An Efficient CPI Stack Counter Architecture for Superscalar Processors

Osman Allam Stijn Eyerman Lieven Eeckhout
Ghent University, Belgium
{osman.allam, stijn.eyerman, lieven.eeckhout}@elis.UGent.be

ABSTRACT

Cycles-Per-Instruction (CPI) stacks provide intuitive and insightful performance information to software developers. Performance bottlenecks are easily identified from CPI stacks, which hint towards software changes for improving performance.

Computing CPI stacks on contemporary superscalar processors is non-trivial though because of various overlap effects. Prior work proposed a CPI counter architecture for computing CPI stacks on out-of-order processors. The accuracy of the obtained CPI stacks was evaluated previously, however, the hardware overhead analysis was not based on a detailed hardware implementation.

In this paper, we implement the previously proposed CPI counter architecture in hardware and we find that the previous design can be further optimized. We propose a novel hardware- and power-efficient CPI counter architecture that reduces chip area by 44% and power consumption by 47% over the best possible prior design, while maintaining nearly the same level of performance and accuracy.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: Modeling of computer architecture; C.4 [**Computer Systems Organization**]: Performance of Systems—*Modeling Techniques*; B.8.2 [**Hardware**]: Performance and Reliability—*Performance Analysis and Design Aids*

General Terms

Design, Performance, Measurement, Experimentation

Keywords

Superscalar processor, CPI stack, Counter architecture

1. INTRODUCTION

A key role of user-visible hardware performance counters is to provide clear and accurate performance information to the software developer, who then uses the information to guide software changes towards improved performance. An intuitive representation of performance is a cycle stack which breaks up total cycle count to execute a unit of work in its cycle components. A cycle stack consists of a base cycle component (number of cycles during which useful work is done), plus a number of 'lost' cycle components due to miss event handling such as cache misses, branch mispredictions, etc. Dividing a cycle stack by the number of dynamically executed instructions then yields a so-called Cycles-Per-Instruction (CPI) stack [3]. The power of a CPI stack is that it visually highlights the major performance bottlenecks by the large CPI components. For example, a large cache miss component implies that the workload is cache-intensive: software optimizations that improve access locality and/or reduce the working set are likely to improve performance significantly.

Constructing CPI stacks is challenging on contemporary superscalar processors. Out-of-order processors are today's prevalent superscalar processors in the server and desktop domain (e.g., Intel Xeon, AMD Opteron, IBM Power7, etc.) and they are gaining popularity in the embedded space as well (e.g., ARM Cortex A9). These processors feature superscalar out-of-order execution, speculative execution, hardware prefetching, ability to service multiple outstanding memory requests at the same time, non-blocking caches, etc. All of these features enable out-of-order processors to achieve high performance by exploiting instruction-level parallelism (ILP) and memory-level parallelism (MLP), the fundamental reason being able to hide latency through parallel execution. This ability for hiding latency complicates CPI stack construction: overlapping events should not be double-counted.

Prior work by Eyerman et al. [6] proposed a method for computing accurate and meaningful CPI stacks on superscalar out-of-order processors. The method was found to be accurate with an average error around 2.5% and a maximum error of less than 4% compared to detailed cycle-accurate simulation over a range of standardized benchmarks; this is substantially more accurate than previously proposed approaches with average errors around 20% [12]. The CPI stack construction method relies on a CPI counter architecture that is to be implemented in hardware. Eyerman et al. described the CPI counter architecture in a level of detail that is common to computer architecture papers, however, the evaluation of its hardware complexity was limited to counting the number of storage bits needed and was not based on a detailed analysis of the actual hardware implementation.

In this paper, we start off from the initial CPI counter architecture designs proposed by Eyerman et al. [6], we implement the counter architectures in hardware, and we quantify their complexity in terms of performance, area and power consumption. Our results confirm the statement made by Eyerman et al. that hardware complexity is low, however, we found several opportunities for further optimizations. In fact, we reduce the complexity of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'12, May 3–4, 2012, Salt Lake City, Utah, USA.
Copyright 2012 ACM 978-1-4503-1244-8/12/05 ...$10.00.

the CPI counter architecture by reducing the amount of storage needed and by removing the need for Content-Addressable Memory (CAM) lookups. This reduces hardware cost by 44% and reduces power consumption by 47%, while achieving the same performance. Overall, the counter architecture requires no more than 0.03 mm^2 of chip area and consumes 5.8mW at 1GHz in a 90nm CMOS standard cell chip technology. We also evaluate the impact of the optimized counter architecture on the accuracy of the resulting CPI stacks. For a set of SPEC CPU benchmarks, we found the error to be within 3% on average. The overall conclusion is that the proposed CPI counter architecture is feasible to implement in hardware at low cost, low power consumption and high performance while yielding accurate and insightful CPI stacks.

2. PRIOR WORK

Although the basic idea of a CPI stack is simple, computing accurate CPI stacks on superscalar out-of-order processors is challenging because of parallel processing of independent operations and miss events. Eyerman et al. [6] proposed a CPI counter architecture, which, in contrast to prior practice, was designed in a top-down fashion from a mechanistic analytical performance model. This counter architecture was found to be accurate within 2.5% on average (4% max error) whereas prior practice led to average errors around 20% [12]. The key difficulty in designing a CPI counter architecture relates to computing frontend miss penalty cycles: whether a speculative path resolves to a correct path (and whether the penalty cycles need to be accounted for) is not known until a branch is executed on a functional unit in the backend of the pipeline. Hence, we need to keep track of the presumed miss penalty for each in-flight branch, and only account for the penalty if the branch was correctly predicted. Eyerman et al. proposed the FMT and sFMT designs to this end; we refer to [6] for a detailed description of the (s)FMT.

3. NOVEL COUNTER ARCHITECTURE

The (s)FMT structure is fairly complex: it involves storage to keep per-entry ROB IDs, timestamps, and local I-cache/TLB counters. In addition, it also involves Content-Addressable Memory (CAM) logic, i.e., an array of comparators, to find the entry corresponding to a particular mispredicted branch as branches may be executed out-of-order. CAM logic does not scale well and consumes considerable power. We now propose the FIFO-sFMT which reduces chip area overhead and power consumption substantially. The FIFO-sFMT is a FIFO queue that contains branch dispatch timestamps only. It does not store ROB IDs, nor does it involve CAM logic. Note that the FIFO-sFMT accounts for cache misses alike the original (s)FMT designs.

The FIFO-sFMT comes with head and tail pointers. The general mechanism is that, whenever a branch is dispatched, the timestamp (current cycle count) is recorded in the FIFO-sFMT entry pointed to by the tail pointer, after which the tail pointer is incremented. When a branch commits, the head pointer is incremented. The FIFO-sFMT also comes with a novel hardware structure, called the *branch miss handler*, which consists of a timestamp register, a branch mispredict bit and a FIFO pointer, see also Figure 1. The purpose of the branch miss handler is twofold: (1) discarding false-path branches that have already dispatched, and (2) calculating the branch misprediction penalty. The branch mispredict bit is set when a mispredicted branch is resolved, i.e., the branch is executed on a functional unit and the hardware figures out that the branch was mispredicted. When committing a mispredicted branch, we compute the branch misprediction penalty, which we describe next.

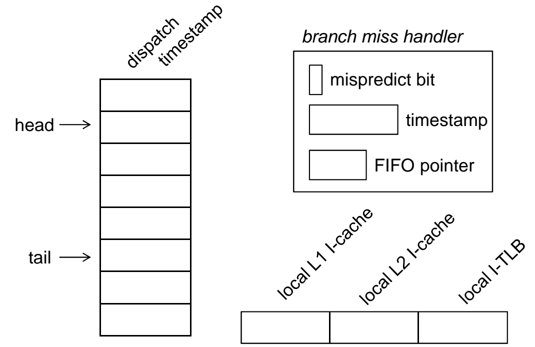

Figure 1: FIFO-sFMT with branch miss handler.

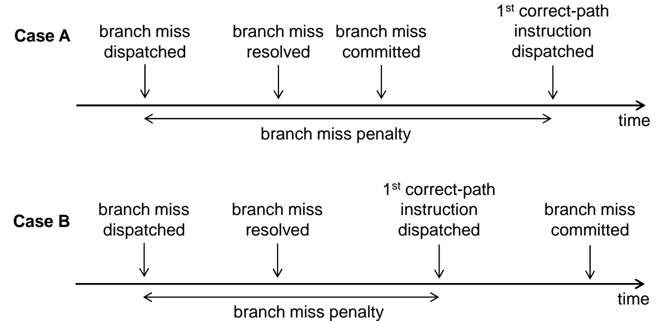

Figure 2: Two possible orderings for committing a mispredicted branch and dispatching correct-path instructions after branch resolution.

To describe the mechanism of the FIFO-sFMT, we consider two cases depending on the relative ordering of when the mispredicted branch commits versus when correct-path instructions are dispatched after the mispredicted branch was resolved.

Case A: Mispredicted branch commits before correct-path instructions are dispatched. The first case happens when the mispredicted branch is committed before new correct-path instructions are dispatched, see Figure 2, case A; this is the most frequent case according to our simulations. Upon the commit of the mispredicted branch, the FIFO-sFMT head pointer points to the dispatch time of that branch. We then store this timestamp value in the branch miss handler's timestamp register, and we set the tail pointer to the entry following the head pointer (thereby discarding the entries containing wrong-path branches). When the first correct-path instruction after the branch misprediction dispatches, the timestamp in the branch miss handler is subtracted from the current cycle count to compute the penalty of the mispredicted branch, and the mispredict bit in the branch miss handler is cleared.

Case B: Mispredicted branch commits after correct-path instructions are dispatched. The second case happens when correct-path instructions are dispatched before the mispredicted branch is committed, see Figure 2, case B. This case is detected if the timestamp in the branch miss handler is not set (because the branch miss is not yet committed) when the first correct-path instruction is dispatched and the branch mispredict bit is set. We then store the dispatch time of the first correct-path instruction in the timestamp of the branch miss handler. We also store the current FIFO-sFMT tail pointer in the branch miss handler's FIFO pointer. When the branch miss eventually commits, we compute the branch penalty by subtracting the timestamp in the FIFO-sFMT pointed to by the head pointer (i.e., dispatch time of the branch) from the timestamp

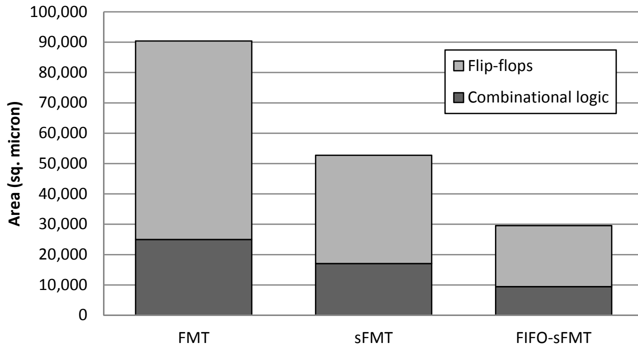

Figure 3: Area in square micron for the FMT, sFMT and FIFO-sFMT.

in the branch miss handler (i.e., dispatch time of the first correct-path instruction after the mispredicted branch). After committing the branch miss, the head pointer needs to be moved to the first correct-path branch, discarding all wrong-path entries. This is done by copying the branch miss handler's FIFO pointer into the FIFO-sFMT tail pointer. Once the penalty is calculated, the mispredict bit in the branch miss handler is cleared, indicating that the branch miss penalty is accounted for.

4. EXPERIMENTAL SETUP

As part of the experimental evaluation, we implement the CPI counter architectures in hardware using VHDL. All experiments are based on standard cell logic synthesis of the counter architectures in isolation. We believe that integrating the model in a full-blown processor and performing physical synthesis would not affect our conclusions. This is due to the compactness of the design and its tolerance to global wire delays. We use a 90nm chip technology from STMicroelectronics — the most advanced technology we have access to; we assume a suppy voltage of 1V (with standard threshold voltage), and we quantify power consumption under typical operating conditions assuming a 1GHz clock frequency. Synthesis is done using Synopsys Design Compiler (version C-2009.06-SP3); RTL and GL simulation is done using Mentor Graphics ModelSim (version 6.3); and power estimation is done using Synopsys PrimeTime-PX (version D-2010.06-SP1).

As a second part of the evaluation, we also evaluate the CPI counter architecture's accuracy. This is done through microarchitecture analysis using detailed cycle-accurate simulation of a 4-wide out-of-order processor. We therefore use the SimpleScalar/Alpha v3.0 simulator [2] along with SPEC CPU2000 benchmarks.

5. RESULTS

We now evaluate the proposed CPI counter architecture through a detailed analysis of its hardware implementation. We subsequently evaluate the impact on accuracy of the obtained CPI stacks. We assume 64 entries for all three CPI counter architecture proposals.

5.1 Hardware implementation

We evaluate the CPI counter architecture in terms of area, power consumption and performance.

Area. Figure 3 quantifies chip area in square micron for the FMT, sFMT and FIFO-sFMT designs, and breaks down total chip area in two terms, namely combinatorial logic versus flip-flops. We observe that the sFMT reduces chip area by 41.6% compared to the FMT. The biggest gain comes from reducing the number of flip-flops in the design by getting rid of the local penalty I-cache/TLB

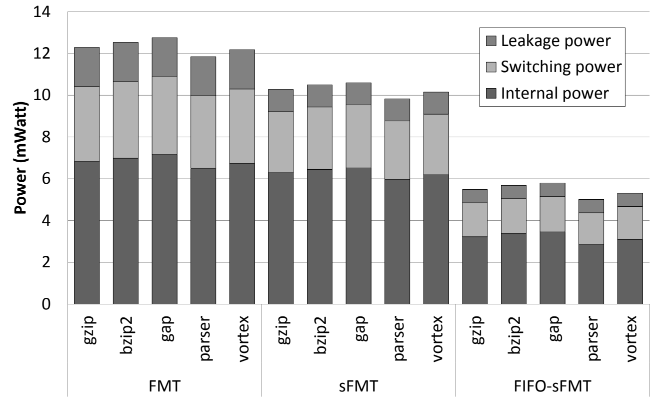

Figure 4: Estimated power consumption in mWatt for the FMT, sFMT and FIFO-sFMT.

counters in the FMT. Combinational logic area savings are achieved by reducing the readout logic of the FMT as a result of reducing the bit width of its entries. The FIFO-sFMT reduces chip area by another 44% over the sFMT. Combinational logic is reduced by removing CAM logic (comparators). The number of flip-flops is reduced by getting rid of the ROB IDs. Overall, the FIFO-sFMT is slightly less than 0.03 mm^2.

Note that these numbers assume a standard cell design using flip-flops. In case of a real processor design, one may use full-custom design: the FMT and sFMT would involve custom CAM-like structures, whereas the FIFO-sFMT would be designed using RAM cells. Clearly, the FIFO-sFMT would involve less design effort and less chip area than both the FMT and sFMT. Unfortunately, we could not study custom designs because we do not have access to RAM macros.

Power consumption. Figure 4 quantifies power consumption. We show a limited number of benchmarks only because of space constraints (we obtained similar results for the other benchmarks). The FIFO-sFMT reduces power consumption by 47% compared to the sFMT. The reduction comes primarily from removing power-hungry CAM logic, which results in a substantial reduction in dynamic power consumption. Leakage power is reduced due to the reduction in the amount of logic and flip-flops in the design. It is further interesting to note that the reduction in power consumption is larger for the FIFO-sFMT compared to the sFMT than for the sFMT versus the FMT. The reason is that the difference between the sFMT and FMT comes from a reduction in the amount of storage, not logic, which leads to a reduction in leakage power; dynamic power is affected less. In other words, we remove relatively inactive circuitry. The FIFO-sFMT on the other hand reduces both leakage and dynamic power over the sFMT due to, as mentioned above, reducing both the number of flip-flops and combinatorial (CAM) logic. The ROB IDs and comparators constitute active circuitry, which explains the large savings in dynamic power.

Performance. We also evaluated performance of the three CPI counter architectures. All three architectures achieved the same performance within 3.3%. The reason is that the critical path in the design is bounded by the number of entries in the structures, which is the same for all three designs.

5.2 Microarchitectural evaluation

Next to understanding the area, power and performance implications of the new CPI counter architecture, it is also important to evaluate the impact on accuracy for the obtained CPI stack. Figure 5 shows the maximum CPI component error across all of the

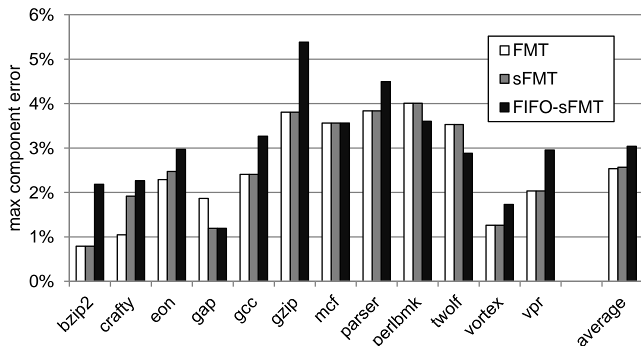

Figure 5: Maximum CPI component error for the FMT, sFMT and FIFO-sFMT.

benchmarks, for the FMT, sFMT and FIFO-sFMT designs. The errors are relative to a CPI stack obtained through detailed cycle-accurate simulation as described in [6]. The average error equals 2.5% and 2.6% for the FMT and sFMT, respectively, with a maximum error of 4%. The FIFO-sFMT has an average error of 3.0% and a maximum error of 5.4%. Although this is a slight decrease in accuracy, the error is still significantly lower than that of the other methods discussed in [6], with average errors around 20%.

There are two reasons for the slight increase in error. First, we assumed only one branch misprediction handler. In case there are multiple outstanding branch misses, we account the penalty of the last one only. Adding multiple branch misprediction handlers will reduce the error compared to the (s)FMT at the cost of involving more hardware. Second, if a branch misprediction is immediately followed by an I-cache miss (i.e., the first correct-path instruction causes an I-cache miss), the FIFO-sFMT will include the instruction cache miss penalty as part of the branch misprediction penalty. This can be solved by not incrementing the timer that generates timestamps if the first correct-path instruction after a branch misprediction results in an I-cache miss. According to our simulations, this would reduce the average error from 3.0% to 2.9%, hence, one may conclude this gain in accuracy does not justify the additional hardware needed.

6. RELATED WORK

A number of proposals have been made for computing CPI stacks for in-order architectures. For example, the Intel Itanium processor family provides a rich set of hardware performance counters for computing CPI stacks [8]. The Digital Continuous Profiling Infrastructure (DCPI) [1] is another example of a hardware performance monitoring tool for an in-order architecture. Computing CPI stacks for in-order architectures, however, is relatively simple compared to computing CPI stacks on out-of-order architectures.

The IBM POWER5 microprocessor was the first out-of-order microprocessor to implement a dedicated counter architecture for computing CPI stacks [12]. The IBM POWER5 approach, however, does not accurately compute the I-cache and I-TLB CPI components; nor does the IBM POWER5 accurately compute the branch misprediction penalty [6]. The Intel Pentium 4 [13] does not have a dedicated counter architecture for computing CPI stacks, but features a mechanism for obtaining non-speculative event counts, i.e., it does not count miss events along mispredicted control flow paths. Cycle accounting for the Intel Nehalem processor is described in [10]. Stall cycles are defined as cycles during which no instructions issue to functional units; further, accounting events to stall cycles is done using heuristics based on existing performance counters.

7. CONCLUSION

CPI stacks provide valuable and insightful performance information to software developers. Computing accurate CPI stacks on contemporary superscalar processors is non-trivial though because of various overlap effects. In this paper, we explored the hardware implementation cost of previously proposed CPI counter architectures and we proposed a new one, called the FIFO-sFMT, which reduces chip area and power consumption substantially by 44% and 47%, respectively, compared to the state of the art. The FIFO-sFMT removes the need for maintaining ROB IDs, thereby reducing chip area and leakage power, and it eliminates CAM logic, thereby reducing both chip area and dynamic power substantially. The FIFO-sFMT achieves these high savings in chip area and power consumption while maintaining high performance and accuracy.

Acknowledgements

Stijn Eyerman is supported through a postdoctoral fellowship by the Research Foundation–Flanders (FWO). Additional support is provided by the FWO projects G.0255.08 and G.0179.10, the UGent-BOF projects 01J14407 and 01Z04109, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

8. REFERENCES

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, Nov. 1997.

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

[3] P. G. Emma. Understanding some simple processor-performance limits. *IBM Journal of Research and Development*, 41(3):215–232, May 1997.

[4] S. Eyerman and L. Eeckhout. A counter architecture for online DVFS profitability estimation. *IEEE Transactions on Computers*, 59(11):1576–1583, Dec. 2010.

[5] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for SMT processor scheduling. In *ASPLOS*, pages 91–102, Mar. 2010.

[6] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, pages 175–184, Oct. 2006.

[7] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2), May 2009.

[8] Intel. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, May 2004. 251110-003.

[9] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In *ISCA*, pages 15–26, June 1998.

[10] D. Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors*. Intel, 2009. http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

[11] Y. Luo, J. Rubio, L. K. John, P. Seshadri, and A. Mericas. Benchmarking internet servers on superscalar machines. *IEEE Computer*, 36(2):34–40, Feb. 2003.

[12] A. Mericas. Performance monitoring on the POWER5 microprocessor. In L. K. John and L. Eeckhout, editors, *Performance Evaluation and Benchmarking*, pages 247–266. CRC Press, 2006.

[13] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, July 2002.

[14] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Supercomputing*, Nov. 1996.