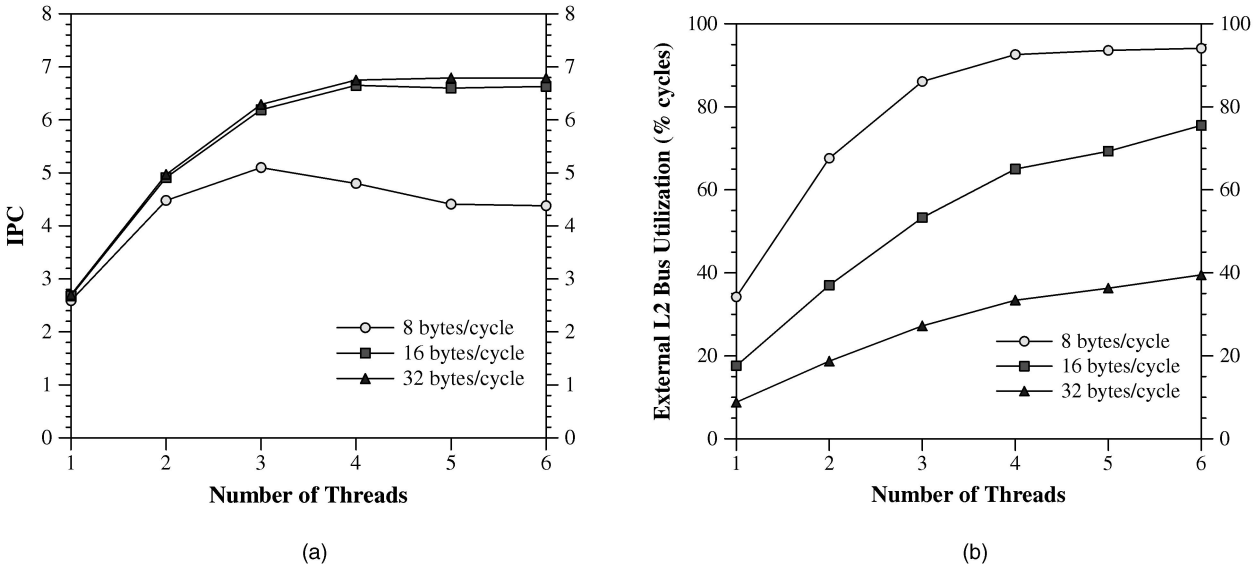

Fig. 7. (a) IPC for several bus bandwidths. (b) External L2 bus utilization for several bus bandwidths.

of decoupling and the potential of multithreading to exploit ILP. We have analyzed the most important factors that determine its performance and the synergistic effect of both paradigms. From this study, we have drawn the following conclusions:

First, multithreading alone has good tolerance for short latencies, like those of functional units, since these stalls are almost eliminated when five threads are running (for a 16 cycle L2 latency, five threads achieve 93 percent utilization of the functional units of the AP, which results in a throughput of 6.35 IPC). But, we also show that it suffers an important performance degradation caused by long memory latencies: When L2 latency is increased from one to 32 or more cycles, the IPC drop is higher than 23 percent in the best case (four threads).

Second, we have demonstrated that access/execute decoupling is a dynamic scheduling policy that performs quite well in the specific task of tolerating a long memory latency, either alone or in conjunction with multithreading: We have found that, when L2 latency is increased from one to 32 or more cycles, the IPC drop is always lower than 3.6 percent and it is quite independent of the number of threads. Furthermore, even for a huge L2 latency of 256 cycles and four threads, the average perceived latency of a load miss is less than six cycles.

Hence, we conclude that multithreading is quite effective to increase ILP, but rather limited to hide memory latency. Therefore, to tolerate long latencies, some sort of dynamic scheduling is needed and decoupling is an excellent alternative. Of course, out-of-order could do it as well, but its issue logic has a much higher complexity, which may have implications in the cycle time.

Third, in a decoupled multithreaded architecture, maximum performance is reached with very few threads: For a 16 cycle L2 latency, 6.19 IPC is achieved with just three threads and 6.65 IPC with four threads. On the other hand, more threads are needed to reach a similar level of performance in an in-order architecture: It achieves 6.35 IPC with five threads and 6.49 IPC with six threads. The number of simultaneously active threads supported by the architecture has a significant impact on the hardware chip area (e.g., number of registers and instruction queues) and complexity (e.g., the instruction fetch and issue logic) and, consequently, in clock cycle.

Reducing the number of threads reduces the number of cache conflicts and also prevents the saturation of the external bus bandwidth, which is usually one of the potential bottlenecks of a multithreaded architecture. We have shown that, in the assumed architecture, the bus bandwidth becomes a bottleneck when the miss latency is 64 cycles if decoupling is disabled and prevents the processor from achieving the maximum performance with any number of threads.

In summary, we can conclude that decoupling and multithreading techniques complement each other to exploit instruction level parallelism and to hide memory latency. This particular combination obtains its maximum performance with few threads, has a reduced issue logic complexity, and it is hardly performance degraded by a wide range of L2 latencies. All those features make it a promising alternative for future increases in clock speed and issue width.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their valuable comments. This work has been supported by the Ministry of Education of Spain under contract CYCIT TIC98-0511 and by the European Union through the ESPRIT program under the MHAOTEU (EP24942) project. The research conducted in this paper has been developed using the resources of the CEPBA.

REFERENCES

[1] A. Berrached, L.D. Coraor, and P.T. Hulina, "A Decoupled Access/Execute Architecture for Efficient Access of Structured Data," *Proc. 26th Hawai Int'l Conf. System Sciences,* vol. 1, pp. 438-447, Jan. 1993.

[2] P.L. Bird, A. Rawsthorne, and N.P. Topham, "The Effectiveness of Decoupling," *Proc. Int'l Conf. Supercomputing,* pp. 47-56, July 1993.

[3] R. Canal, J.-M. Parcerisa, and A. González, "A Cost-Effective Clustered Architecture," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques,* Oct. 1999.

[4] G.E. Daddis and H.C. Torng, "The Concurrent Execution of Multiple Execution Streams on Superscalar Processors," *Proc. Int'l Conf. Parallel Processing,* pp. 76-83, Aug. 1991.

[5] Digital Equipment Corp., *Alpha 21164 Microprocessor Hardware Reference Manual,* Or. Num. EC-QAEQB-TE, Maynard, Mass., Apr. 1995.

[6] K.I. Farkas, P. Chow, N.P. Jouppi, and Z. Vranesic, "The Multicluster Architecture: Reducing Cycle Time through Partitioning," *Proc. 30th. Ann. Symp. Microarchitecture,* Dec. 1997.

[7] J.R. Goodman, J.T. Hsieh, K. Liou, A.R. Pleszkun, P.B. Schechter, and H.C. Young, "PIPE: A VLSI Decoupled Architecture," *Proc. 12th Int'l Symp. Computer Architecture,* pp. 20-27, June 1985.

[8] L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report,* vol. 9, no. 2, pp. 9-15, Feb. 1995.

[9] L. Gwennap, "Digital 21264 Sets New Standard," *Microprocessor Report,* vol. 10, no. 14, Oct. 1996.

[10] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," *Proc. 19th Ann. Int'l Symp. Computer Architecture,* pp. 136-145, May 1992.

[11] P.Y.-T. Hsu, "Designing the TFP Microprocessor," *IEEE Micro,* vol. 14, no. 2, pp. 23-33, Apr. 1994.

[12] M. Johnson, *Superscalar Microprocessor Design.* Englewood Cliffs, N.J.: Prentice Hall, 1991.

[13] G.P. Jones and N.P. Topham, "A Limitation Study into Access Decoupling," *Proc. Third Euro- Par Conf.,* pp. 1102-1111, Aug. 1997.

[14] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. Int'l Symp. Computer Architecture,* pp. 364-373, 1990.

[15] G.A. Kemp and M. Franklin, "PEWs: A Decentralized Dynamic Scheduler for ILP Processing," *Proc. Int'l Conf. Parallel Processing,* vol. 1, pp. 239-246, 1996.

[16] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *Proc. Eighth Int'l Symp. Computer Architecture,* pp. 81-87, May 1981.

[17] A. Kumar, "The HP-PA8000 RISC CPU: A High Performance Out-of-Order Processor," *Proc. Hot Chips VIII,* pp. 9-20, Aug. 1996.

[18] L. Kurian, P.T. Hulina, and L.D. Coraor, "Memory Latency Effects in Decoupled Architectures," *IEEE Trans. Computers,* vol. 43, no. 10, pp. 1129-1139, Oct. 1994.

[19] D. Levitan, T. Thomas, and P. Tu, "The PowerPC 620™ Microprocessor: A High Performance Superscalar RISC Microprocessor," *Proc. COMPCON '95,* pp. 285-291, 1995.

[20] J.M. Parcerisa and A. González, "The Latency Hiding Effectiveness of Decoupled Access/Execute Processors," *Proc. 24th Euromicro Conf.,* pp. 293-300, Aug. 1998.

[21] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-Effective Superscalar Processors," *Proc. 24th Int'l Symp. Computer Architecture,* pp. 1-13, 1997.

[22] S. Palacharla and J.E. Smith, "Decoupling Integer Execution in Superscalar Processors," *Proc. 28th Ann. Symp. Microarchitecture,* pp. 285-290, Nov. 1995.
[23] A.R. Pleszkun and E.S. Davidson, "Structured Memory Access Architecture," *Proc. 1983 Int'l Conf. Parallel Processing,* pp. 461-471, Aug. 1983.
[24] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J.E. Smith, "Trace Processors," *Proc. 30th Ann. Symp. Microarchitecture,* pp. 138-148, Dec. 1997.
[25] S.S. Sastry, S. Palacharla, and J.E. Smith, "Exploiting Idle Floating-Point Resources for Integer Execution," *Proc. Int'l Conf. Programming Language Design and Implementation,* 1998.
[26] J.E. Smith, "A Study of Branch Prediction Strategies," *Proc. Eighth Ann. Int'l Symp. Computer Architecture,* pp. 135-148, May 1981.
[27] J.E. Smith, "Decoupled Access/Execute Computer Architectures," *ACM Trans. Computer Systems,* vol. 2, no. 4, pp. 289-308, Nov. 1984.
[28] J.E. Smith, G.E. Dermer, B.D. Vanderwarn, S.D. Klinger, C.M. Rozewski, D.L. Fowler, K.R. Scidmore, and J.P. Laudon, "The ZS-1 Central Processor," *Proc. Second Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 199-204, Oct. 1987.
[29] J.E. Smith and A.R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," *Proc. 12th Int'l Symp. Computer Architecture,* pp. 36-44, June 1985.
[30] J.E. Smith, S. Weiss, and N.Y. Pang, "A Simulation Study of Decoupled Architecture Computers," *IEEE Trans. Computers,* vol. 35, no. 8, pp. 692-702, Aug. 1986.
[31] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. 22nd Int'l Symp. Computer Architecture,* pp. 414-425, 1995.
[32] G.S. Sohi and M. Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 53-62, Apr. 1991.
[33] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *Proc. SIGPLAN Conf. Programming Language Design and Implementation,* pp. 196-205, June 1994.
[34] Standard Performance Evaluation Corp., *SPEC Newsletter,* Fairfax, Va., Sept. 1995.
[35] R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Research and Development,* vol. 11, no. 1, pp. 25-33, Jan. 1967.
[36] N.P. Topham, A. Rawsthorne, C.E. McLean, M.J.R.G. Mewissen, and P. Bird, "Compiling and Optimizing for Decoupled Architectures," *Proc. Supercomputing '95,* Dec. 1995.
[37] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* pp. 191-202, 1996.
[38] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 392-403, 1995.
[39] G. Tyson, M. Farrens, and A.R. Pleszkun, "MISC: A Multiple Instruction Stream Computer," *Proc. 25th Ann. Symp. Microarchitecture,* pp. 193-196, Dec. 1992.
[40] W.A. Wulf, "An Evaluation of the WM Architecture," *Proc. 19th Int'l Symp. Computer Architecture,* pp. 382-390, May 1992.
[41] K.C. Yaeger, "The Mips R10000 Superscalar Microprocessor," *IEEE Micro,* vol. 16, no. 2, pp. 28-41, Apr. 1996.
[42] Y. Zhang and G.B. Adams III, "Performance Modelling and Code Partitioning for the DS Architecture," *Proc. 25th Int'l Symp. Computer Architecture,* pp. 293-304, June 1998.

Joan-Manuel Parcerisa received the degree in computer science in 1993 from the Universitat Politècnica de Catalunya at Barcelona, Spain. Since 1994, he has been a full-time teaching assistant and a PhD student in the Computer Architecture Department at the Universitat Politècnica de Catalunya. His current research topics include cluster architectures for dynamically scheduled processors, multithreading, access/execute decoupling, and value prediction.

Antonio González received the degree in computer science in 1986 and the PhD degree in computer science in 1989, both from the Universitat Politècnica de Catalunya at Barcelona, Spain. He has occupied different faculty positions in the Computer Architecture Department at the Universitat Politècnica de Catalunya since 1986, with tenure since 1990. His research interests center on computer architecture, compilers, and parallel processing, with a special emphasis on processor microarchitecture, memory hierarchy, and instruction scheduling. Dr. González is a member of the IEEE Computer Society.