Hiding Cache Miss Penalty Using Priority-based Execution for Embedded Processors

Sanghyun Park, †Aviral Shrivastava, and Yunheung Paek
Seoul National University, Korea †Arizona State University, USA
shpark@optimizer.snu.ac.kr †Aviral.Shrivastava@asu.edu ypaek@snu.ac.kr

***Abstract*—The contribution of memory latency to execution time continues to increase, and latency hiding mechanisms become ever more important for efficient processor design. While high-end processors can use elaborate techniques like multiple issue, out-of-order execution, speculative execution, value prediction etc. to tolerate high memory latencies, they are often not viable solutions for embedded processors, due to significant area, power and chip complexity overheads. This paper proposes a hardware-software cooperative approach, called priority-based execution to hide cache miss penalty for embedded processors. The compiler classifies the instructions into low-priority and high-priority instructions. The processor executes the high-priority instructions, but delays the execution of low priority instructions. They are executed on a cache miss to hide the cache miss penalty. We empirically evaluate our proposal on the Intel XScale compiler and microarchitecture. Experimental results on benchmarks from Multimedia, MediaBench, MiBench, and SPEC2000 demonstrate an average 17% performance improvements, hiding 75% cache miss penalty.**

I. Introduction

Embedded processors today are facing increasingly larger memory latencies. With the growing disparity between processors and memory speeds, the operations that cause cache misses out to main memory take hundreds of processor cycles to complete execution [29]. As memory access operations result in approximately 30-40% of the total of instructions in applications, reducing the average memory access time is crucial for any type of architectures (see Figure 1). While high-end processors can use elaborate techniques like multiple issue, out-of-order execution, speculative execution, value prediction etc., to tolerate the high memory latencies, it is often not a viable solution for the embedded processors due to the significant overheads of these techniques in terms of area, power and chip complexity [11], [14], [17]. Consequently, the most existing embedded processors are single-issue, non-speculative processors [1], [23], [30]. In particular, all the implementations of ARM, which is the most popular embedded processor, are single-issue and non-speculative processors. Therefore embedded processors require alternative mechanisms to hide the memory latency with minimal power and area costs.

The fundamental problem with the single-issue processors is that the instructions which have long latency stall the pipeline, and do not allow the incoming instructions to be executed, even if they are independent and ready to be executed otherwise. Hardware solutions, e.g., out-of-order and speculation mechanisms perform the complex analysis to discover the independent operations in the schedule and execute them. The hardware required for the analysis and the execution-replay support often consumes too much power and has complex logic, which is in turn not adequate for the embedded processors.

In this paper, we intend to place some of the analysis complexity in the compiler's custody. The compiler will classify the instructions into the low priority instructions and the high priority instructions. The low priority instructions are those, whose results are not needed immediately to continue the pipeline execution. Normally, the processor will execute only the high priority instructions, and will postpone the execution of the low priority instructions. The register operands of the low priority instructions are renamed and queued in a instruction buffer. The low priority instructions will be executed on a cache miss, effectively hiding the cache miss latency. We call this hardware-software cooperative approach to instruction reordering as *priority based execution*. Only a simple register renaming mechanism and a buffer to keep the state of the low priority instructions are required to suspend the execution of the low priority instructions.

The Intel XScale processor is the most advanced implementation of ARM, but is a single-issue processor. We implement our proposal on our compiler and cycle accurate simulator of the Intel XScale microarchitecture. Our experiments on a set of benchmarks collected from MultiMedia [5], MediaBench [12], MiBench [21], DSPStone [32] and Spec2K [13] demonstrate an average 17% performance improvement and 75% hiding of cache miss penalty over the default execution model.

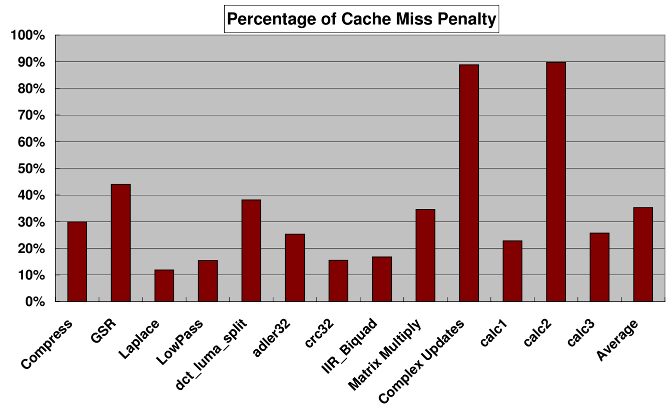


Fig. 1. On average 35% of total execution time in Intel XScale is spent on cache misses

The rest of the paper is organized as follows: Section II discusses other latency hiding mechanisms. We describe our approach of priority based execution in detail in Section III. After describing our experimental setup in Section IV, we perform several experiments to demonstrate the effectiveness of our approach in Section V. Finally, Section VI summarizes the paper.

II. Related Work

There has been extensive research to tolerate memory latency. Prefetching [4], [6], [7] is the most popular scheme to hide the memory latency. In software prefetching [6], a compiler inserts a *prefetch* instruction before the actual load instruction is issued. Hardware prefetching [18], [19] predicts cache misses using the past access patterns at runtime. These techniques highly rely on the memory access pattern and are ineffective for many applications which have irregular memory access patterns. Wrong prediction will cause the pollution of the cache, degrading performance significantly. In multithreaded processor, thread-based prefetching [2], [31] uses another thread to prefetch data into shared cache using a technique called pre-execution [2], [7]. However they need one or more thread contexts to prefetch data for a primary thread, implying the under-utilization of the multithreaded processor.

Out-of-order processors can inherently tolerate the memory latency to some extent using the reorder buffer. However, the memory latency it can hide is limited due to the size of the reorder buffer. To alleviate this problem, many techniques have been proposed to virtually enlarge the instruction window by complex checkpointing mechanisms [25] and *Window Instruction Buffer* (WIB) [20]. Also *Kilo-instruction processors* [8] have been proposed to allow thousands of in-flight instructions in the processor. *Run-ahead execution* [9], [26] was proposed to utilize cache miss duration as an alternative to large instruction windows. Pajuelo et al. [27] propose a scheme to speculatively execute independent instructions in the cache miss duration even if there is no available entry in the reorder buffer.

In [14], the authors present the cost and performance trade-offs of out-of-order execution. They discuss that even though in-order execution suffers from a 46% performance gap compared with out-of-order execution, out-of-order mechanism is too costly in terms of complexity and design cycles for embedded systems. Acknowledging that out-of-order issue mechanisms are very expensive, in [17] the authors propose a technique to reduce hardware cost of out-of-order execution, in which they present *access decoupled machine* to reduce instruction window logic complexity and memory latency. However their work is completely a hardware approach which is still costly, while we use compiler supports to hide memory latency. [11] propose *Delayed Issue* technique as a cost-effective alternative for out-of-order execution. They utilize the compiler to specify explicit delays for data dependent instructions, and use the delays to place instructions into per-functional unit delay queues so that each functional unit can execute those instructions in order. Although they can reduce the cost of out-of-order execution, they should maintain multiple instruction queues for each functional unit which means high area overheads. And more importantly, their work is not specifically aimed to hide the data cache miss penalty.

III. Our Approach

In this section, we use an example to describe the intuition and the details of priority based execution model. We consider the assembly instruction code in Figure 2(a), which shows the innermost loop in the compress benchmark from the MultiMedia benchmark suite.

A. Priority of Instructions

Intuitively the high priority instructions are those which are required to continue the operations of the instruction fetch and data transfers. The high priority instructions are defined as:

1) The branch instruction is high priority
2) All loads are high priority
3) All instructions that generate the source operands for a high priority instruction are high priority

All branch instructions are the high priority instructions because if we do not execute the branch instructions, we cannot know what to fetch next. Similarly all load instructions are the high priority instructions to keep the bus between the processor and data memory busy. In addition, all instructions that generate the source operands for the high priority instructions also need to be executed with high priority. All the other instructions in a loop are the low priority instructions.

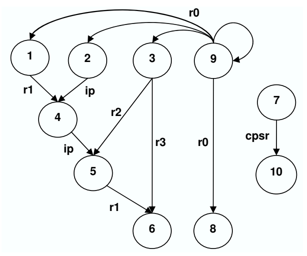
The priority of the instructions are determined as following steps.

1) Mark all load and branch instructions of a loop.
2) Use UD chains to find instructions that define the operands of already marked instructions, and mark them.
3) Find conditional instructions of which already marked instructions are control-dependent on, and mark them.
4) Recursively continue Step 2 and 3 until no more instructions can be marked.
5) Marked instructions are high priority, and unmarked instructions are low priority.

Our low priority selection algorithm operates on the Data Flow Graph (DFG) of the loop. The loop is represented as a DFG $G = (N, E)$, where each $n \in N$ is an operation in the loop body. There exists a directed edge $e = (n_i, n_j) \in E$ if operation n_i of any loop iteration is data dependent on operation n_j of any loop iteration. Thus there will be a directed edge between two operations even if they have a loop carried

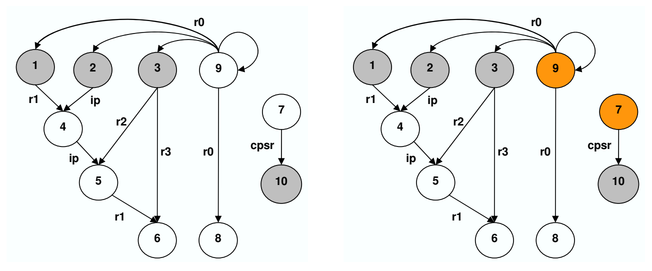
```
01: .L19: ldr     r1, [r0, #-404]
02:       ldr     ip, [r0, #-400]
03:       ldmda   r0, r2, r3
04:       add     ip, ip, r1, asl #1
05:       add     r1, ip, r2
06:       rsb     r2, r1, r3
07:       subs    lr, lr, #1
08:       str     r2, [r0]
09:       add     r0, r0, #r
10:       bpl     .L19
```

(a) Assembly code



(b) Data flow graph

Fig. 2. Innermost loop of the Compress benchmark



(a) Using clauses 1 and 2 of the definition (b) Repeatedly using clause 3 of the definition

Fig. 3. Finding high priority instructions

dependency. Figure 2(b) shows the data dependence graph of the innermost loop of the compress benchmark shown in Figure 2(a). The annotation on each edge is the dependent operand.

Using the definition and Step 1, all the loads are considered to be the high priority instructions. Therefore, in Figure 2(b), instructions 1, 2 and 3 are high priority. The branch instruction at the end of the loop is important to continue to fetch new instructions. Therefore instruction 10 is high priority (Figure 3(a)). Then using Step 2 and 3, more high priority instructions are detected. All instructions which compute the source operands of branch are also high priority. Instruction 7 subtracts register *lr* and sets the zero flag in the condition code, which is used by the branch instruction (10) to decide the execution flow. In addition, instruction 9 computes the register *r0*, which will be used by the high priority instructions, 1, 2 and 3, therefore instruction 9 is also a high priority instruction (Figure 3(b)). Finally, among the rest of the instructions, the memory operands of the store instructions are checked. If the store instruction feeds a high priority load, it is not marked as a low priority instruction. In this example, 4, 5, 6 and 8 are marked as low priority instructions.

B. Scope of Our Analysis

Our technique considers the instructions only in the loops. When the processor finishes the execution of a loop, it flushes the instruction window by executing the pending low priority instructions. Since a cache miss rarely occurs outside the loop body, we execute all the pending low priority instructions rather than let them wait in the reorder buffer (ROB) until the next cache miss. For our transformation, there should be no other branch instructions inside the loop. If there are branch constructs like “if-then-else”, they have to be converted into conditional execution, or predicated instructions [24]. Predication is a mechanism of converting control dependencies into data dependencies, and it allows our technique to consider more candidates of the low priority instruction in a loop.

For the memory operations, our compiler disambiguates the memory dependency statically using techniques proposed in [10]. If the static memory disambiguation approaches cannot disambiguate memory operations in a loop, we do not consider those operations as the low priority instructions. It is worth mentioning that our priority based execution technique is orthogonal to the memory disambiguation; our technique can hide memory latency even without memory disambiguation, but it can perform better if we can disambiguate memory dependencies more.

C. ISA Enhancements

To support flushing operation and priority classification, we need Instruction Set Architecture (ISA) modifications. We annotate 1-bit priority information for every instruction to make the processor perform a priority based execution, and add a *flushLowPriority* instruction to execute the pending low priority instructions at the end of the loop execution. These modifications can raise the issue of binary compatibility, however for the embedded applications, this issue is less critical than the applications for general-purpose computing and can be considered as acceptable.

D. Execution Model

There are two modes of the execution in our priority based execution model; the high priority execution mode and the low priority execution mode. In the high priority execution mode, only the high priority instructions are executed. The operands of the low priority instructions are renamed and the instructions are stored for a while in a ROB. When a data cache miss occurs, the blocked high priority instructions are flushed from the processor pipeline. Then the processor switches to the low priority execution mode and starts executing the low priority instructions. During the low priority execution, if the cache miss is resolved, the processor immediately resumes the high priority execution mode. If there are no more low-priority instructions to execute even before the cache miss is serviced, the processor switches back to high priority execution mode and stalls until the miss is resolved. Note that the low priority instructions cannot stall the processor pipeline due to cache miss since no load instruction is low priority. This scheme results in performance improvements due to two main reasons. Firstly, since we skip the low priority instructions during the loop iteration, the number of effective instructions in a loop is reduced and therefore the loop executes faster. Secondly, the low priority instructions are executed on a cache miss and the cache miss latency is therefore utilized for the useful work.

E. Architecture Model

Architecturally, priority based execution is similar to out-of-order execution with compiler support. The processor microarchitecture should provide support for the full register renaming so as not to violate the dependencies between the instructions.

Figure 4 shows our architecture model. It consists of the ROB, the rename manager, the rename table, and the priority selector. The ROB contains the decoded instructions and an extra 1-bit indicator P, in which ‘0’ indicates the low priority and ‘1’ means the high priority. The rename table keeps mapping information of the real register into the rename register. The instructions in the ROB are renamed dynamically by the rename manager with information in the rename table. The priority selector consists of 3 comparators that compare the source registers with the register which misses the cache, and it decides the execution mode of the processor.

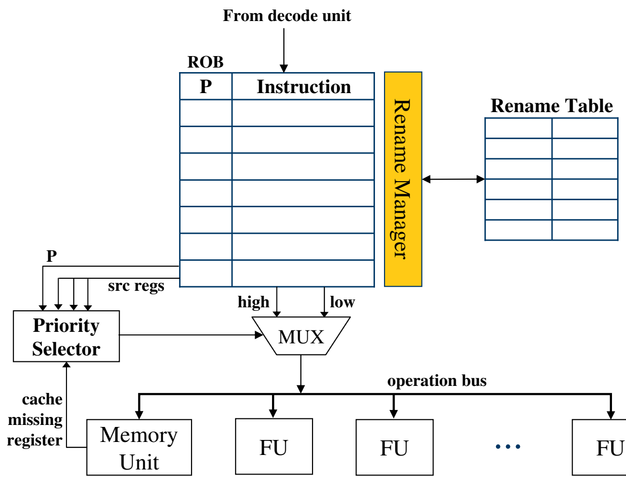


Fig. 4. Architecture Model

When the instructions are pushed into the ROB from decode stage, the rename manager first finds out the instructions which define the source operands of the low priority instructions. Then it renames the source operands of the low priority instruction and the destination operands of the parent instructions by help of the rename table. There are two cases that the rename manager can encounter; 1) the parent instruction which defines the source operand resides in the ROB, and 2) the parent instruction has already been issued to the operation bus and is not in the ROB. In the first case, the destination and source registers can be easily renamed as the traditional register renaming mechanism does. However in the second case, to suspend the execution of the low priority instruction, we should preserve the value of the source registers before it gets changed by the already issued instruction. To do this, the mov instruction which shifts the value from the real register to the rename register is inserted to the ROB. Then the source operand of the low priority instruction is renamed.

In this manner, the low priority instructions get accumulated in the ROB during the high priority execution mode. The low priority instructions start their execution if the priority selector generates '0' and gives it to the mux, or if the ROB is full of the low priority instructions (even if the cache miss is not occurred yet). During the high priority execution, the priority selector keeps generating '1' until the cache miss occurs. When the processor cannot issue the instruction since it should wait for the value from the cache, the priority selector gives inverted P to the mux, and the low priority instructions start their executions. The low priority execution continues until the missed value is available in the cache. When the priority selector detects that the cache miss is now serviced, it generates the high priority signal to the mux and the processor resumes the high priority execution.

IV. Experimental Setup

We demonstrate the effectiveness of the proposed priority based execution technique on a modified version of the Intel XScale microarchitecture [16]. The Intel XScale is a single-fetch, single-decode, single-issue 7-stage out-of-order super-pipelined processor, with 32K 32-way set associative I-Cache and D-cache. We have developed a cycle accurate simulator [28] to model the Intel XScale. Our cycle accurate simulator models the Intel XScale structurally, and is validated against 80200 EVB [15]. Our cycle accurate simulator is less than 10% inaccurate in cycle measurements over benchmarks from the MiBench suite. Note that after simplescalar [3] is parameterized to match the XScale configuration, it is more than 20% inaccurate [28]. We have also integrated the power models from PTScalar [22] to our cycle accurate simulator. PTScalar contains the leakage models as well as thermal models to model both dynamic and the leakage power in the processors.

For the experimental results, we perform the transformation on only the innermost loops of the several multimedia applications. We use 4 MultiMedia applications, H.264 decoder from MediaBench [21], 2 checksum benchmarks from MiBench [12], 3 kernels for swim from SPEC2K [13], and 3 benchmarks from DSPStone [32]. These benchmarks are chosen since they are the important kernels of the multimedia applications and show non-negligible cache miss penalty due to their data intensive calculations. To compile our benchmarks, we used GCC with all the optimizations turned on (-O3).

V. Experiments

A. Effectiveness of Priority-based Execution

We implemented our technique with a ROB of 100 entries. Figure 5 plots the improvement in execution time due to our priority based execution mechanism. In this graph performance improvement is measured as percentage reduction in the runtime of the application.

In the GSR benchmark, much execution time is spent in one loop, in which our compiler was able to discover about 50% of the instructions as the low priority instructions. Consequently, priority based execution was extremely effective and resulted in 39% performance improvement. On an average, our priority based execution could achieve more than 17% performance improvement.

Figure 6 plots the percentage of the cache miss penalty that our technique hides in each benchmark. In more than half of

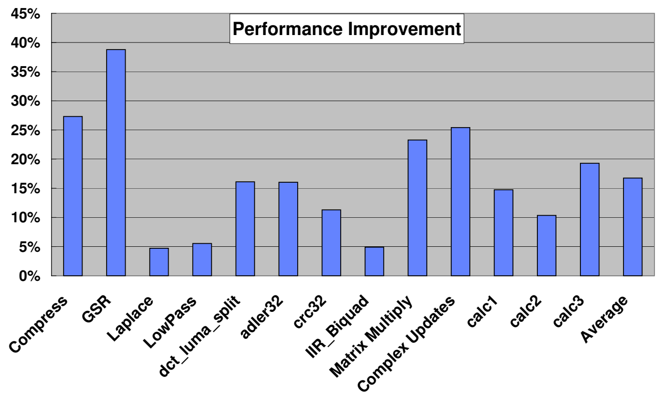


Fig. 5. Our technique achieves 17% reduction in runtime

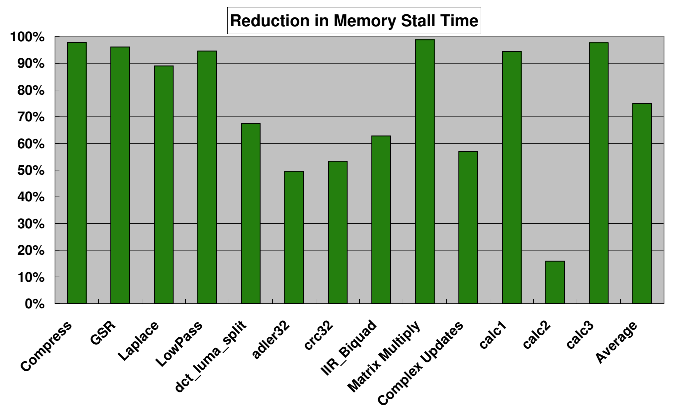


Fig. 6. Our technique hides 75% of cache miss penalty

the benchmarks, our technique can utilize over 90% of the cache miss time by executing the low priority instructions. On the average, priority based execution can hide 75% of the cache miss penalty.

An interesting case is the one of Laplace benchmark. The performance improvement for the laplace benchmark is very small, barely 5%. The reason for this is that the memory behavior of Laplace was very good; only 12% of the total execution time was attributed to cache misses. Thus there was not much room for the performance improvement. The importance of Figure 6 is in showing that irrespective of the scope of the improvement, our technique can hide significant portions of memory latency.

B. Varying ROB Size

Figure 7 shows that the average amount of the miss penalty reduction for the varying size of the ROB. With the small size of the ROB, we can hide about 20% of the cache miss penalty since the ROB can hold the very limited number of the low priority instructions. When all of the low priority instructions in the ROB are executed and the ROB is filled up with only high priority instructions, the processor must stall until the cache miss is resolved. As the size of the ROB increases, more low priority instructions can be accumulated in the ROB, resulting in more reduction of miss penalty. The percentage of the reduction is saturated from the ROB size of 100. The number of the low priority instructions we can execute in the cache miss duration is limited because the average memory latency in the Intel XScale is 75 cycles. Therefore there is no significant change when the ROB contains over 100 entries in it.

C. Power and Performance Trade-offs

To demonstrate the effectiveness of our technique, we compare the energy consumption and performance of our proposed technique (Single Fetch - Single Decode - Single Issue (1F-1D-1I) with priority execution) with similar designs.

Figure 8 plots the power and performance of the four processor configurations running Anagram benchmark of SPEC2000 benchmark. As compared to the 1F-1D-1I in-order processor, our technique has much better performance by hiding cache miss penalty. However it consumes slightly more power than the 1F-1D-1I in-order processor because of the larger instruction window. the 2F-2D-2I in-order processor has slightly better performance than our technique even if it is a multiple issue processor, because the 2F-2D-2I processor cannot hide cache miss penalty which is the fundamental problem of in-order processor. Besides, the 2F-2D-2I processor consumes much more power than our technique. Although the performance of the 2F-2D-2I out-of-order processor is very good, it may consume too much power to be suitable for the embedded systems due to the significant overheads in terms of area, power and chip complexity. An argument could be made that the power consumption of the 2F-2D-2I processor could be reduced by voltage and frequency scaling, but it will still have high area overhead.

VI. Summary

The memory gap has been continuously widening due to the faster rate of the increase of the processor clocks as compared to the memory clocks. Consequently, memory latency hiding techniques are becoming even more important. While the high-end processors can use elaborate techniques like multiple issue, out-of-order execution, speculative execution, value prediction etc., to tolerate high memory latencies, they are often not viable solutions for the embedded processors due to significant area, power and chip complexity overheads. In this

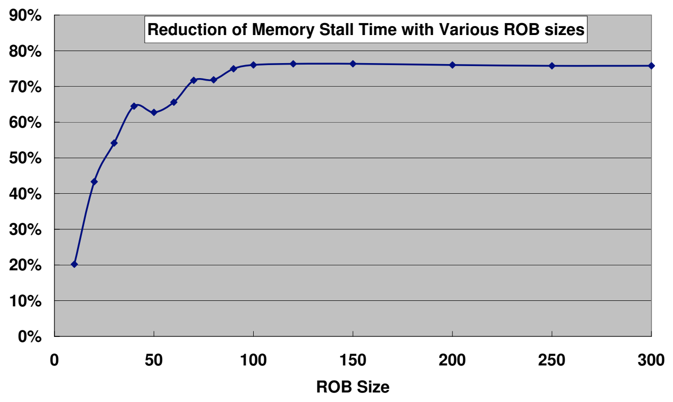


Fig. 7. Cache miss penalty decreases with increasing ROB size

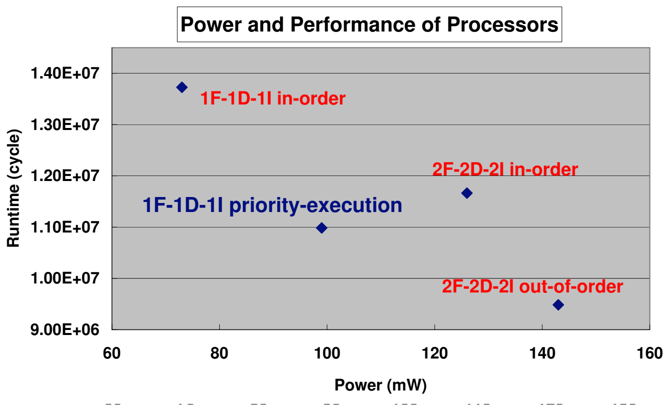


Fig. 8. Power and performance Trade-offs

paper, we present a priority based execution scheme, in which the compiler classifies the instructions into the high priority instructions and the low priority instructions. We are able to hide the cache miss latency by executing the low priority instructions on cache misses. We empirically evaluate our proposal on the Intel XScale compiler and microarchitecture. Experimental results on benchmarks from Multimedia, MediaBench, MiBench, and Speck2K demonstrate an average 17% performance improvements, hiding 75% cache miss penalty. Measuring the effectiveness of this technique on high-issue processors is our future effort.

VII. Acknowledgement

This work is partially supported by a generous grant from Microsoft, IDEC, MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment) (IITA-2006-C1090-0603-0020), the IT R&D program of MIC/IITA. [2007-S001-01, Components/Module technology for Ubiquitous Terminals], Human Resource Development Project for IT SoC Architect, Nano IP/SoC promotion group of Seoul R&BD Program in 2007.

References

[1] Advanced RISC Machines Ltd. *ARM7TDMI (Rev 4) Technical Reference Manual*.

[2] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. *SIGARCH Comput. Archit. News*, 29(2):52–61, 2001.

[3] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[4] A.-H. A. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 486–500, New York, NY, USA, 2001. ACM Press.

[5] H. Balakrishnan and R. Garg. Multimedia benchmarks: A performance comparison of multimedia programs on different architectures.

[6] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 40–52, New York, NY, USA, 1991. ACM Press.

[7] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317, Washington, DC, USA, 2001. IEEE Computer Society.

[8] A. Cristal, O. J. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero. Kilo-instruction processors: Overcoming the memory wall. *IEEE Micro*, 25(3):48–57, 2005.

[9] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 68–75, New York, NY, USA, 1997. ACM Press.

[10] D. Gallagher. Memory disambiguation to facilitate instruction-level parallelism compilation, 1995.

[11] J. P. Grossman. Cheap out-of-order execution using delayed issue. In *ICCD '00: Proceedings of the 2000 IEEE International Conference on Computer Design*, page 549, Washington, DC, USA, 2000. IEEE Computer Society.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[13] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[14] S. Hily and A. Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 64, Washington, DC, USA, 1999. IEEE Computer Society.

[15] Intel Corporation, http://www.intel.com/design/iio/manuals /273411.htm. *Intel 80200 Processor based on Intel XScale Microarchitecture*.

[16] Intel Corporation, http://www.intel.com/design/intelxscale/ 273473.htm. *Intel XScale(R) Core: Developer's Manual*.

[17] G. Jones and N. Topham. Simplifying Hardware for Out of Order Execution using the Decoupling Paradigm. Technical Report ECS-CSG-32-97, 1997.

[18] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pages 252–263, New York, NY, USA, 1997. ACM Press.

[19] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, New York, NY, USA, 1990. ACM Press.

[20] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 59–70, Washington, DC, USA, 2002. IEEE Computer Society.

[21] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[22] W. Liao, L. He, and K. Lepak. Ptscalar version 1.0, 2004.

[23] LSI Logic. *TinyRISC LR4102 Microprocessor Technical Manual*.

[24] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[25] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *MICRO-35: Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 3–14, Washington, DC, USA, 2002. IEEE Computer Society.

[26] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 129, Washington, DC, USA, 2003. IEEE Computer Society.

[27] A. Pajuelo, A. Gonzalez, and M. Valero. Speculative execution for hiding memory latency. *SIGARCH Comput. Archit. News*, 33(3):49–56, 2005.

[28] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau. Operation tables for scheduling in the presence of incomplete bypassing. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 194–199, New York, NY, USA, 2004. ACM Press.

[29] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 25–34, Washington, DC, USA, 2002. IEEE Computer Society.

[30] StarCore. *SC1000-Family Processor Core Reference Manual*, June 10, 2004.

[31] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Retrospective: simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 115–116, New York, NY, USA, 1998. ACM Press.

[32] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr. Dspstone: A dsp-oriented benchmarking methodology. In *Proceedings of Signal Processing Applications and Technology*, Dallas, 1994.