2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing

Profiling and Reducing Micro-Architecture Bottlenecks at the Hardware Level

Francis B. Moreira, Marco A. Z. Alves,
Matthias Diener, Philippe O. A. Navaux
Informatics Institute, Federal University of Rio Grande do Sul
Porto Alegre, Brazil
E-mail: {fbmoreira, mazalves, mdiener, navaux}@inf.ufrgs.br

Israel Koren
Dept. of Electrical and Computer Engineering
University of Massachusetts
Amherst, United States
E-mail: koren@ecs.umass.edu

***Abstract*—Most mechanisms in current superscalar processors use instruction granularity information for speculation, such as branch predictors or prefetchers. However, many of these characteristics can be obtained at the basic block level, increasing the amount of code that can be covered while requiring less space to store the data. Furthermore, the code can be profiled more accurately and provide a higher variety of information by analyzing different instruction types inside a block. Because of these advantages, block-level analysis can offer more opportunities for mechanisms that use this information. For example, it is possible to integrate information about branch prediction and memory accesses to provide precise information for speculative mechanisms, increasing accuracy and performance.**

We propose a Block-Level Architecture Profiler (BLAP), an online mechanism that profiles bottlenecks at the micro-architectural level, such as delinquent memory loads, hard-to-predict branches and contention for functional units. BLAP works at the basic block level, providing information that can be used to reduce the impact of these bottlenecks. A prefetch dropping mechanism and a memory controller policy were developed to use the profiled information provided by BLAP. Together, these mechanisms are able to improve performance by up to 17.39% (3.90% on average). Our technique showed average gains of 13.14% when evaluated under high memory pressure due to highly aggressive prefetch.

I. Introduction

Characterization of basic blocks is an important, recurring technique, used for automatic optimization of several kinds. Software tools such as Vtune [1] allow manual analysis to detect performance improvement opportunities, such as rewriting code to avoid high cache miss rates or high branch misprediction rates for specific basic blocks, known as hotspots. The basic block granularity is especially useful [2] as basic blocks represent portions of code that always end with conditional or unconditional branch instructions. A program's execution path is therefore defined by basic block execution sequences, enabling a program phase characterization and dynamic optimization. A recent example is the work of Kambadur et al. [3], which uses basic blocks to characterize the thread-level parallelism of an application in its different phases.

General-purpose processor designs [4] only collect information at the instruction level. Although several research papers used basic block analysis, most did so using a software approach, even for hardware adaptations [5], [6]. One of the few techniques that actually performed basic block analysis at the hardware level was the rePlay framework [7]. It analyzes the code to perform on-line code optimization which is stored in a trace cache for future use, although no bottleneck profiling is performed. Block profiling is usually done in software due to the high complexity of detailed profiling and the analysis required. Nevertheless, profiling in hardware is interesting as it can leverage current hardware state information to efficiently generate relevant information of a program's execution, requiring no pre-analysis or source code modification.

In this paper, we introduce our Block-Level Architecture Profiler (BLAP). BLAP characterizes basic blocks according to the most relevant delays occurring per block, thus allowing improvement of a block's future executions. BLAP has several advantages over other mechanisms. It adapts to program phase changes, as it dynamically keeps track of basic blocks. It requires less storage than instruction-granularity mechanisms, as we aggregate the behavior per block. We are able to use the Branch Target Buffer (BTB) to efficiently store this information, as it retains the initial address of each block. BLAP is capable of detecting different types of performance-related issues within a block, thus being able to provide information to a wide range of mechanisms.

In order to show the potential of BLAP, we explore the use of its profiling information to design an improved memory controller. Compared with the instruction-granularity information used by Ghose et al. [8] and Lee et al. [9], our mechanism achieves better performance with a scalable hardware overhead. Moreover, BLAP's basic implementation can be extended to provide detailed information regarding a wide range of bottlenecks at low hardware costs. To the best of our knowledge, no previous research has profiled basic blocks in hardware. Moreover, we present an integration between BLAP and other mechanisms, in order to show the usefulness of the profiled information.

The main contributions of this paper are the following:
Characterization Mechanism: We propose BLAP, an efficient detection mechanism capable of characterizing applications at the basic block level during their execution.
Low Overhead Profile: Our mechanism requires negligible storage to keep information about the relevant characteristics for each basic block. Such a mechanism can be implemented by extending the BTB with a few extra bits per entry.
Performance Improvement: We integrated BLAP with previous mechanisms that improve memory performance, by adapting them to use information provided by BLAP. We also combine BLAP with a new policy to drop prefetches.

1550-6533/14 $31.00 © 2014 IEEE
DOI 10.1109/SBAC-PAD.2014.19

The main objective of this work is to propose and study a hardware mechanism that is capable of detecting the blocks which build a program and characterize their behavior. Such characterizations can make it possible to improve the processor performance through its usage by other mechanisms, such as prefetchers or priority policies.

II. Motivation

In this section, we will explore the relationship between basic blocks and processor performance. We use here a relaxed definition of a basic block [2], [10]. A basic block is a portion of code with a single point of entry and a single point of exit. Thus, every basic block ends with a branch instruction, either a conditional or unconditional branch. This enables mechanisms based on basic blocks to keep up with the program phase automatically, as a program's phase is characterized by the blocks being used [6]. Our definition allows for multiple entry points, as it is not possible to efficiently detect the beginning of a block which was not targeted by a branch.

A design issue to be considered when extending the BTB is that it only records information for blocks after taken branches. Given that the behavior to be exploited is usually repetitive, this is normally not a problem, as the repetition of blocks begins after taken branches. Another issue is that we cannot recognize branch targets unless their corresponding branch occurred. This breaks the definition of a basic block, as we will likely record blocks with overlapping information. These blocks will aggregate behavior from all the instructions in the few, smaller real basic blocks inside them, and thus will not be characterized separately. However the smaller basic blocks will be correctly characterized once they are targeted by a branch, thus obtaining their correct starting address. As in most cases smaller blocks represent conditions inside loops, they will be executed enough times to be characterized. If they do not, then they are likely not relevant.

To demonstrate the behavior that can be observed for our relaxed block definition and its correlation with performance, we statistically correlated execution events (such as branch mispredictions) to performance, using the Pearson Moment-Product Correlation Coefficient. The resulting coefficient lies between -1 and 1. The higher the absolute value the stronger is the correlation between the parameters. If the coefficient is negative, the parameters are inversely correlated (e.g. the value of the parameters influence each other, but when one increases, the other decreases). If it is positive, they are correlated, both values increase or decrease together.

The details of the configuration and benchmarks used can be found in Section V. To calculate the correlation, we generated a trace of the execution. This trace contained the most important processor events relevant for execution performance: L1 data cache misses, L2 cache misses, Last-Level Cache (LLC) misses, branch mispredictions, and number of floating point arithmetic-logic instructions and division instructions. Whenever a basic block finished executing, we recorded the number of instructions the block contained, and how many cycles it took to execute, in order to measure its performance. We then recorded how many of the events happened during the execution of that block. For each parallel application from the NAS-NPB and SPEC-OMP2001 suites, each correlation

TABLE I. Pearson moment-product correlation coefficients between events within a block and the Instructions per Cycle.

	Benchmark	L1D Misses	L2 Misses	LLC Misses	Branch Mispred.	FP ALU Inst.	FP DIV Inst.
NAS-NPB	BT	-0.28	-0.34	**-0.39**	-0.01	-0.14	-0.15
	CG	**-0.63**	-0.44	-0.48	0.00	0.13	0.13
	FT	**-0.51**	-0.31	-0.25	-0.05	0.04	0.05
	IS	**-0.18**	-0.16	-0.16	-0.01	-0.00	0.00
	LU	0.04	0.02	**-0.14**	0.01	0.11	0.10
	MG	**-0.34**	-0.28	-0.28	-0.02	0.06	-0.23
	SP	**-0.40**	-0.36	-0.31	-0.05	-0.27	-0.34
SPEC-OMP 2001	Applu	**-0.45**	-0.45	-0.39	-0.01	0.26	0.00
	Apsi	-0.13	-0.14	-0.14	0.01	**-0.27**	-0.26
	Fma3d	-0.27	**-0.33**	**-0.33**	-0.03	-0.00	0.12
	Galgel	-0.18	**-0.21**	-0.08	-0.01	0.21	0.25
	Mgrid	-0.30	-0.29	-0.28	-0.01	**-0.49**	-0.45
	Swim	**-0.69**	-0.60	-0.59	-0.01	-0.40	-0.32
	Wupwise	**-0.58**	-0.50	-0.47	-0.00	0.01	-0.06

coefficient was calculated considering blocks from all the threads together.

The correlation results are shown in Table I. The highest correlation coefficients for each benchmark are marked in bold. Looking at the cache misses correlation coefficients, we can observe a diminishing correlation as we go from smaller and faster to slower and larger caches. Although a miss in the LLC means a main memory access, which is likely to stall the processor, the number of accesses the LLC receives is small, because most accesses are serviced by higher level caches. Therefore, although a single LLC miss has a considerable impact on the final performance, it happens much less frequently than L1 and L2 misses, such that it does not correlate highly with the performance differences between blocks.

Although a branch misprediction in a 15-stage pipeline results a large number of stall cycles, the correlation coefficient of the Branch Misprediction is consistently lower than the other coefficients. This low value of the correlation coefficient is due to the low branch misprediction rate of less than 1% in the benchmarks. Floating-point instructions per block correlate well on a few benchmarks. We can observe that for Apsi and Mgrid, the number of floating-point ALU instructions per block has the highest correlation with degraded performance.

Following this analysis which shows that L1D misses have the highest correlation with processor performance, we seek to improve the memory access bottleneck. However, obtaining detailed hardware statistics per block during execution is a complex matter. Attempting to aggregate behavior and singularly identify blocks using statistics has three challenges. First, the statistic must show a direct impact on the performance. While cache misses are intuitively relevant [5], current architectures are usually tolerant to L1D misses due to high Instruction Level Parallelism (ILP), which provides enough computation to overlap the cache access latency. That is, for most cases, L1D misses do not stall the processor.

Second, different hardware events cannot be directly compared. When a L1D miss occurs, we know that it will take at least L1D access time plus L2 access time for a request to be serviced, but we do not know the state of the Miss-Status Handling Registers (MSHR) of each cache, or even if the cache line will be serviced in L2. Even such a large latency could be hidden in the presence of a branch misprediction. If we want to find what was the most relevant bottleneck in a block, we cannot compare such a value directly to the delay generated by, for example, a floating-point division unit, as we do not know whether there is any instruction that depends on the division result, or if it is actually going to stall the commit stage. Third, hardware counters cannot be used directly to profile the block. As blocks of instructions are committed, we do not know which statistics belong to which block. As an example, if instructions from a block have executed and are ready to commit, we gathered all its statistics, and once the last instruction from the block commits, we should reset the counters to gather statistics for the next block. However, instructions from the next block might be altering these statistics, preventing us from accurately profiling a block.

To overcome these challenges, we exploit the commit stage. Instructions only cause bottlenecks or delay the pipeline if eventually this leads to the commit stage being blocked. So, in order to compare instruction delays, we only look at how many cycles each instruction stalled the commit stage. This approach will obtain information that directly impacts performance (first issue), since we are focusing on the commit stage stalls. We can directly compare commit stalls between instructions, since they are measured in terms of cycles (second issue). Finally, as we do not use hardware counters, the statistics are not skewed (third issue).

In summary, a potential hardware mechanism that identifies the bottlenecks using the commit stage stalls has new relevant applications and requirements. It must be able to meaningfully characterize blocks, requiring small logic overhead. This is possible by recording the number of stalls of the instruction at the head of the Reorder Buffer (ROB), and detecting branch instructions to observe block boundaries. It is also required to efficiently store this profile. Therefore, the information for each block should be kept to a minimum. Finally, the mechanism should be able to provide varied characterization, so multiple mechanisms can use the profile provided. Based on the correlation results, we chose to record the following characteristics: *None* to denote that the block has negligible stalls, *Brch* to denote hard-to-predict branches, *Mem* to denote commit stalls due to loads and *FP* to denote commit stalls due to any floating-point unit. Although the correlation for branches has shown low values, we keep this option for future extensions as we observed high correlation coefficients for sequential benchmarks. Correlation values for sequential benchmarks are omitted for brevity.

III. Related Work

Sherwood et al. [11] is the precursor to SimPoints [12] and other works, such as Pinpoints [13]. The authors characterize the behavior of entire programs based on the analysis of basic block execution distribution. The concept of a Basic Block Vector (BBV) is first introduced to characterize a program. A basic block vector contains execution counts for all basic blocks, normalized by the total amount of executions. Therefore, the vector gives the execution frequency of each basic block in relation to the entire program. In this way, the authors are able to compare the behavior for executions of different instruction counts and for different inputs. Next, a BBV comparison is performed, by treating each BBV as a fingerprint of the program slice observed. To generate this difference, the BBVs are subtracted from each other, and the differences are summed up.

With this comparison, the authors show a variety of features of their implementation. They are able to identify the different phases of a program, such as the initialization phase of a program, due to the considerable difference between the BBV obtained for the first 100 million instructions and the BBV of the entire program. With the BBV of the entire program, they are also able to identify or create BBVs that have a near-identical fingerprint, but with a much smaller number of instructions. They show that the behavior of selected program slices with near-identical BBV are similar, with statistics pertaining to cache misses, branch mispredictions and overall type of instructions executed differing at most by 3%. This was further improved in SimPoints, which can use the Pin instrumentation tool [14] to build simulation points based on this technique. The importance of these techniques for our work is that our methodology uses Pinpoints to simulate programs in a reasonable time. Sherwood's work also shows that by improving only the performance of the repetitive blocks that define the entire program behavior, we can achieve overall improved performance.

The rePLay framework [7] has a similar concept to our work. In this work, the authors use an extended definition of a block called a *frame*. A frame aggregates several basic blocks, as it ignores unconditional branches, and promotes easy-to-predict branches into assertions [15]. They also provide a scheme to replay a frame in case an assertion fires, which signals a misprediction. In this way, they achieve a coarse granularity, enabling compiler-like code optimizations during execution, and alongside the rollback mechanism, the opportunity for very aggressive speculative techniques, such as value prediction and value reuse [16]. Although the framework is described, it is not explored in the paper. Frames intuitively have few opportunities for value reuse and value prediction, as they are coarse enough to capture several executions of loops, and seem to represent distinct phases of data progression in programs. The authors only show manual optimizations in single frame examples, and do not show any mechanism that can make automatic runtime optimizations using the frames collected. In our work, we characterize application behavior on a finer granularity, so we can better inform other mechanisms.

The recent work of Kambadur et al. [3] also uses a simple profiling method they call Parallel Basic Block Vectors. It can be viewed as an extension of Sherwood's work, as now every entry in the vector contains n slots, each representing a degree of parallelism. When a basic block is executed, the number of parallel threads is observed and used as index to increment the appropriate part of the entry. This allows the authors to identify which basic blocks execute at which frequency and at which parallelism level, clearly identifying sequential and parallel code blocks. They also identify the most critical regions of code in terms of performance. Several scenarios are illustrated, showing how this analysis can be applied, such as serial and parallel application partitioning, or analysis of program features by degree of parallelism and parallelism hotspots.

The Criticality Binary Prediction (CBP) mechanism [8] observes how many cycles each load instruction stalls the commit stage and gives priority to the loads that stall. As it only keeps track of the loads, it uses a small 64-bit tagless table per core, which is reset every 100000 cycles to adapt to the current program phase. It then gives priority to the loads found in the table. The paper explores more options, such as storing the number of stalled cycles for more complex policies, but overall it uses only 1 bit per table entry and for all memory request buffer entries for the best trade-off.

Compared to rePlay and CBP, we follow a different approach, by using the behavior detected to improve existing hardware mechanisms that need to detect phase changes and bottlenecks during execution. We evaluate CBP and the prefetch-aware DRAM controller (PADC) [9] by integrating them with BLAP and show the resulting improvements in Section VI.

IV. Block Level Architectural Profiler

In this section, we introduce the Block-Level Architecture Profiler (BLAP) and present its implementation in hardware. BLAP consists of three parts: Behavior Detection, Behavior Labeling and Behavior Storage. Afterwards, we explain potential critical path implications and how they were avoided. We then list the hardware overhead costs of these three stages and describe additions to further improve the profile information.

Figure 1 shows an abstraction of the BLAP implementation inside an Out-of-Order (OoO) processor. BLAP's basic idea is to characterize the bottleneck of a block by detecting instruction stalls and storing the information in the BTB when the last instruction of the block commits.

A. Behavior Detection

With an in-order commit stage, instructions that take long to commit may stall the whole processor. We consider the instructions which stall the commit stage for the longest amount of time to be the instructions that characterize the block's performance issues. We developed BLAP, which modifies the commit stage to obtain profile information at a basic block level.

In Figure 2, we show a flowchart of the commit stage in a superscalar processor with the additional events needed to implement our detection mechanism. Notice that the BLAP implementation requires an in-order commit stage, which is widely used in current commercial processors.

In the commit stage, we must check whether the oldest instruction is ready to commit $\boxed{1}$. Whenever an instruction stalls the commit, a *Stall Counter* is incremented $\boxed{2}$. When the instruction is ready to be committed, we must check if it is a branch $\boxed{3}$, as a branch indicates the end of a block. If it is not a branch, we must also check whether it is the first instruction being committed in that cycle, as we only need the instructions which stalled the commit $\boxed{4}$. If it is not the first instruction being committed and it is not a branch, no action is necessary, as the instruction has not blocked the stage nor is it a branch. However, if this is the first instruction, we must compare its accumulated stall time to the previous largest stall time of the current block $\boxed{5}$. If the stall is larger, we update the register keeping track of the bottleneck with the current instruction type $\boxed{6}$ and reset the stall counter $\boxed{11}$. Otherwise, we skip the update and reset the stall counter at $\boxed{11}$.

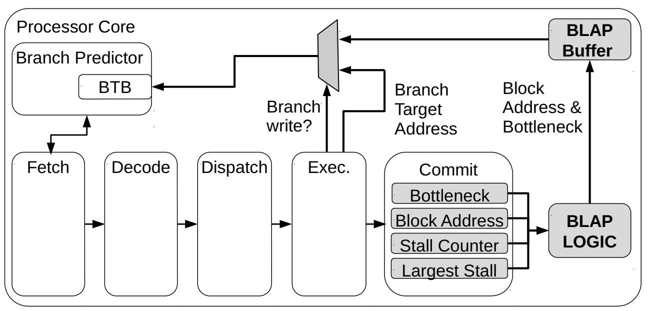
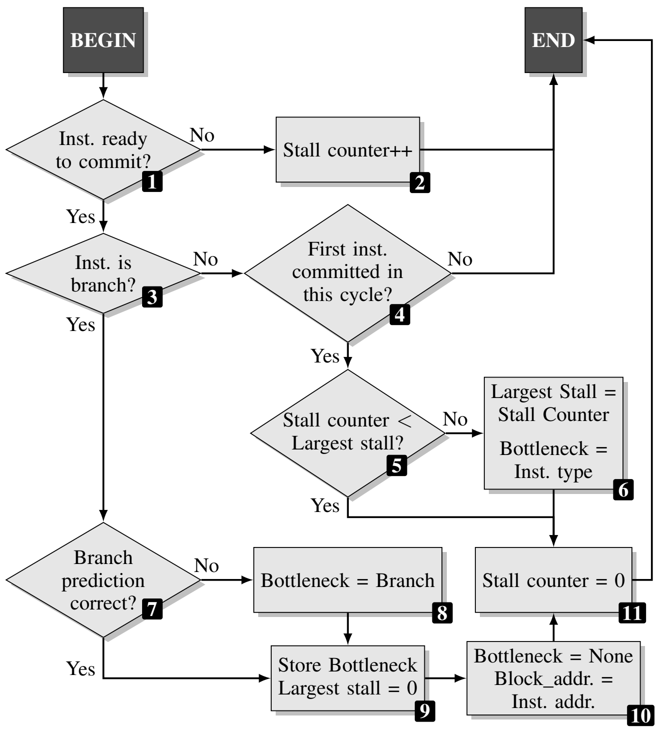


Fig. 1. Overview of the operation of BLAP in a superscalar processor. Parts in gray represent BLAP's modifications or additions to the processor.


BEGIN
END
Inst. ready to commit? 1
No
Stall counter++ 2
Yes
Inst. is branch? 3
No
First inst. committed in this cycle? 4
No
Yes
Yes
Stall counter < Largest stall? 5
No
Largest Stall = Stall Counter
Bottleneck = Inst. type 6
Yes
Branch prediction correct? 7
No
Bottleneck = Branch 8
Stall counter = 0 11
Yes
Store Bottleneck Largest stall = 0 9
Bottleneck = None Block_addr. = Inst. addr. 10
Fig. 2. Flow chart of commit stage modifications.

If the instruction is a branch, we must store the block information. First, we check whether the branch was correctly predicted $\boxed{7}$. As branch instructions do not stall the commit stage, the only way to characterize a block as branch is to find out whether it was mispredicted. If the branch is not correctly predicted, we change the block's bottleneck type to *Brch* $\boxed{8}$. We then store the bottleneck type in the BTB, using the value of the Block_Address register as index (the address of the branch that started the block) and store the instruction address of the current branch instruction in the Block_Address (as we are starting a new block) $\boxed{10}$. In this way, at the end of each block, we store the block information using the instruction address of the branch of the previous block as the index $\boxed{9}$. We also reset the counters related to largest stall $\boxed{9}$ and bottleneck $\boxed{10}$, followed by resetting the stall counter $\boxed{11}$.

In order to prevent profiling characteristics to be affected by cold start effects in caches, prefetchers and branch predictors, we designed a stabilizer for BLAP. This stabilizer deals with problems that may arise when a block has an unstable characteristic in its first executions. It uses a saturating counter to record the number of times the basic block was executed. When this counter saturates, the last detected characteristic is considered to be stable, and thus it is identified as the bottleneck for that block. Further changes in the block bottleneck will not overwrite the BTB entry, in order to avoid disabling the improvements that may have caused the bottleneck reduction and subsequent characterization change.

B. Behavior Storage

In order to use the profile, we must store it for future use. Based on the correlation results, we use 2 bits per block, expressing 4 characteristics (*None*, *Brch*, *Mem*, *FP*). The number of characteristics can be increased by using more bits per entry, for future extensions. As the BTB contains all the conditional and unconditional branch targets, we can extend the BTB to store characteristics for each block targeted by a branch. That way, if a branch is taken, we can load the characteristic stored in the BTB entry as we know that it is the characteristic profiled for the next block. The register mentioned in the previous section, *Block_Address*, is responsible for indexing each block in the BTB. A 2-bits saturating counter is also used per entry to stabilize a block's characteristic.

C. Behavior Labeling

To use our profile information, we created a general approach to allow implementation of multiple mechanisms. When a branch is predicted at the fetch stage, we access the BTB using the address of the branch instruction. The characteristic is loaded into a new register called *Block Characteristic*. The information of this register is copied to a new field for every entry for that instruction (e.g. the ROB) until the content of *Block Characteristic* is updated by the next block profile information. Thus, the fetch buffer's entries, the decode buffer's entries and the ROB entries are all augmented by 2 bits to store the characteristic pertaining to the block.

D. Critical Path Implications

The detection scheme of BLAP requires additional hardware to support the update of the mechanism's registers. There is a special case which occurs when two branches commit in the same cycle. This means that the block initiated by the first branch had no stalls, so we aggregate the block with whatever instructions are committed after the second, ignored branch. In our evaluations, cycles which committed more than one branch represented less than 1% of the execution cycles for NAS-NPB and SPEC-OMP2001 benchmarks. Finally, storing information in the BTB in the same cycle could require a longer cycle time. Thus, we write to a buffer and create an additional pipeline stage used only for BLAP, which stores the information received by the last block in the BTB. This extra stage does not affect the processor's throughput.

The extra stage in BLAP is used to pipeline the actual write to ensure synchronization with the BTB reads performed by all instructions being fetched, so the written value is only valid for the next cycle. Labeling has no implications on the critical path, requiring only additional information bits going through the pipeline along with their respective instructions.

E. Hardware Costs and Design Considerations of BLAP

BLAP's hardware costs can be divided into detection, storage and labeling. Detection requires two 8-bit counters, Stall and Largest Stall. It also requires an 8-bit adder and an 8-bit comparator for these registers. We use two 2-bit Bottleneck registers, and two Block Address registers, to pipeline the actual write to the BTB with an additional BLAP Internal stage. To determine whether a branch was mispredicted, we add 1 bit for each reorder buffer entry.

For storage, we modify the BTB write port to write the extra BTB bits. The value in BLAP's write buffer waits until no branch instruction has a branch target address to write in the same cycle, and one extra bit indicates the entry bits to be written (branch target address or BLAP information). If another block information coming from BLAP would overwrite BLAP's buffer while it waits for a write opportunity, we ignore the second block information as the stall value is likely low for such conflict to occur. Optionally, we can add an additional write port to avoid this issue.

We store the labels in a Block Characteristic register, as we obtain the information bits from the BTB in the fetch stage. Every following instruction of the block must copy this information, so we must add these bits to the entries of all structures. We add 2 bits to every entry of the fetch buffer, decode buffer and ROB.

The hardware costs are shown in Table II. For each core, BLAP requires a total storage of 2142 bytes, plus three 2-bit multiplexer, one 8-bit multiplexer, one 64-bit multiplexer, an 8-bit adder and an 8-bit comparator. Therefore, the total area overhead per core consists of 206548 transistors. In Sandy Bridge, that means 1652384 transistors, out of 2.27 billion transistors, corresponding to an overhead of less than 0.08%.

TABLE II. HARDWARE COSTS OF BLAP.

Item	Cost

Detection	8-bit Stall counter; 8-bit Largest Stall counter;
	8-bit adder for Stall counter;
	8-bit comparison (Stall counter with Largest Stall);
	2-bit Bottleneck reg. (Commit);
	64-bit Block Addr. reg. (Commit);
	1-bit pred. info. per ROB entry (168 entries total);
	1 2-input 2-bit multiplexer, uses branch pred. info. to update BLAP;
	1 2-input 2-bit multiplexer, bottleneck evaluation and selection;
	1 2-input 8-bit multiplexer, stall evaluation and selection;
	Total of 316 bits for detection;

Storage	2-bit Bottleneck reg. per BTB entry (4096 entries total);
	2-bit Stabilizer counter per BTB entry (4096 entries total);
	2-bit Multiplexer, selects between BLAP and regular branch write;
	2-bit Bottleneck reg. (BLAP Buffer);
	1 64-bit Block Addr. reg. (BLAP Buffer);
	1 2-input 2-bit multiplexer (selects BTB write data);
	1 2-input 64-bit multiplexer (selects BTB write address);
	Total of 2048 bytes of storage;

Label	2-bit Block Characteristic reg.; 2-bit Block Characteristic per fetch buffer entry (18 entries total); 2-bit Block Characteristic per decode buffer entry (28 entries total); 2-bit Block Characteristic per ROB entry (168 entries total); Total of 430 bits for labeling;

V. Evaluation Methodology

A. Simulation Environment and Workloads

To validate our mechanism, we used a cycle-accurate in-house simulator [17]. It accurately simulates the micro-architecture, modeling all functional unit contention, register dependencies, processor system restrictions (e.g. memory disambiguation), in addition to cache architecture, DRAM memory and interconnections. In Table III, we specify the details of the simulated system, whose configuration is based on the Intel Sandy Bridge micro-architecture.

We used two parallel benchmark suites for evaluation, each running with 8 threads. Seven applications from the NAS-NPB benchmark suite, with the *A* input size and seven applications from the SPEC-OMP2001 benchmark suite with *ref* input size. Each thread executes 150 million instructions on average. The trace of each application corresponds to a single execution of the application phase. The applications use OpenMP and were compiled with gcc 4.6.3, with the *-O3* option.

B. Evaluated Memory Controller Policies

Given the correlation coefficients presented in Section II, we chose to improve memory controller because memory accesses had the highest correlation with performance. Following the proposals of Ghose et al. [8] and Lee et al. [9], we designed an improved memory controller that can use the profile information provided by BLAP to assign different priorities to memory accesses depending on their relevance for the application's critical path. The baseline for the memory controller policies is the FR-FCFS (First Row - First Come First Serve) [18] policy, which gives priority to row hits (First Row), thus lowering the average memory wait time, and then priority to older accesses (First-Come, First-Serve). In order to compare our solution with the state-of-the-art, we also implemented the original CBP from Ghose et al. [8] and the Prefetch-Aware DRAM Controller (PADC) from Lee et al. [9]. Due to different configurations of the processor, larger cache and lower memory latency, we were not able to achieve improvements as high as those reported by the original papers. We have however observed that, as memory pressure increases, the improvements achieved by these mechanisms and our implementations also increase.

TABLE III. Simulated architectural parameters.

Item	Baseline configuration

Processor cores	8 cores OoO @ 2.66 GHz, 32 nm; in-order front-end and commit;
	16 stages (3-fetch, 3-decode, 3-rename, 2-dispatch, 3-commit stages);
	16 B fetch block size (up to 6 instructions);
	Decode and commit up to 5 instructions;
	Rename/dispatch/execute up to 5 μ instructions;
	18-entry fetch buffer, 28-entry decode buffer;
	3-alu, 1-multiplication and 1-division integer units (1-3-32 cycle);
	1-alu, 1-multiplication and 1-division floating-point units (3-5-10 cycle);
	1-load and 1-store functional units (1-1 cycle);
	MOB entries: 64-read, 36-write; 168-entry ROB;

Branch predictor	1 branch per fetch; 8 parallel in-flight branches;
	4 K-entry 4-way set-assoc., LRU policy BTB;
	Two-Level PAs 2-bit; 16 K-entry BHT;

L1D cache	32 KB, 8-way, 64 B line size; LRU policy; 2-cycle;
	MSHR: 8-request, 10-write-back, 1-prefetch;
	Stride prefetch: 1-degree, 16-strides table;

L1I cache	32 KB, 8-way, 64 B line size; LRU policy; 2-cycle;
	MSHR: 8-request, 1-prefetch;
	Stride prefetch: 1-degree, 16-strides table;

L2 cache	Private 256 KB, 8-way, 64 B line size; LRU policy;
	MSHR: 4-request, 6-write-back, 4-prefetch; 4-cycle;
	Stream prefetch: 4-degree, 64-dist., 64-streams;

L3 cache	Shared 16 MB (8-banks), 2 MB per bank; MOESI coherence protocol;
	16-way, 64 B line size; LRU policy; 6-cycle; Inclusive;
	MSHR: 8-request, 12-write-back; Bi-directional ring interconnection;

DRAM and Bus	On-chip DRAM controller, 8 banks/channel; 4-channels; DDR3 1333 MHz;
	8 burst length; 4 KB row buffer per bank, Open-row first;
	4 core-to-bus frequency ratio; 9-CAS, 9-RP, 9-RCD and 28-RAS cycles;
	MSHR: 128-request, 64-write-back, 32-prefetch;

The *CBP* mechanism gives priority to the load instructions that stall the commit stage. As it only keeps track of the loads, it uses a small 64-bit tagless table per core, which is reset every 100000 cycles to adapt to program phase. It gives priority to all loads found within this internal table as well as any load that stalls the commit stage. The authors explore more options, such as storing the number of stalled cycles for more complex policies, but the results using only 1 bit per table entry proved to have the best trade-off between hardware cost and performance, and we used it in our evaluations.

For the *PADC* mechanism, each cache line is extended by adding 2 bits: a prefetch bit and an access bit. These bits track which prefetches were useful. By measuring prefetch accuracy every 100000 cycles, PADC decides whether it should give the same priority to prefetches and demands, or whether it should prioritize demands and drop prefetches based on the prefetch accuracy. The authors define 4 levels for their architecture. Over 70% prefetch accuracy, the mechanism treats all requests equally and does not drop prefetches. Between 30 % and 70% prefetch accuracy, it prioritizes demand requests and drops prefetches that waited in the memory request buffer for longer than 50000 cycles to be serviced. Between 10% and 30% accuracy, the mechanism drops those prefetches who waited for longer than 300 cycles to be serviced. If accuracy is lower than 10%, any prefetch which waits for more than 100 cycles to be serviced is dropped.

The BLAP-based mechanisms were implemented as follows. *BLAP-CBP* is the adaptation of CBP using the basic block profile information provided by BLAP. We give priority to blocks that BLAP characterized as *Mem*, through the CBP memory controller policy. BLAP-CBP sets the following priority order: 1) Critical row hits; 2) Non-critical row hits; 3) Critical non-row hits, and 4) Non-critical, non-row hits.

In *BLAP-PADC-8L*, generated prefetches get BLAP information from the requests that triggered them. To emulate the concept of PADC, we drop prefetches that waited more than the average demand request wait time. We implemented an 8-level priority memory controller. As we have information of which demand requests are critical, which prefetch requests are critical, and whether the request is a row hit, we need 2^3 levels of priorities. The 8 levels are: 1) Critical demand row hit requests; 2) Critical prefetch row hit requests; 3) Non-critical demand row hit requests; 4) Non-critical prefetch row hit requests; 5) Critical demand requests on another row; 6) Critical prefetch requests on another row; 7) Non-critical demand requests on another row; and 8) Non-critical prefetch requests on another row.

Figure 3 illustrates the request selection logic for different memory controller mechanisms. The mechanism compares the

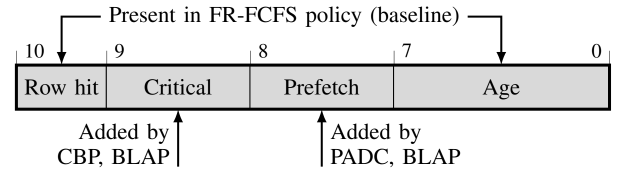


Fig. 3. Request selection logic for different memory controller mechanisms.

information bits from the request as a single number, by concatenating all bits and considering the left-most bits as most significant. The age represents how many cycles the request has been waiting for service in the memory controller request buffer. The prefetch bit is set to 0 on prefetches and 1 on demand requests, to give priority to demand requests. The critical bit is the information fed either by CBP or BLAP. Finally, row hit is 1 if the address of the request matches the currently open row.

In comparison to CBP, the first advantage of BLAP-CBP is that we can exploit other processor bottlenecks beyond memory pressure. The second advantage of this characterization that yields performance gains is that it also provides information on branch mispredictions. We will not give priority to loads that are followed by a mispredicted branch as doing so would not help the block performance. Third, as we can address blocks and store their information using the branch target buffer, we are able to store a much larger amount of information, 4096 entries, compared to 64 entries in CBP's table. Both implementations require the same amount of hardware in the memory controller and channels to pass the information bit that indicates critical requests.

In the evaluated architecture, BLAP-PADC-8L requires four times less storage than PADC by using the BTB to store the profile information.

VI. Experimental Results

Figure 4 shows results for different mechanisms running NAS-NPB and SPEC-OMP2001 benchmarks. In the figure, we show the speedup in terms of execution time for all benchmarks, normalized to the baseline. The first observation is that the average gains of both related work are different from the ones found in their work, due to different benchmarks, architectural parameters and simulators. Although the impact of the mechanism implementation is noticeable, as seen in the IS benchmark, the average benchmark improvement is low.

For this experiment, PADC offers the highest improvement, achieving 19.02% for IS. We have average performance improvements of 1.89% for CBP, 0.80% for BLAP-CBP, 3.10% for PADC and 3.90% for BLAP-PADC-8L. In general, CBP achieves better results than BLAP-CBP. This is due to CBP's information being specifically designed for load instructions, while BLAP profiles at a coarser granularity.

BLAP-PADC-8L outperforms PADC on average as we adapted it to perform in a flexible way, by using the average demand request time. Using BLAP information, the mechanism is able to drop prefetches more aggressively while still servicing important prefetches. This is because the prioritized prefetches come from critical, repetitive blocks. This fact also makes BLAP-PADC-8L more likely to drop false-positive triggered prefetches, as they have a low priority.

In order to stress the memory controller mechanisms and their profiling methods, we used a stream prefetcher with increased aggressivity. A stream prefetcher normally looks for cache misses within a range (search distance), and, if the sequential misses fall within this distance, we allocate a stream. If any cache access falls in the range of m accesses (where *m* is the *prefetch distance* parameter) starting at the stream's initial address, we prefetch *n* cache lines (where *n* is the *prefetch degree* parameter) starting from address *prefetch triggering address + (prefetch distance * cache line size)*, then we set the starting address to the value of the requested address that triggered the prefetches. In our baseline, we used prefetch degree 4 and prefetch distance 64.

Figure 5 shows the results with an increased stream prefetcher aggressivity for CBP, PADC, BLAP-CBP and BLAP-PADC-8L, with prefetch degree 8 and prefetch distance 128. This experiment shows that BLAP-PADC-8L offers the highest improvement, achieving 37.05% for Wupwise. We have average improvements of 3.99% for CBP, 1.72% for BLAP-CBP, 4.24% for PADC and 13.14% for BLAP-PADC-8L.

CBP performed better than in the baseline experiment as it only assigns a higher priority to demand requests. Thus,

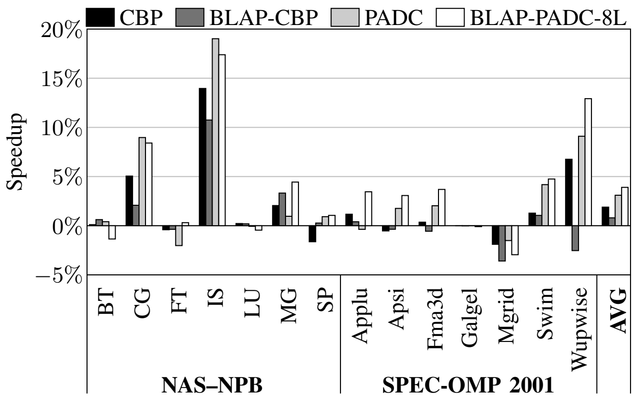


Fig. 4. Performance results for NAS-NPB and SPEC-OMP2001, relative to the FR-FCFS baseline.

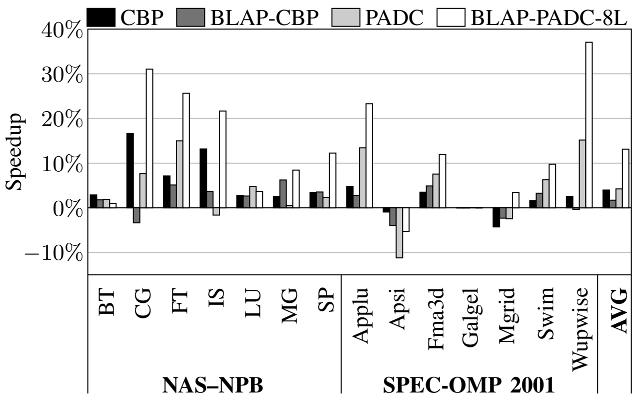


Fig. 5. Performance results for NAS-NPB and SPEC-OMP2001 with increased aggressivity prefetcher, relative to the FR-FCFS baseline.

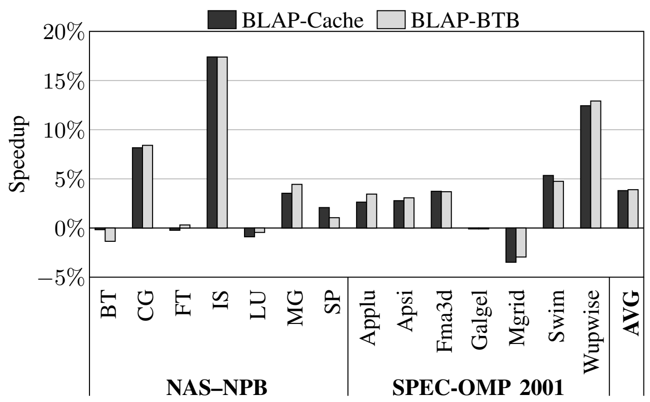


Fig. 6. Performance results comparison between BTB and a large cache, relative to the FR-FCFS baseline.

it improves performance by only servicing prefetches when there are no critical demand requests. For the reasons mentioned for the previous experiment, BLAP-CBP also improves performance, but does not reach the same level as CBP.

PADC achieves the same performance improvements as in the baseline. This happens because our evaluations used the same parameters proposed by the authors, although different system architectures may require different internal parameters. For this reason, PADC is not able to drop prefetches as aggressively as needed. On the other hand, BLAP-PADC-8L achieved high performance improvements for several benchmarks due to its highly aggressive prefetch dropping.

Figure 6 compares two implementations of BLAP-PADC-8L. The first implements BLAP using the BTB while the second uses a cache which is large enough to avoid any conflict and capacity misses in all benchmarks. Moreover, it is able to differentiate and store blocks targeted by branches as well as fall-through blocks.

Comparing the BTB and the large cache implementations, we can notice similar performance improvements over the baseline. It shows that the large number of entries in the BTB is sufficient to keep the profile information for most of the benchmarks.

VII. Conclusions

Our results show that basic block granularity can be as relevant as single instruction granularity for memory accesses. The findings indicate that as basic blocks naturally track a program's phase progression, we are able to more accurately adapt to different memory pressures that occur in different program phases. On average, we were able to improve performance by 3.9% compared to the baseline FR-FCFS, with a low hardware overhead. We have also shown that our technique scales better than the state-of-the-art when faced with more aggressive prefetch policies that result in a higher memory pressure.

In the future, we intend to characterize blocks regarding more events, such as data-dependent branches [19]. The idea of basic block detection at the commit stage can also be overlapped with group commit [20], and can enable the implementation of several ideas based on basic block analysis.

Acknowledgment

This work was partially supported by CNPq and Capes.

References

[1] J. Reinders, *VTune performance analyzer essentials*. Intel Press, 2005.

[2] J. Cocke, "Global common subexpression elimination," *SIGPLAN Not.*, vol. 5, no. 7, Jul. 1970.

[3] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: collection and analysis of parallel block vectors," in *Int. Symp. on Computer Architecture (ISCA)*, 2012.

[4] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, "A fully integrated multi-cpu, gpu and memory controller 32nm processor," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2011.

[5] V.-M. Panait, A. Sasturkar, and W.-F. Wong, "Static identification of delinquent loads," in *Code Generation and Optimization (CGO)*, 2004.

[6] P. Ratanaworabhan and M. Burtscher, "Program phase detection based on critical basic block transitions," in *Int. Symp. on Performance Analysis of Systems and software (ISPASS)*, 2008.

[7] S. J. Patel and S. S. Lumetta, "replay: A hardware framework for dynamic optimization," *Trans. on Computers*, vol. 50, no. 6, 2001.

[8] S. Ghose, H. Lee, and J. F. Martínez, "Improving memory scheduling via processor-side load criticality information," in *Int. Symp. on Computer Architecture (ISCA)*, 2013.

[9] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware dram controllers," in *Int. Symp. on Microarchitecture (MICRO)*, 2008.

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

[11] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[12] 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, 2005.

[13] 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 *Int. Symp. on Microarchitecture (MICRO)*, 2004.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*, 2005.

[15] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, 1998.

[16] M. L. Pilla, P. O. A. Navaux, B. R. Childers, A. T. da Costa, and F. M. G. Franca, "Value predictors for reuse through speculation on traces," in *Int. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2004.

[17] M. Alves, "Increasing energy efficiency of processor caches via line usage predictors," Ph.D. dissertation, Universidade Federal do Rio Grande do Sul, May 2014.

[18] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory access scheduling," in *Int. Symp. on Computer Architecture (ISCA)*, 2000.

[19] M. U. Farooq, K. Khubaib, and L. K. John, "Store-load-branch (slb) predictor: A compiler assisted branch prediction for data dependent branches," in *Int. Symp. on High Performance Computer Architecture (HPCA)*, 2013.

[20] F. Afram, H. Zeng, and K. Ghose, "A group-commit mechanism for rob-based processors implementing the x86 isa," in *Int. Symp. on High Performance Computer Architecture (HPCA)*, 2013.