The Synergy of Multithreading and Access/Execute Decoupling

Joan-Manuel Parcerisa and Antonio González
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya - Barcelona (Spain)
Email: {jmanel,antonio}@ac.upc.es

Abstract

This work presents and evaluates a novel processor microarchitecture which combines two paradigms: access/execute decoupling and simultaneous multithreading. We investigate how both techniques complement each other: while decoupling features an excellent memory latency hiding efficiency, multithreading supplies the in-order issue stage with enough ILP to hide the functional unit latencies. Its partitioned layout, together with its in-order issue policy makes it potentially less complex, in terms of critical path delays, than a centralized out-of-order design, to support future growths in issue-width and clock speed.

The simulations show that by adding decoupling to a multithreaded architecture, its miss latency tolerance is sharply increased and in addition, it needs fewer threads to achieve maximum throughput, especially for a large miss latency. Fewer threads result in a hardware complexity reduction and lower demands on the memory system, which becomes a critical resource for large miss latencies, since bandwidth may become a bottleneck.

1. Introduction

Dynamic scheduling is a latency tolerance technique that can hide much latency of memory and functional units. However, as memory latencies and issue widths continue to grow in the future, dynamically scheduled processors will need larger instruction windows. As reported in [4], the hardware complexity of some components in the critical path that determines the clock cycle time may prevent centralized architectures to scale up to faster clock frequencies. Therefore, several architectures have been proposed recently, either in-order or out-of-order, which address this problem by partitioning critical components of the architecture and/or providing less complex scheduling mechanisms [6, 1, 3, 4, 9]. This work focuses on one of these partitioning strategies: the access/execute paradigm [5], which was first proposed for early scalar architectures to provide them with dual issue and a limited form of dynamic scheduling that is especially oriented to tolerate memory latency.

On the other hand, simultaneous multithreading has been shown to be an effective technique to boost ILP [8]. In this paper, we analyze its potential when implemented on a decoupled processor.

We show in this study that the combination of decoupling and mulithreading takes advantage of their best features: while decoupling is a simple but effective technique for hiding high memory latencies with a reduced issue complexity, multithreading provides enough parallelism to hide functional unit latencies and keep them busy. In addition, multithreading also helps to hide memory latency when a program decouples badly. However, since decoupling hides most memory latency, few threads are needed to achieve a near-peak issue rate. This is an important result, since having few threads reduces the memory pressure, which is a major bottleneck in multithreading architectures, and reduces the hardware cost and complexity.

The rest of this paper is organized as follows. Section 2 quantifies the latency hiding effectiveness of decoupling. Section 3 describes and evaluates a multithreaded decoupled architecture. Section 4 summarizes the main conclusions.

2. Latency Hiding Effectiveness of Decoupling

Since the interest of decoupling is closely related to its ability to hide memory latencies without resorting to other more complex issue mechanisms, we have first quantified such ability for a wide range of L2 cache latencies, from 1 to 256 cycles. We have evaluated a 4-way issue, single-threaded, decoupled architecture with 4 general purpose functional units and a 2-port L1 data cache. The latencies and other architectural parameters are those of Figure 2.

The baseline single-threaded decoupled architecture consists of two superscalar decoupled processing units: the Address Processing unit (AP) and the Execute Processing unit (EP). Precise exceptions are supported by means of a reorder buffer, a graduation mechanism, and a register renaming map table. The Instruction Queue in the EP allows the AP to execute ahead of the EP, providing the necessary slippage between them to hide the memory latency, and the Store Address Queue allows loads to bypass stores. For

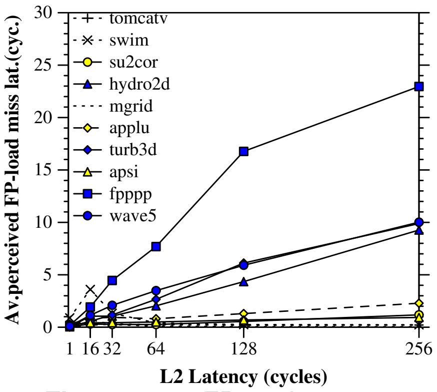


Figure 1-a: FP loads.

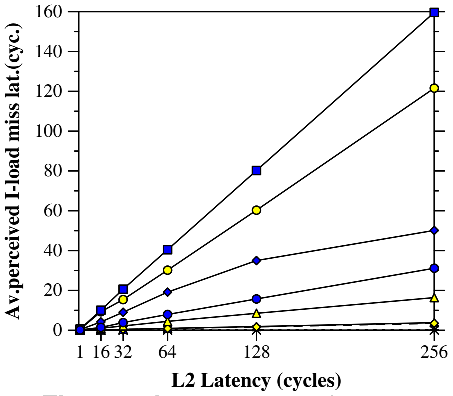


Figure 1-b: Integer loads.

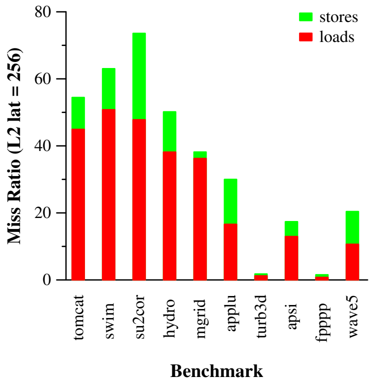


Figure 1-c: Miss Ratios

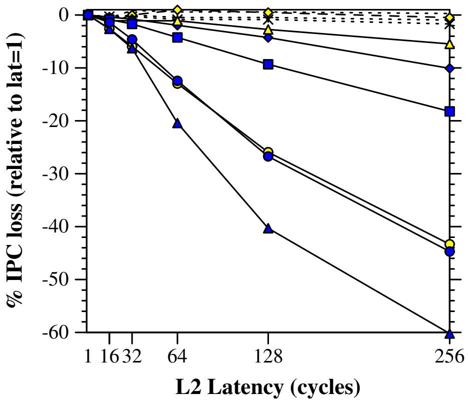


Figure 1-d: Impact of latency on performance.

these experiments, the sizes of all the architectural queues and physical register files are scaled up proportionally to the L2 latency.

The instruction stream, which is based on the DEC Alpha ISA, is dynamically split: instructions are dispatched to either the AP or the EP following a simple steering mechanism based on their data type (int or fp), except for memory instructions, which are all sent to the AP. Although this rather simplistic scheme mostly benefits to 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. Techniques to decouple integer codes can be found elsewhere [5].

Since one of the main arguments for the decoupled approach is the reduced issue logic complexity, each thread issues instructions in-order within each processing unit. It may be argued that in-order processors have a limited potential to exploit ILP. However, current compiling techniques can extract much ILP and thus, the compiler can pass this information to the hardware instead of using runtime schemes (this is the approach that emerging EPIC architectures take [2]).

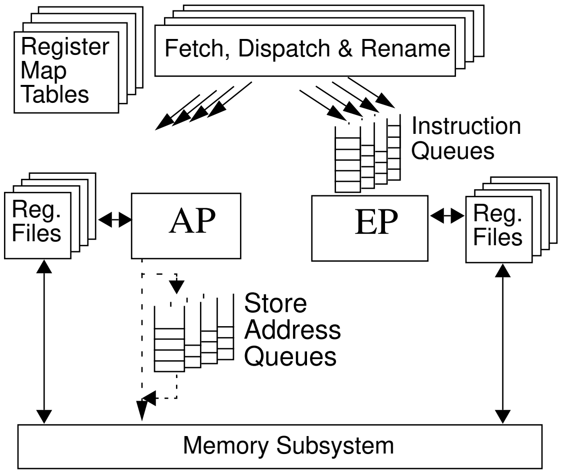
The experiments consisted of a set of trace driven cycle-by-cycle simulations of the SPEC FP95 benchmark suite. The traces were obtained by instrumenting the DEC Alpha binaries with the ATOM tool, and running them with their largest available input data sets. However, due to the detail of the simulations, we only run 100M instructions of each benchmark, after skipping the initial start-up phase.

In addition to the IPC, we have also measured separately the average "perceived" latency of integer and FP load misses, i.e., the average number of stall cycles of instructions that use data from a previous uncompleted load. Since we are interested in the particular benefit of decoupling, independently of the cache miss ratio, this average does not include load hits.

The perceived latency of FP load misses measures the EP stalls caused by misses, and reveals the "decoupled behavior" of a program, i.e., the amount of slippage of the AP with respect to the EP. As shown in Figure 1-a, except for *fpppp*, more than 96% of the FP load miss latency is always hidden. The perceived latency of integer load misses measures the AP stalls caused by misses, and it depends on the ability of the compiler to schedule integer loads ahead of other dependent instructions. As shown in Figure 1-b, *fpppp*, *su2cor*, *turb3d* and *wave5* are the programs that experience the largest integer load miss stalls.

Regarding the impact of the L2 latency on performance (see Figure 1-d), although programs such as *fpppp* or *turb3d* perceive much load miss latency, they are hardly performance degraded due to their extremely low miss ratios (see Figure 1-c). The most performance degraded programs are those with both high perceived miss latency and significant miss ratios: *hydro2d*, *wave5* and *su2cor*.

To summarize, performance is little affected by the L2 latency when either it can be hidden efficiently (*tomcatv*, *swim*, *mgrid*, *applu* and *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*). 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.



AP functional units	4 (latency = 1 cycle)
EP functional units	4 (latency = 4 cycles)
Control speculation at AP	4 unresolved branches
L1 on-chip I-cache	2 ports, infinite
L1 on-chip data cache	4 ports, lockup-free (16 MSHRs),
64 KB, dir.map.,	32 byte/line, write back, 1 cycle hit
L2 off-chip cache	infinite, multibanked, 16 cycle hit
L1-L2 interface	128-bit wide bus, 16 bytes/cycle.
	Per thread:
AP physical registers	64
EP physical registers	96
Instruction Queue	48 entries
Store Address Queue	32 entries
BHT	2K entries x 2 bit

Figure 2: Scheme and main parameters of the multithreaded decoupled processor

3. A Multithreaded Decoupled Architecture

A multithreaded decoupled architecture (Figure 2) supports multiple hardware contexts, each executing in a decoupled mode. The fetch and dispatch stages - including branch prediction and register map tables - and the register files and queues are replicated for each context. The issue logic, functional units and caches are shared by all the threads. Up to 8 instructions from different threads can be issued per cycle to 8 general purpose functional units. All the threads are allowed to compete for each of the 8 issue slots each cycle, and priorities among them are round-robin (similar to the *full simultaneous issue* scheme reported in [8]). Each cycle, only two threads have access to the I-cache, and each of them can fetch up to 8 consecutive instructions (or up to the first taken branch). The two chosen threads are those with less instructions pending to be dispatched (similar to the RR-2.8 with I-COUNT schemes, reported in [7]).

Early experiments revealed that in a single-threaded architecture most of the wasted issue slots are caused by true data dependences between EP register operands, due to the

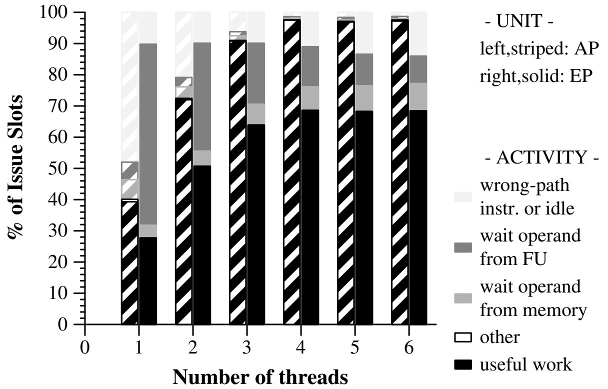


Figure 3: issue slots breakdown

restricted ability of the in-order issue model to exploit ILP. Therefore, as far as decoupling hides memory latency and multithreading supplies enough amounts of parallelism to remove the remaining stalls, we expect important synergistic effects between these two techniques in a hybrid architecture.

For the experiments in this section, the multithreaded decoupled architecture parameters are those in Figure 2. The simulator is fed with independent threads. Each thread consists of a sequence of traces from all SpecFP95 programs, in a different order for each thread.

3.1. Sources of Wasted Issue Slots

The first column pair in Figure 3 represents the case of a single thread, showing that the major bottleneck is caused by the EP functional units latency, as discussed above. When two more contexts are added, multithreading drastically reduces these stalls in both units, and produces a 2.31 speed-up (from 2.68 IPC to 6.19 IPC). Since with 3 threads the AP functional units are nearly saturated (90.7%), negligible speed-ups are obtained by adding more contexts (6.65 IPC is achieved with 4 threads).

Note that although the AP almost achieves its maximum throughput, the EP functional units are not saturated due to the load imbalance between the AP and the EP. Therefore, the effective peak performance is reduced by 15%, from 8 to 6.8 IPC. This problem could be addressed with a different issue width 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 higher demands on the external bus bandwidth. On average, there are more pending misses, which increases the effective load miss latency, and the EP stalls caused by *waiting operands from memory* (Figure 3). However, in the AP, since integer loads are much less frequent than fp loads, the ad-

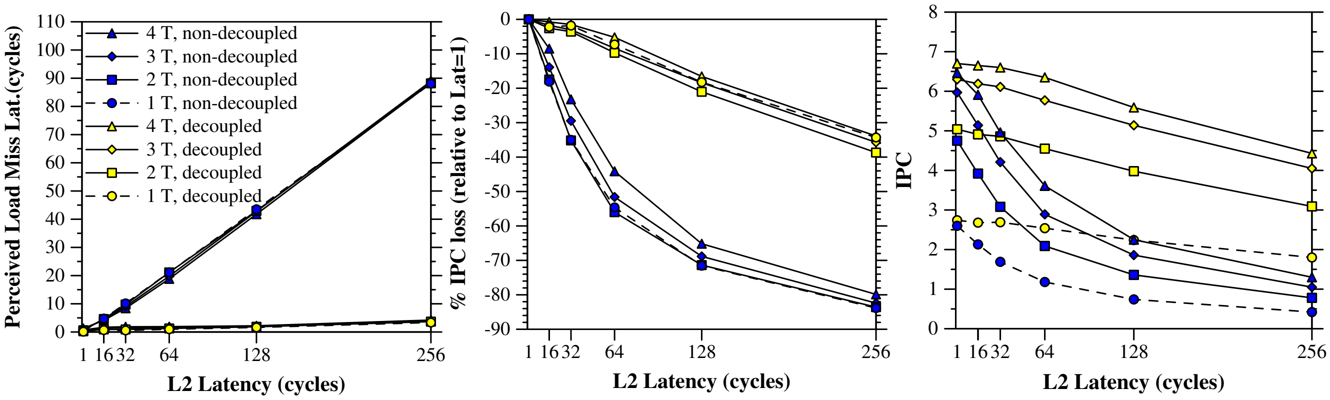


Figure 4: (a) Perceived latency. (b) Relative IPC loss. (c) Effects of decoupling and multithreading on IPC.

ditional parallelism provided by multithreading eliminates almost all of this kind of stalls.

3.2. Latency Hiding Effectiveness

Multithreading and decoupling are two different approaches to tolerate high memory latencies. We have run some experiments to quantify the latency tolerance of a multithreaded decoupled processor for 1 to 4 threads. In addition, some other experiments are also carried out to reveal the contribution of each mechanism to the latency hiding effect. They consist of a set of identical runs on a degenerated version of our multithreaded architecture where the instruction queues are disabled (i.e. a non-decoupled multithreaded architecture).

Figure 4-a shows the average perceived load miss latency, when varying L2 latency from 1 to 256 cycles for the 8 configurations (combinations of 1 to 4 threads with/without decoupling). This metric expresses the average number of cycles that an instruction that uses a load value cannot issue although there is a free issue slot. It can be seen that decoupling hides almost all memory latency, even when it is very high, whereas multithreading helps very little.

Figure 4-b shows the corresponding relative performance loss (with respect to the 1-cycle L2 latency) of each of the 8 configurations. Notice that this metric compares the tolerance of these architectures to memory latency, rather than their absolute performance. Several conclusions can be drawn from these graphs. First, it is shown that when the L2 memory latency is increased from 1 to 32 cycles, the decoupled multithreaded architecture experiences performance drops of less than 4%, while the performance degradation observed in all non-decoupled configurations is greater than 23%. Even for a huge memory latency of 256 cycles, the performance loss of the decoupled configurations is lower than 39% while it is greater than 79% for the non-decoupled configurations.

Second, multithreading provides some additional latency tolerance improvement, especially in the non-decoupled configurations, but it is much lower than that provided by decoupling.

Some other conclusions can be drawn from Figure 4-c. While multithreading raises the performance curves, decoupling makes them flatter. In other words, while the main effect of multithreading is to provide more 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.

3.3. Reduction in Hardware Contexts

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 required number of hardware contexts.

We have measured the performance of several configurations having from 1 to 7 contexts, for a decoupled multithreaded architecture and a non-decoupled multithreaded architecture (Figure 5, solid lines). While the decoupled configuration achieves the maximum performance with just 3 or 4 threads, the non-decoupled configuration needs 6 threads to achieve similar IPC rates.

Multithreading is usually claimed to be able 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 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. To illustrate this fact, we have run a similar experiment for 1 to 16 contexts and a L2 memory latency of 64 cycles. As shown in Figure 5 (dotted lines), while the decoupled architecture achieves the maximum performance with just 4 or 5 threads, the non-decoupled architecture cannot reach a similar performance with any number of threads, because it would need so many that they would saturate the external

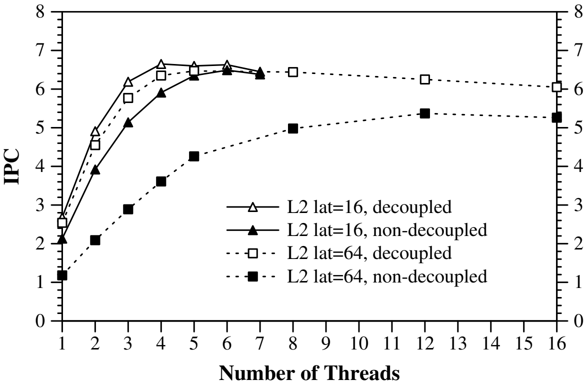


Figure 5: Decoupling reduces hardware contexts and avoids external bus saturation

L2 bus: the average bus utilization is 89% for 12 threads, and 98% for 16 threads. Moreover, note that the decoupled architecture requires just 3 threads to achieve about the same performance as the non-decoupled architecture with 12 threads. Thus, decoupling significantly reduces the amount of thread-level parallelism required to reach a certain level of performance.

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

4. Summary and Conclusions

In this paper we have analyzed the synergy of multithreading and access/execute decoupling. A multithreaded decoupled architecture takes advantage of the latency hiding effectiveness 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.

A multithreaded decoupled architecture hides efficiently the memory latency: the average perceived load miss latency is less than 5 cycles in the worst case (with 4 threads and a L2 latency of 256 cycles). We have also found that, for L2 latencies lower than 32 cycles, their impact on the performance is quite low: less than 4% IPC loss, relative to the 1-cycle latency scenario, and it is quite independent of the number of threads. On the other hand, this impact is greater than a 23% IPC loss if decoupling is disabled. This latter fact points out that decoupling is the main contributor to memory latency tolerance.

The architecture reaches maximum performance with very few threads, significantly less than in a non-decoupled architecture. The number of simultaneously active threads supported by the architecture has a significant impact on the hardware chip area and complexity, which may compromise the clock cycle.

Reducing the number of threads also reduces the cache conflicts and the required memory bandwidth, which is usually one of the potential bottlenecks of a multithreaded architecture. We have shown that if decoupling is disabled, the external L2 bus bandwidth becomes a bottleneck when the miss latency is 64 cycles, which prevents the processor from achieving the maximum performance for any number of threads.

In summary, we can conclude that decoupling and multithreading techniques complement each other to exploit parallelism and to hide memory latency. A multithreaded decoupled processor 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 of these features make it a promising alternative for future increases in clock speed and issue width.

Acknowledgements

This work has been supported by grant CYCIT TIC98-0511 and the ESPRIT Project MHAOTEU (EP24942).

References

[1] K.I.Farkas, P.Chow, N.P.Jouppi, Z.Vranesic. The Multicluster Architecture: Reducing Cycle Time Through Partitioning. In *Proc of the Micro-30*, Dec. 1997

[2] L. Gwennap. Intel, HP Make EPIC Disclosure. *Microprocessor Report*, 11(14), Oct. 1997.

[3] G.A.Kemp, M.Franklin. PEWs: A Decentralized Dynamic Scheduler for ILP Processing. In *Proc. of the ICPP.* 1996, v.1, pp 239-246.

[4] S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-Effective Superscalar Processors. In *Proc of the 24th. ISCA*, 1997, pp 1-13.

[5] S.S.Sastry, S.Palacharla, J.E.Smith. Exploiting Idle Floating-Point Resources For Integer Execution. In *Proc. of the PLDI.* Montreal, 1998.

[6] J.E. Smith. Decoupled Access/Execute Computer Architectures. *ACM Trans. on Computer Systems*, 2 (4), Nov. 1984, pp 289-308.

[7] G.S.Sohi, S.E.Breach, and T.N.Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd ISCA*. 1995, pp 414-425.

[8] D.M. Tullsen, et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23rd. ISCA*. 1996, pp 191-202.

[9] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. of the 22nd. ISCA*. 1995, pp 392-403.

[10] Y.Zhang, G.B.Adams III. Performance Modelling and Code Partitioning for the DS Architecture. In *Proc. of the 25th. ISCA*, Jun. 1998, pp 293-304.