Influence of Processor and Application Characteristics at the Basic Block Level

Francis B. Moreira, Marco A. Z. Alves, Matthias Diener, Philippe O. A. Navaux
Instituto de Informática
Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil
{fbmoreira, mazalves, mdiener, navaux}@inf.ufrgs.br

***Abstract*—This paper evaluates processor and application performance on the basic block level, using a microarchitecture simulator to dynamically profile the behavior of each basic block of an application. We analyze several performance characteristics of each basic block, such as memory accesses, cache misses, branch mispredictions and contention on functional units.**

With this analysis, we can determine the influence of such characteristics on the performance of each workload and investigate optimizations to increase instruction level parallelism on the processor level.

We show that the most important characteristics for processor performance are last level cache (LLC) misses and branch mispredictions. Contention on functional units and L1/L2 cache misses are less relevant as they are hidden by the superscalar processor and the prefetcher.

Eliminating these two sources of performance degradation, using perfect architectures without LLC misses and branch mispredictions, performance was improved by up to 78% and 127% (10% and 23% on average), respectively.

***Keywords*-Computer architecture; Performance analysis; Basic block;**

I. Introduction

Industry continues to increase the performance of cores through instruction level parallelism (ILP) [1], vectorization and multi-core technologies [2]. However, details such as the number of functional units and registers are not available to the current compilers. Thus, they lack knowledge regarding these increasingly complex architectures to make optimization decisions when generating code. This becomes evident with the large amount of hardware in modern superscalar processors, yet small performance increases in instructions per cycle (IPC) [3].

This inefficiency derives from code dependencies between instructions, intrinsic to algorithms and necessary for their correctness. Besides, lack of optimization in the expression of the algorithm, such as unaligned memory accesses and unpredictable branching behavior, results in performance degradation for the code. But such constraints do not occur in every code block, and identifying in which blocks they do and how it affects performance can give us an estimation of the bottlenecks in current architectures. Examples of such problems are delinquent loads [4] and hard-to-predict branches [5].

In our work we use a dynamic analysis of performance at the basic block level. The definition of a basic block [6], [7] is a stretch of code with only one point of entry and one point of exit. Thus, it always begins in an address that is branched to, and generally ends in a branch, or before another instruction that is a branch target. The dynamic analysis results in accurate understanding of representative blocks of the algorithm, unlike static analysis, which suffers of lack of hardware knowledge.

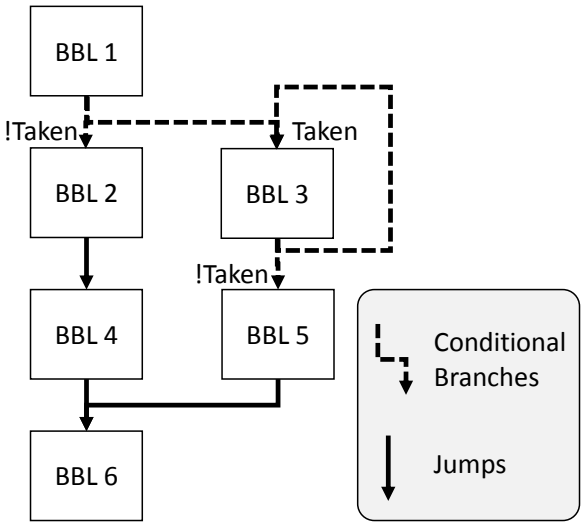

Figure 1. Graph illustrating the difference between basic blocks and branched blocks.

Objectives: The objective of this work is to identify performance bottlenecks in the current superscalar architectures. To this end, we analyze the behavior of the SPEC-CPU2006 workload using a cycle accurate simulator to dynamically obtain statistics for each block, which enables us to obtain information internal to the hardware, such as the functional units contention.

Contributions: Our main contribution in this paper is the evaluation of current architectural characteristics. This evaluation is made considering the main bottlenecks found in our experiments. We change individually each of the bottlenecks, showing the maximum gains if only the problems with said bottleneck were to be eliminated.

II. Motivation

In this work, we use a simpler block definition which we call *branched block*. Only conditional branches are treated as exit points. This gives us coarser granularity and the possibility to detect them at first execution dinamically.

In Figure 1 the difference between basic blocks and branched blocks is shown. Each box represents a basic block, a vertex in the graph. As only conditional branches

Executing core	14 stages, decodes/commits up to 4 inst/cycle, Dispatch/executes out-of-order up to 8 micro ops; reorder buffer of 168 entries
Front end	up to 1 branch per fetch cycle, 4k-entries large BTB, 64k-entries in the hybrid branch predictor (two-level, static-taken)
On-chip caches	L1 I-cache: 32KB, 8-way, 2-cycle, 64B line; L1 D-cache: 32KB, 8-way, 2-cycle, 64B line L2 private cache : 256KB, 8-way, 7-cycle, 64B line ; NUCA LLC shared cache: 16MB, 16-way, 11 cycle (local), 64B line
Prefetchers	L1 data cache:Stride Prefetcher and L2 Data Stream prefetcher, both with 32 table entries, prefetch degree 2, prefetch distance up to 32 cache lines for stream prefetcher, content-directed prefetcher at L2 cache
Memory Controller	*Row-buffer hit* Policy and *service-write-at-no-read*, 64 mshr entries
DDRAM Memory and Bus	DDR3 1600 Mhz, Burst length 8, 4 channels, 8 banks per channel, Latencies: 9-9-9ns (tRP, tRCD, tCL)

Table I
SYSTEM SIMULATION DETAILS

end a branched block, the basic blocks are aggregated in instruction flows which have their behavior aggregated.

The critical point in analyzing behavior at branched block level is what we call *information skew*. In a superscalar architecture, the in-order commit warrants that we can correctly detect blocks. However, statistics take different times to be registered, which can generate a variable distortion for each block. To overcome this, in our simulation we create a table to store the statistics timely for each block.

III. METHODOLOGY

To get statistics, the following was changed in the core: 1) Whenever a conditional branch is inserted in the fetch buffer, an entry is created for the instruction with it's opcode number. 2) Whenever a statistic is registered, we look in the table with the entries, ordered by opcode number, for the first entry whose opcode number is greater than the opcode number of the instruction that generated the statistic. We then increment the statistic in that entry. 3) When a conditional branch commits, we compare all statistics multiplied by their weights and choose the largest result as the relevant characteristic representing the block.

We register the following statistics per block: all cache misses, branches misprediction and contention on ALU integer functional units and all floating point units.

Processor Model: In our tests we simulate a current x86 architecture, Sandy Bridge. All contention in functional units, register dependencies and system constraints are faithfully simulated, with the exception of instruction extensions, such as SSE and MMX. In Table I the architecture specifications are detailed. This core architecture was chosen as it represents a modern, state-of-the-art execution core.

Workload: We use the SPEC-CPU2006 workload. Each program was compiled with gcc 4.7.6 with options -O1 and -static. These are used to ensure no instruction extensions which are not modeled will be used. Each program executes the reference input for 200 million instructions selected by Pinpoints [8].

Characteristic and Simulation Parameters: The characteristics weights are 8 for L1 data cache misses, 32 for L2 data cache misses, 200 for LLC misses, 16 for branch mispredictions and 1 for all contention related, as these are registered every cycle. To measure the performance of a single core, we take into account the average instructions per cycle of the entire execution trace, after a warm-up of 10 million instructions. To evaluate whether our method can meaningfully characterize blocks, we eliminate each bottleneck individually to observe if the performance gains correlate with the characteristics that were pointed as the most frequent source of performance degradation.

IV. RESULTS

Figure 2 shows stacked bars representing the characteristics distribution for all branched block executions in the program. *Others* represent blocks where none of the events were registered. In the remaining figures performance is illustrated per benchmark with a perfect architecture execution for each characteristic, removing it's bottleneck.

In Figure 3 we can observe that performance improvements for a perfect branch predictor correlate well with the programs that had a large rate of mispredictions. Programs *gobmk* and *sjeng* double their IPC, with gains of 127% and 98%. The average gain was of 21.55%.

In Figure 4, all levels show significant performance improvements, but they do not correlate directly to the characteristic of the same level. Rather, they all correlate to the last level cache miss characteristic, as the processor cannot mask this latency. Gains were of up to 96%, 90% and 78% performance for the three cache levels of benchmark *milc*, and 94%, 64% and 27% for benchmark. We can see that *milc* can hide latencies of cache levels better than *mcf*. The average gains for all benchmarks were of 18%, 14% and 10% for each level of perfect cache.

In Figure 5 we see results eliminating functional units contention. The only relevant results are seen in *gromacs* and *bwaves*, where the high latency of the division functional unit cannot be masked. Performance was improved by 26% and 11% respectively. Still the average improvement was at maximum of 2% for division.

V. RELATED WORK

In Panait *et al.* [4] the authors classify load instructions statically according to several heuristics. They define *delinquency* of a load, identifying a small number of loads that are responsible for most of the cache misses. With static analysis, the authors were able to point 10% of the loads that are responsible for more than 90% of the L1 data cache misses. By profiling the basic blocks with the compiler, this number was reduced to 1.3% of loads responsible by 82% of all L1 data cache misses. This is an example of how effective basic block analysis can be.

Sherwood *et al.* [9] characterize the behavior of entire programs based in their basic blocks. They create the concept of basic block vectors (BBV) to characterize

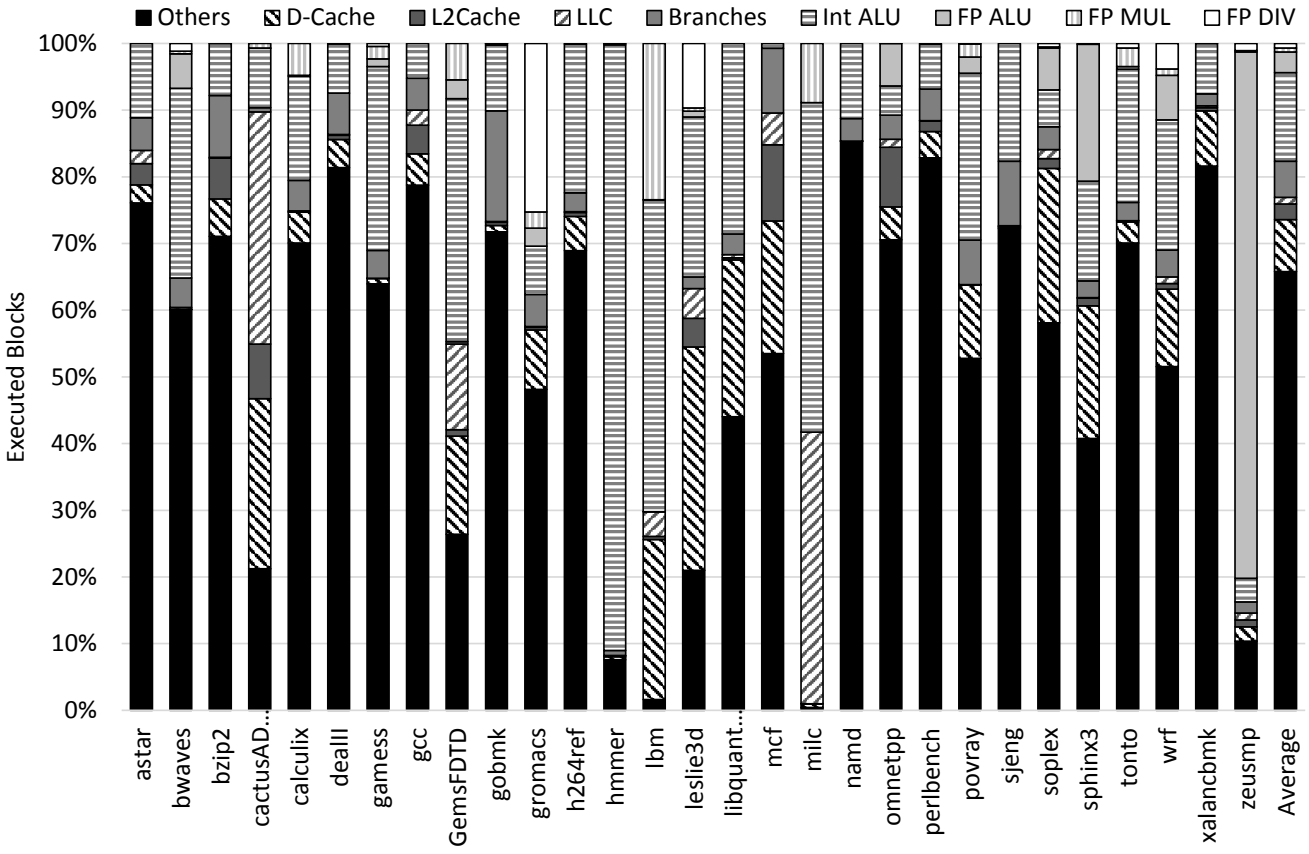

Figure 2. Statistics obtained for the SPECCPU2006 workload. All blocks execution instances are expressed without aggregation per block.

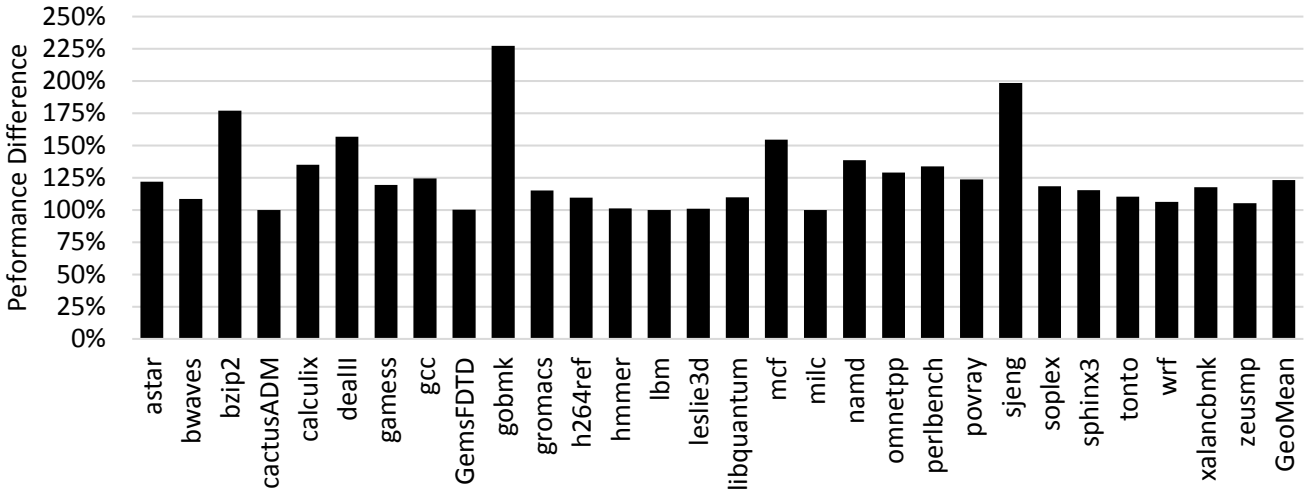

Figure 3. Performance of a perfect branch predictor vs. base processor

a program, and choose program slices to represent the entire program given BBV similarity between the slice and the entire program. Thus, they can also identify program phases given BBV similarity to the entire program. This technique led to Simpoints [10].

Recent work by Kambadur *et al.* [6] creates a static compiler pass analysis method called *Parallel Basic Block Vectors*. Each entry in the vector contains a histogram of how many threads are running for each basic block execution. This allows them to see which basic blocks execute at which levels of parallelism, defining sequential and parallel basic blocks. The work shows several scenarios were the mechanism is useful, such as partitioning of parallel applications, program characterization by parallelism degree and parallelism hotspot analysis.

The advantage of our method is that the statistics obtained are more accurate due to no information skew and greater detail due to knowledge over hardware state.

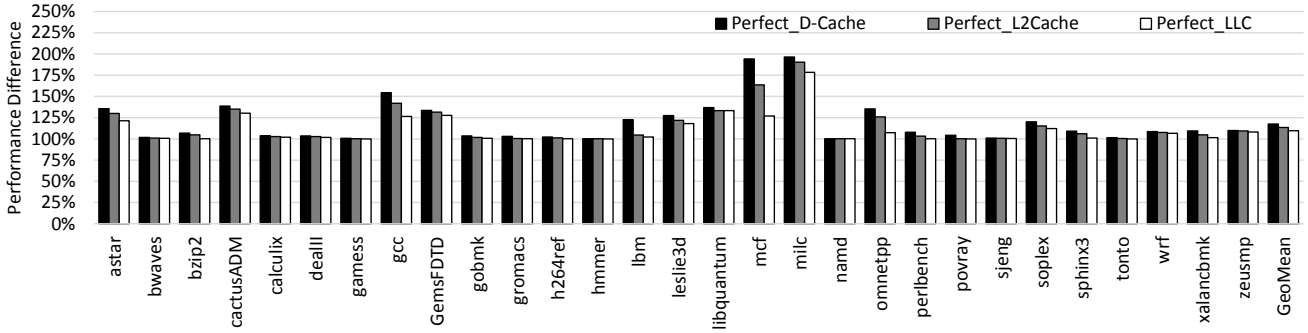

Figure 4. Performance given 100% hit rate for all cache levels

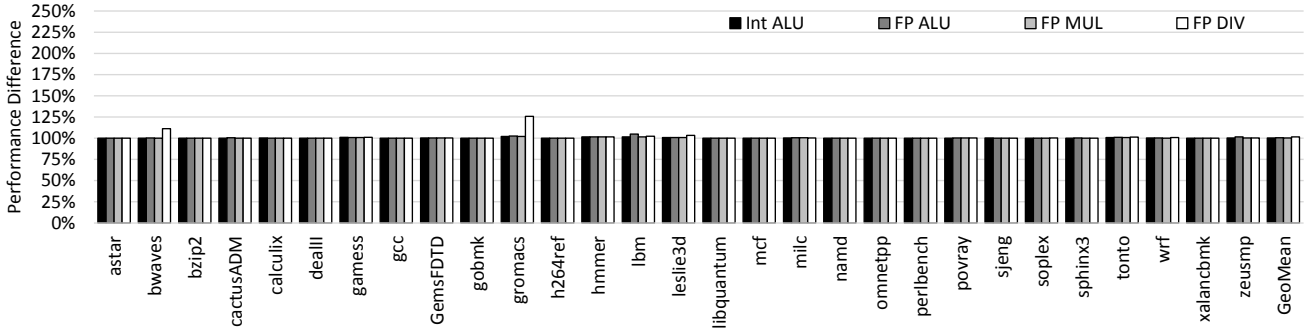

Figure 5. Performance given infinite functional units for each type of functional unit

VI. Conclusions

With our results we were able to conclude that, despite being the focus of processor research in the last years, cache memory and branch prediction are still the most relevant bottlenecks in the processor. The characteristic with the largest influence, even though it the least present, is the branch prediction. The average gains with a perfect branch prediction were of 23%, with maximum of 127% in the benchmark *gobmk*. Although current processors use massive resources to increase hit rate, no technique is used to alleviate the misprediction, and it has too much of a heavy toll on performance, as there is no way to mask it.

Future Work: The aggregation of characteristics was not able to gather all characteristics relevant to the code, and was polluted by functional units contention, which proved to be irrelevant. For the future, we aim to collect statistics that take into account clear register dependencies between instructions, which should occupy and eliminate the space of *others* in our tests.

References

[1] S. Jarp, A. Lazzaro, J. Leduc, and A. Nowak, "Evaluation of the intel sandy bridge-ep server processor," *CERN openlab: Switzerland, March*, 2012.

[2] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2011, pp. 365–376.

[3] D. W. Wall, "Wrl research report 93/6," DEC Western Research Laborator, Tech. Rep., 1993.

[4] V.-M. Panait, A. Sasturkar, and W.-F. Wong, "Static identification of delinquent loads," in *International Symposium on Code Generation and Optimization, CGO*, 2004, pp. 303–314.

[5] H. Gao, Y. Ma, M. Dimitrov, and H. Zhou, "Address-branch correlation: A novel locality for long-latency hard-to-predict branches," in *IEEE 14th International Symposium on High Performance Computer Architecture, HPCA.*, 2008, pp. 74–85.

[6] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: collection and analysis of parallel block vectors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA 2012*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 452–463. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337211

[7] J. Huang and D. Lilja, "Extending value reuse to basic blocks with compiler support," *Computers, IEEE Transactions on*, vol. 49, no. 4, pp. 331–347, 2000.

[8] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel itanium programs with dynamic instrumentation," in *37th International Symposium on Microarchitecture, MICRO-37*, 2004, pp. 81–92.

[9] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT*, 2001, pp. 3–14.

[10] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.