The Latency Hiding Effectiveness of Decoupled Access/Execute Processors

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

Abstract

Several studies have demonstrated that out-of-order execution processors may not be the most adequate organization for wide issue processors due to the increasing penalties that wire delays will cause in the issue logic. The main target of out-of-order execution is to hide functional unit latencies and memory latency. However, the former can be quite effectively handled at compile time and this observation is one of the main arguments for the emerging EPIC architectures. In this paper, we demonstrate that a decoupled access/execute organization is very effective at hiding memory latency, even when it is very long. This paper presents a thorough evaluation of such processor organization.

First, a generic decoupled access/execute architecture is defined and evaluated. Then the benefits of a lockup-free cache, control speculation and a store-load bypass mechanism under such architecture are evaluated. Our analysis indicates that memory latency can be almost completely hidden by such techniques.

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 growing their capabilities to exploit parallelism by means of greater issue widths and deeper pipelines, which makes even higher the negative impact of memory latencies on the performance. To alleviate this problem, most current processors devote a high fraction of their transistors to on-chip caches [20], in order to reduce the average memory access time. Several prefetching techniques have been also developed, both hardware and software [3].

Some processors, commonly known as out-of-order issue [33, 17, 15, 7, 8], include dynamic scheduling techniques, most of them based on the Tomasulo algorithm [29] or variations of it, that allow them to tolerate both memory and functional unit latency, by overlapping it with useful computations of other independent instructions. To implement out-of-order issue, the processor is capable of filling issue slots with independent instructions by looking forward in the instruction stream, into a limited instruction window. This is a general mechanism that aggressively extracts the instruction parallelism available in the instruction window.

A decoupled access/execute architecture [22, 23, 6, 32, 31, 19, 2, 11] includes some limited kind of dynamic scheduling which is especially oriented to tolerate memory latency. It splits - statically or dynamically - the instruction stream into two. One stream is composed of all those instructions involved in the fetch of data from memory, and it executes asynchronously respect the other one, which is formed with the instructions that process these data. Both streams are executed on independent processing units (called AP and EP respectively, in this paper) which communicate mutually and with the memory system through queues. The AP is expected to execute in advance of the EP and to prefetch the data from memory so that the EP can consume the data without any delay. This anticipation or *slippage* may involve multiple conditional branches, so it actually performs a kind of dynamic loop unrolling. However, the amount of slippage between the AP and the EP highly depends on the program ILP, because data and control dependencies can force both units to synchronize - the so called Loss of Decoupling events [2, 30] - producing a serious performance degradation.

1089-6503/98 $10.00

As memory latencies continue to grow in the future, out-of-order processors will need longer issue 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 [18], the networks involved in the issue stage, and also - although to a less extent - those of the renaming stage, 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 has a component that increases quadratically with the window length. And, although linearly, it also depends strongly on the issue width. Moreover, higher density technologies only accelerate the increase in these latencies. Their analysis suggest that out-of-order architectures could find in the future a serious boundary on their clock speeds.

A decoupled processor provides an alternative to this problem. It has a reduced issue and data bypass logic complexity, not only because of its in-order issue policy, but also because these tasks are subdivided into two processing units, with independent register files and pipelines. Therefore, it 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. It may be argued that in-order processors have a limited potential to exploit ILP. However, current compiling techniques can extract much of the parallelism in a program and thus, providing a means for the compiler to communicate parallelism to the hardware is the approach that emerging EPIC (Explicitly Parallel Instruction Computing) architectures will take [9]. This paper presents an exhaustive evaluation of decoupled access/execute processors and in particular, of its ability to hide memory latency.

We first analyze a generic decoupled architecture with dynamic instruction split [23, 11, 32] and a data cache. Other studies on decoupled machines have been carried out before [1, 22, 6, 25, 23, 32, 31, 16, 5, 10, 13], but they did not analyze techniques like store-load forwarding, control speculation or lockup-free caches. In this paper we evaluate specifically the impact of these techniques when applied to a decoupled processor, and quantify the memory latency sensitivity of this architecture.

The rest of this paper is organized as follows. Section 2 describes the basic decoupled architecture. Section 3 analyzes the performance of this architecture and identifies its major strengths and weaknesses. Finally, we summarize the main conclusions of this work in Section 4.

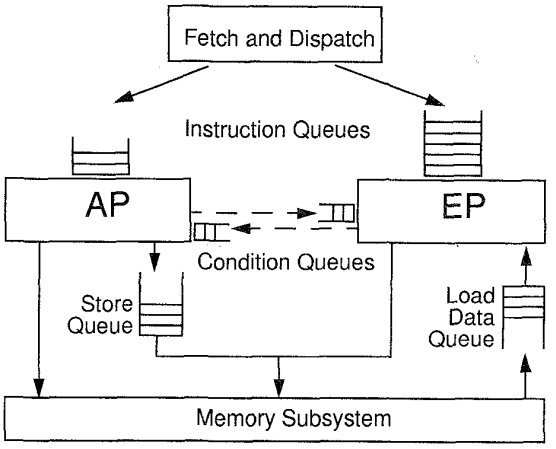

Figure 1: Scheme of a decoupled processor

2. A Decoupled access/execute processor

The processor microarchitecture evaluated in this paper (Figure 1) consists of two superscalar decoupled processing units: the Address Processing unit (AP) and the Execute Processing unit (EP). The decoupled processor executes a single instruction stream, based on the DEC-alpha ISA [4], by splitting it dynamically and dispatching the instructions to either the AP or the EP. There are two separate register files, one in the AP with 32 integer registers, the other in the EP with 32 FP registers. Both units share a common fetch and dispatch stages, while they have separate issue, execute and write-back stage pipelines. Next, there is a brief description of each stage:

The fetch stage reads up to 4 consecutive instructions per cycle from an ideal I-cache (less than 4 if there is a taken branch among them). It is also provided with a conditional branch prediction scheme based on a 2K entry Branch History Table, with a 2-bit saturating counter per entry [21].

The dispatch stage decodes up to 4 instructions per cycle and sends them to either the AP or the EP instruction queue, depending on whether they are integer or floating point instructions, respectively, in a similar way to the ZS-1 [23] or the MIPS R8000 [11]. These queues have 4 and 64 entries, respectively. As an exception, Floads and Fstores are sent to both the AP and the EP, because while they move data to/from EP registers, their effective address calculation involves AP registers. Whether a conditional branch is dispatched to the AP or to the EP depends on the kind of comparison. The instructions that follow the branch are fetched based on the prediction outcome, and some of them will be sent to the same processing unit than the

branch, while others will be sent to the other processing unit. Each processing unit must be able to identify exactly which are the instructions that follow this branch, and to squash them in case of misprediction. It is easy to identify those instructions sent to the same processing unit than the branch, but there is nothing that distinguishes those sent to the other processing unit. The more simple solution would be to send the branch to both processing units, like in the ZS-1[23] among others, one of them being just a single token to indicate that the instructions following it are speculative. However, to avoid such code expansion, since this token has no operands, it has been replaced by a single bit added to the first next instruction dispatched to that processing unit. When the branch is resolved, the outcome is sent to the other processing unit through a Condition Queue.

Since the AP usually executes ahead of the EP, and such slippage is a key factor on performance, in most of our experiments we assume that the AP can issue and execute speculatively instructions beyond a certain number of EP branches. However, this requires that the AP has the appropriate hardware to recover from mispredicted branches. Just for comparison, we have also implemented a non-speculative model where AP stalls and waits for EP branches to be resolved before issuing the instructions that follow them. Speculative execution has not been implemented in the EP because the required hardware is quite complex, and the EP does not naturally tend to execute ahead of the AP.

Each processing unit is provided with 2 general purpose and fully pipelined functional units. Each processing unit can read and issue up to 2 instructions per cycle. For the sake of simplicity, the latencies of all operations in the EP are assumed to be 4 cycles, while those in the AP are only 1 cycle, except accesses to the data cache, which need one additional cycle in case of hit, and some more cycles in case of miss.

The AP portion of an Fload calculates the effective address and sends it to the memory system. The data is finally delivered to the Load Data Queue (LDQ), from where it will be popped out, and written to a register, by the corresponding dummy Fload in the EP. Similarly, the AP calculates Fstore effective addresses and holds them in the Store Queue (SQ), until the data is delivered by the corresponding dummy Fstore in the EP. Both queues have 32 entries. Loads are allowed to execute ahead of uncompleted stores, after being disambiguated against all the addresses held in the SQ. In most of our experiments there exists also a forwarding mechanism that allows dependent loads to be put aside in a pending queue until they receive the data directly from the store, thus avoiding to stall the AP.

The primary data cache is on-chip, 2-ported [26], 8 KB[1], direct-mapped, with a 32 byte block length, and it implements a write-back policy to minimize off-chip bus traffic. We assume that primary cache misses always hit in a large ideal off-chip L2 cache, and they have a 16 cycle latency plus any penalty due to bus contention. On most of our experiments, we also assume that the L1 data cache is lockup-free [14], and it is modelled like that of the Alpha 21164 [4], but augmenting to 16 the number of (primary) misses to different lines because the miss latency is also longer. It can also merge up to 4 (secondary) misses per pending line. The L1-L2 interface consists of a 128-bit wide data bus which completes one transaction each 2 CPU cycles (i.e. every cache line keeps the bus busy during 4 cycles) by overlapping several transactions.

To maintain precise exceptions we assume that there exists an elementary reorder buffer, a graduation mechanism and some exception recovery hardware [12, 24] for the AP. The recovery hardware for the EP is greatly simplified by just preventing the EP from issuing ahead of uncompleted AP instructions (including conditional branches). As far as the AP executes ahead of the EP, this constraint saves lots of hardware complexity at the expense of very little penalties.

3. Performance evaluation

In this section we identify through experimental evaluation the major sources of wasted cycles in a typical decoupled architecture, and the effectiveness of several techniques commonly used to alleviate these problems: lockup-free caches, store-load forwarding and control speculation. We also evaluate the latency hiding effectiveness of this architecture. The discussion highlights the major weaknesses and strengths of the decoupled approach.

3.1. Simulation methodology and workload

Experiments are carried out with a trace driven simulator. The binary code is obtained by compiling the SPEC FP95 benchmark suite [28], for a DEC AlphaStation 600 5/266, with the DEC compiler applying full optimizations. The trace is generated by running this code previously instrumented with the ATOM tool [27]. The simulator models, cycle-by-cycle, the architecture described in the previous section, and runs the SPEC FP95 benchmarks, fed with their largest available input data sets.

1. . The relatively small L1 cache has been chosen to stress the latency hiding requirements of the processor.

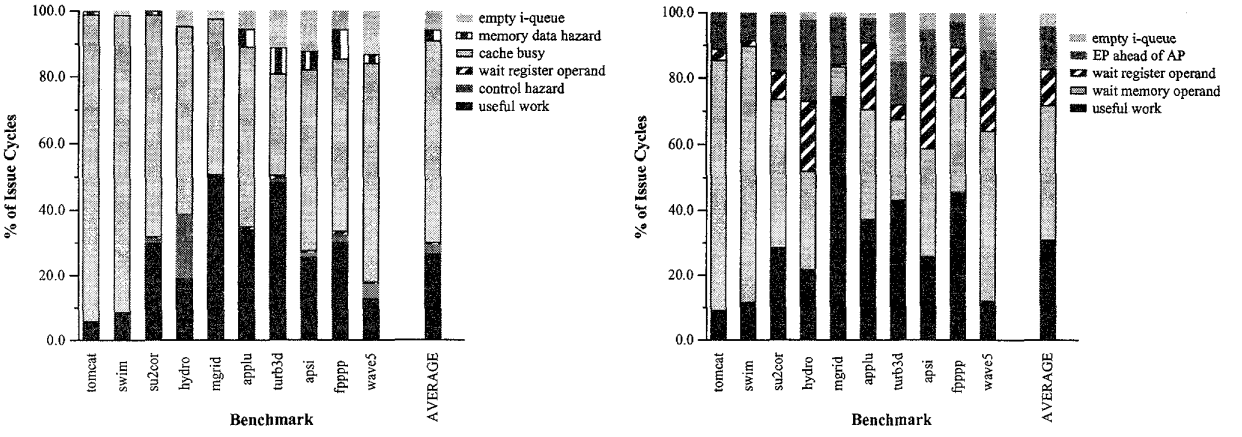

Figure 2: AP (left) and EP (right) issue cycle breakdowns, for a basic decoupled architecture with non-blocking, forwarding and AP speculation disabled.

Because of the detail of the simulations, they are very slow. Therefore, we simulate only a portion of 100 million instructions of each benchmark, after skipping an initial start-up phase. To determine the appropriate initial discarded offset we compared the instruction-type frequencies of such a fragment starting at different points, with the full run frequencies. We found that this phase has not the same length for all the benchmarks: about 5000 M instructions for 101.tomcatv and 103.su2cor; 1000 M for 104.hydro2d and 146.wave5; and just 100 M for the rest of the benchmarks.

3.2. Sources of wasted cycles

We have first measured the throughput of the Issue stage in terms of the percentage of committed instructions over the total issue slot count (i.e. % of cycles where it is really doing useful work) for a basic architecture having disabled the lockup-free cache, the store-load forwarding and the AP speculation. The results are shown in Figure 2. Figure 3 shows the results of a similar analysis (only the average of the ten benchmarks) when these three techniques are enabled separately, and also when they are combined in several ways.

The wasted throughput is also characterized, by recording the cause for each empty issue slot. The label *control hazard* means that the AP cannot issue an instruction because it depends on an unresolved EP conditional branch. Register interlocks are labelled as *wait register operand*. The label *cache busy* means that a memory instruction cannot be executed because the L1 cache is either being accessed by the L2 cache (for a line replacement), or it is processing a blocking miss. Notice that this is the main cause for AP stalls in this architecture.

The label *memory data hazard* means that a Load stalls the AP because it references the same address than a previous pending Fstore. The label *wait memory operand* means that a Load instruction in the EP cannot read its data because this is not yet delivered to the Load Data Queue. This is the main source of wasted cycles in the EP. The label *EP ahead of AP* means that the EP is stalled in order not to overtake the AP, due to the restriction imposed to simplify precise exceptions. Finally, the label *empty i-queue* includes the slots wasted by uncommitted instructions (those squashed in case of a branch misprediction) and the slots lost because the instruction queue is empty. This latter cause is observed in programs that show a bad load balance between both processing units. Since more than one of these causes may overlap for a given instruction in a clock cycle, the stall is accounted to the first of these causes, in the order given in the legend of the plot (top-down order). We discuss below the main conclusions drawn from these figures.

3.2.1. Effectiveness of a lockup-free cache (load miss stalls). As shown in Figure 2 (labels *cache busy* and *wait memory operand*), when a lockup-free cache is not used, the AP is stalled by load misses and the EP is waiting for memory data, for most of the time. Miss latency increases the AP cycle count far above the EP cycle count. The AP execution time becomes the bounding limit of the global performance, and decoupling can hardly hide memory latencies. The nature of these stalls is a structural hazard, and they can be reduced by providing the processor with a lockup-free cache. As shown in Figure 3 (column labelled *nonbl*), with this cache, this kind of stalls are almost eliminated. Of course, this uncovers other overlapped causes, but the overall improvement in the performance achieves an impressive 89.7% increase (from 0.88 IPC to 1.67

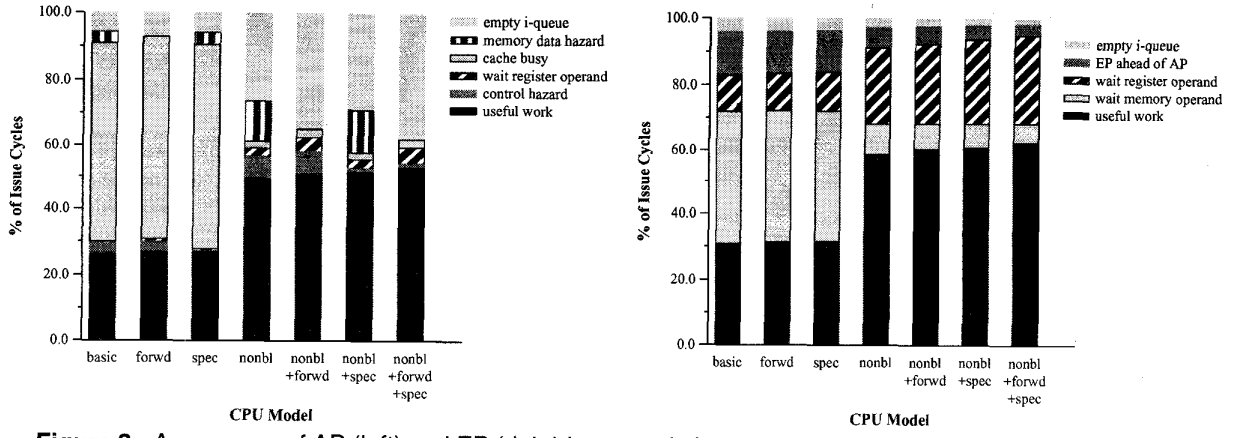

Figure 3: A summary of AP (left) and EP (right) issue cycle breakdowns, with the "average" columns of all the models

IPC).

3.2.2. Effectiveness of store-load forwarding (memory data hazard stalls). Another source of wasted cycles are memory data hazards (Figure 2, left) detected during memory disambiguation. With forwarding disabled, when a load matches the effective address of a pending Fstore, the AP pipeline is blocked until the store is issued to the cache. The EP is indirectly affected by these stalls: the slippage between the AP and the EP gets reduced - we call this event a loss of decoupling, or LOD [2, 30] - and any other subsequent Fload is exposed to be penalized by the memory latency in case of a cache miss.

Our experiments (Figure 3 left, column *forwd*), show that store-load forwarding removes completely these stalls, but since most of the benchmarks are compute bound (except for *turb3d*), the overall speed-up will be, at most, that of the EP, and it depends on how much it was penalized by these LODs. That is, it depends on how frequent the stalls were, whether they made the amount of slippage drop below the threshold of the memory latency, and whether a subsequent Fload missed in the cache before the AP recovered the decoupling. Figure 3 (columns *forwd*) shows that, despite the significant amount of saved stalls achieved by forwarding in the AP (labelled *memory data hazards*), the EP lost slots (labelled *wait memory operand*) are very little reduced. The average performance improvement is just a 2.27% increase (from 0.88 IPC to 0.90 IPC).

3.2.3. Effectiveness of AP speculative execution (control hazard stalls). Another source of wasted cycles are control hazards originated by FP conditional branches. When speculative execution is disabled, the AP instructions that follow the branch must wait until the condition is delivered by the EP to the Condition Queue. This kind of LODs are removed by enabling the AP to execute speculatively instructions beyond one ore more branches. In case one of the branches is found to be mispredicted, the hardware must be able to recover the state previous to the branch [12, 24]. The cost of this hardware depends on the particular implementation and the speculation depth, which is the number of unresolved EP branches beyond which the instruction issue mechanism stalls. We have assumed a speculation depth of 4, which is the same as the MIPS R10000 [33] and the PowerPC 620 [17].

Our experiments show that, although control hazard stalls are almost completely removed (Figure 3, left), the average IPC increases only by 2.2%. This is due to the low average frequency of these branches (0.36% of all the instructions). However, this technique provides significant improvements to particular programs where they are more frequent. These is the case of *hydro2d* (2.35% of the completed instructions), which experiments a 20% increase of the IPC. It can be seen in Figure 2 (left) that this program experiences a significant penalty due to control hazards. It can be also noticed that when combining speculation with a lockup-free cache, the benefits of this technique are slightly higher (3.5% IPC increase) because the extra slippage provided to the AP by speculation is not lost by miss stalls, so that the latency perceived by Floads in the EP is reduced.

To summarize, we can conclude that store-load forwarding has a minor influence on performance. AP control speculation has also little impact on performance, but it is slightly higher if a lockup-free cache is present. In contrast, a lockup-free cache produces by itself such a high improvement that it is essential to a decoupled processor.

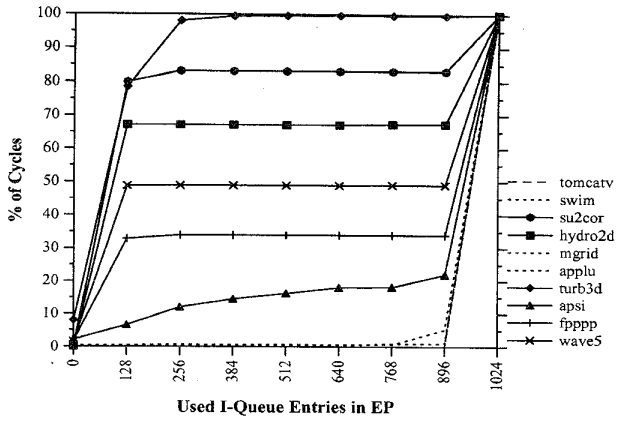

Figure 4-a: Utilization of the EP instruction queue entries, as an indicator of the decoupling behaviour.

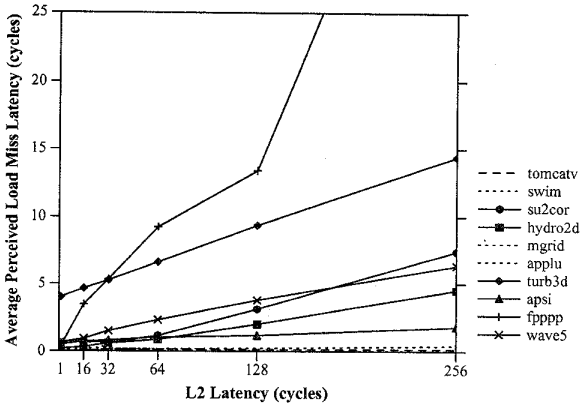

Figure 4-b: Perceived miss latency. The value for fpppp with a 256 L2 latency is 49.

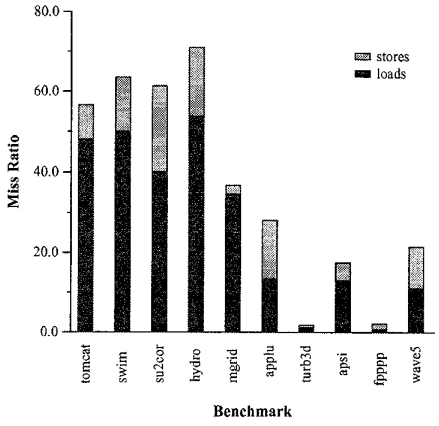

Figure 4-c: Miss Ratios of Loads and Stores, when L2 latency is 256 cycles.

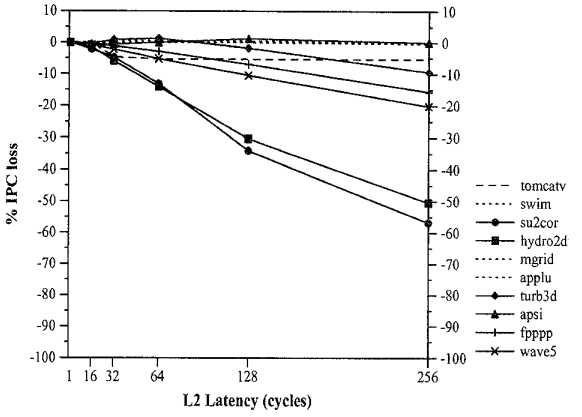

Figure 4-d: Impact of latency on performance (loss relative to the 1-cycle L2 latency case).

3.3. Latency hiding effectiveness

The interest of a decoupled architecture is closely related to its ability to hide long memory latencies. The latency hiding potential of a decoupled processor depends strongly on the decoupling behavior of the programs being tested. For some programs, the scheduling ability of the compiler to remove LOD events, which force the AP and the EP to synchronize, is also a key factor. However, the compiler we have used (Digital f77) is not especially tailored to a decoupled processor. Therefore, to validate our conclusions, we are interested in having an assessment of the latency hiding effectiveness of our basic architecture without any specific compiler support.

We have run the 10 benchmarks with the external L2 memory latency varying from 1 to 256 cycles. The simulations assume that the L1 cache is 64 KB and direct mapped, and the length of the processor architectural queues and the number of pending misses supported by the lockup-free cache are scaled up proportionally to the L2 latency. We have measured the miss ratio, the execution time, the utilization of the EP instruction queue, and the average "perceived" latency of load misses. Since we are interested in the particular benefit of decoupling, independently of the cache miss ratio, this average does not include load hits.

Figure 4-a shows the cumulative distribution of the utilization of the EP instruction queue entries. The number of used entries in the EP is closely related to the amount of slippage between the AP and EP. Therefore, programs that use few entries during many cycles (*turb3d, su2cor, hydro2d, wave5, and fpppp*) are said to "decouple badly".

Figure 4-b shows the average perceived latency of load misses, which quantifies the non-hidden latency of load misses. It shows that *tomcatv, swim, mgrid, applu* and *apsi* are almost not affected by the L2 latency, in spite of the

quite high load miss ratios of some of them (see *Figure 4-c*) because they decouple quite well, hiding efficiently the miss latencies. Such high miss ratios are produced when memory latency is so large because, after a pending miss (*primary*), subsequent loads to the same line are more likely to produce new misses (*secondary*). These new misses do not necessarily increase the number of requests to the L2 cache, if the implementation can merge them in a single request. We can also observe that the perceived latency of *turb3d* has a small positive offset in relation to others, which is caused by bus contention delays. When the L2 latency is 256 cycles, more than 97.2% of it is hidden for all the programs excepting *fpppp* and *turb3d*.

Figure 4-d shows the loss of performance, relative to the 1-cycle latency case, when L2 memory latency ranges from 1 to 256 cycles. The impact of the memory latency on the performance depends on both the perceived load miss latency (which is the effective stall caused by a miss, and it is closely related to the decoupling behavior of the program, as previously shown) and also on the miss ratio (which is related to the frequency of the stalls). For example, although *fpppp* and *turb3d* show the highest perceived latencies as a consequence of a bad decoupling behavior, they are little performance degraded because of their extremely low load miss ratios (see Figure 4-c). On the other hand, *su2cor* and *hydro2d* have the highest impact of latency on their performance because they have a bad decoupling behavior together with high miss ratios. To summarize, performance is little affected when programs show either a good decoupling behavior (*tomcatv*, *swim*, *mgrid*, *applu* and *apsi*), or a low miss ratio (*fpppp* and *turb3d*), but it is seriously degraded if they lack both features (*su2cor* and *hydro2d*).

4. Conclusions

In the this paper we have performed a detailed analysis of the main factors that influence the performance of a decoupled processor, identifying its major strengths and weaknesses:

- We have analyzed the effectiveness of a lockup-free cache, and we have found an average 1.89 speed-up over a processor with a blocking cache. A lockup-free cache reduces drastically the AP stall cycles caused by the latency of load misses, thus allowing the AP to run far ahead of the EP.
- We have also quantified the effectiveness of control speculation in the AP. It only produces significant improvements on programs with many FP conditional branches, like *hydro2d* (1.2 speed-up).
- We have also evaluated a Store-Load forwarding mechanism, and we have found that it has little impact on the performance, probably because when forwarding is not enabled, this hazard only reduces partially the decoupling, but it does not eliminate it completely.
- We have quantified the latency hiding potential of a decoupled processor: when the L2 latency is as large as 256 cycles, decoupling still hides more than 97.2% of it, for 8 of the benchmarks (and it hides 94.3% and 80.2% on the other two). The impact of the memory latency on the performance depends on two factors: the latency effectively perceived by the program (which is closely related to its decoupling behaviour), and its miss ratio. With a 64 KB direct-mapped cache, when L2 latency is 256 cycles, the performance loss with respect to the 1-cycle latency case is less than 20% on 8 of the benchmarks. We found that high degradations are only produced when both hit ratio and decoupling are low, which is the case of the other 2 benchmarks (50.3% and 56.7% of IPC reduction).
- The main cause that prevents to achieve the peak performance are true data dependencies between EP register operands. This penalty can be reduced with appropriate compiling techniques like those that will support future EPIC architectures.

5. References

[1] A. Berrached, L.D. Coraor, and P.T. Hulina. A Decoupled Access/Execute Architecture for Efficient Access of Structured Data. In *Proc. of the 26th. Hawai Int. Conf. on System Sciences*. IEEE, Los Alamitos, CA, USA, Jan. 1993. Vol. 1, pp 438-447.

[2] P.L. Bird, A. Rawsthorne, N.P. Topham. The Effectiveness of Decoupling. In *Proc. of the Int. Conf. on Supercomputing*. Tokyo, Japan, July 1993, pp 47-56.

[3] Chen, T-F. and J-L. Baer. A performance study of software and hardware data prefetching schemes. In *Proc. of the 21st. Int. Symp. on Comp. Architecture*. 1994, pp 223-232.

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

[5] M.K.Farrens, Pius Ng, and P.Nico. A Comparison of Superscalar and Decoupled Access/Execute Architectures. In *Proc. of the Micro-26*. Nov. 1993.

[6] J.R. Goodman, J.T. Hsieh, K. Liou, A.R. Pleszkun, P.B. Schechter, and H.C. Young. PIPE: A VLSI Decoupled Architecture. In *Proc of the 12th. Int. Symp. on Comp. Architecture*. Boston, MA, June 1985, pp 20-27.

[7] L. Gwennap. Intel's P6 Uses Decoupled Superscalar Design. *Microprocessor Report*, 9 (2), Feb. 1995, pp 9-15.

[8] L. Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, 10 (14), Oct. 1996.

[9] L. Gwennap. Intel, HP Make EPIC Disclosure. *Microprocessor Report*, 11(14), Oct. 1997, pp. 5-9

[10] T.J. Harris, and N.P. Topham. The Use of Caching in Decoupled Multiprocessors with Shared Memory. In *Proc. of the Scalable Shared Memory Workshop, at the Int. Parallel Processing Symposium*. Cancun, Mexico, 1994.

[11] P.Y-T. Hsu. Designing the TFP Microprocessor. *IEEE Micro*, 14(2), Apr. 1994, pp 23-33.

[12] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[13] G.P. Jones, and N.P. Topham. A Limitation Study into Access Decoupling. In *Proc. of the 3rd. Euro-Par Conference*. Aug. 1997, pp 1102-1111.

[14] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proc. of the 8th. Int. Symp. on Comp. Architecture*. May 1981, pp 81-87.

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

[16] L. Kurian, P.T. Hulina, and L.D. Coraor. Memory Latency Effects in Decoupled Architectures. *IEEE Trans. on Computers*, 43(10), Oct. 1994, pp 1129-1139.

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

[18] S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-Effective Superscalar Processors. In *Proc of the 24th. Int. Symp. on Comp. Architecture*, 1997, pp 1-13.

[19] A.R. Pleszkun, and E.S. Davidson. Structured Memory Access Architecture. In Proc. *of the 1983 Int. Conf. on Parallel Processing*. Bellaire Mich., Aug. 1983. pp 461-471.

[20] A.J. Smith. Cache Memories. *Computing Surveys*, 14(3), Sept. 1982.

[21] J.E. Smith. A Study of Branch Prediction Strategies. In *Proc. of the 8th Ann. Int. Symp. on Comp. Architecture.* May 1981, pp 135-148.

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

[23] 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. In *Proc. of the 2nd. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, C.A., October 1987, pp 199-204.

[24] J.E. Smith, A.R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. In *Proc. of the 12th. Int. Symp. on Computer Architecture*, June 1985, 36-44.

[25] J.E. Smith, S. Weiss, and N.Y. Pang. A Simulation Study of Decoupled Architecture Computers. *IEEE Transactions on Computers*, C-35(8), Aug. 1986, pp 692-702.

[26] G.S. Sohi, and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proc. of the 4th. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. Apr. 1991, pp 53-62.

[27] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation,* June 1994, pp 196-205.

[28] Standard Performance Evaluation Corporation. *SPEC Newsletter.* Fairfax, Virginia, Sept. 1995.

[29] R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11 (1), Jan. 1967, pp 25-33.

[30] N.P.Topham, A. Rawsthorne, C.E. McLean, M.J.R.G. Mewissen and P. Bird. Compiling and Optimizing for Decoupled Architectures. In *Proc. of the Supercomputing '95*, San Diego, Dec. 1995.

[31] G. Tyson, M. Farrens, and A.R. Pleszkun. MISC: A Multiple Instruction Stream Computer. In *Proc of the 25th. Ann. Symp. on Microarchitecture*, Portland, Oregon, Dec. 1992, pp 193-196.

[32] Wm.A. Wulf. An Evaluation of the WM Architecture. In *Proc. of the 19th. Int. Symp. Comput. Architecture*, Gold Coast, Australia, May 1992, pp 382-390.

[33] K.C. Yeager. The Mips R10000 Superscalar Microprocessor. *IEEE Micro*, April 1996, 16 (2) pp 28-41.