Profiling and Optimizing Micro-Architecture Bottlenecks on 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
E-mail: {fbmoreira, mazalves, mdiener, navaux}@inf.ufrgs.br

Israel Koren
Dept. of Electrical and Computer Engineering
University of Massachusetts at Amherst
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 provide more opportunities for mechanisms that use this information. For example, it is possible to integrate information of branch prediction and memory accesses to provide precise information for speculative mechanisms, increasing accuracy and performance.**

We propose BLAP, an online mechanism that profiles bottlenecks on the micro-architectural level, such as delinquent memory loads, hard-to-predict branches and contention for functional units. BLAP works on the basic block level, providing information that can be used to optimize these bottlenecks. A prefetch dropping mechanism and a memory controller policy were created to use the profiled information provided by BLAP. Together, these mechanisms are able to improve performance by up to 17.39% (3.9% on average). Our technique showed average gains of 13.14% when evaluated with higher memory pressure due to higher prefetch aggressivity.

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. Thus, a program's execution path is defined by basic block execution sequence due to these branches, which enables a general 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 on 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 on hardware level was the rePlay framework [7]. It analyses 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 analysis available. 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 propose a Block-Level Architecture Profiler (BLAP). It 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 automatically 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 also capable of detecting different types of performance 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 explored 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's profile provided better execution performance with scalable hardware overhead. Moreover, BLAP's basic implementation can be extended to provide detailed information regarding a wide range of bottlenecks at efficient 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 relevance 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 on 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 mechanisms that improve memory performance, by adapting them to use the profile information or by creating a new mechanism that used their concept based on BLAP.

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

II. Motivation

In this section, we will explore the relationship between blocks and performance. We actually use a relaxed definition of a basic block [10], [2], [11]. 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 that begin after a taken branch. Given that the behavior to be exploited is usually repetitive, this is normally not a problem, as code layout of loops will likely make repetitive blocks begin after taken branches. Another issue is that we cannot recognize branch targets unless their respective branch occurred. This breaks the definition of 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. But the smaller basic blocks will be correctly characterized once they are branched to, 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. This is a generalization of the linear regression model, and it is used to observe how closely two different sets of data correlate. 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 closer to 0, the smaller is the correlation between parameters.

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 amount of floating point arithmetic-logic instructions and division instructions. Whenever a basic block finished executing, we recorded the amount 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 for that block. For each parallel application from NAS-NPB and SPEC-OMP2001 suites, each correlation coefficient was calculated considering blocks from all the threads together.

TABLE I. Pearson moment-product correlation coefficients of absolute statistics per block and performance in IPC.

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

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

Although a misprediction in a 15-stage pipeline will result in flush latency and lack of instructions, the other instruction types correlation seem to render the branch instruction correlation useless. This low correlation coefficient is because the low branch misprediction rate of less than 1% in the benchmarks. Floating points instructions per block correlate well on a few benchmarks. We can observe that for Apsi and Mgrid, floating point ALU instruction count is the statistic that correlates the most with degraded performance.

Following this analysis, we seek to improve the memory access bottleneck. However, obtaining detailed hardware counter statistics per block during execution is a complex matter. As we aim to aggregate behavior and singularly identify blocks, for a reduced storage, using statistics gives us three challenges. First, a statistic must show a direct impact on the performance. While cache misses are intuitively correct in expressing delinquent loads [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 a floating point division unit, as we do not know whether there is any instruction that actually depends on the unit 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 instruction 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 making accesses to the data cache or committing floating point instructions might be altering these statistics, preventing us to accurately evaluate 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 technique will obtain information that directly impacts performance (first issue), since we are looking the commit stage stalls. We can directly compare commit stall 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 stall of the instructions at the head of the Reorder Buffer (ROB), and detecting branch instructions to observe block boundaries. It is also required to effectively storage this profile. Therefore, the information for each block should be kept to a minimum. Additionally, the size of the information has an impact as we need to communicate the profile to different mechanisms. Finally, the mechanism should be able to provide varied characterization, so multiple mechanisms can use the profile provided. Accordingly to the correlation results, we chose to record the following characteristics: *None* to denote that the block presents no problems, *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.

III. Related Work

Sherwood et al. [12] is the precursor to SimPoints [13] and other works, such as Pinpoints [14]. The authors characterize the behavior of entire programs based on the analysis of basic block execution distribution. The concept of Basic Block Vector (BBV) is first introduced to characterize a program. A basic block vector is a simple vector containing execution count 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. This way the authors are able to compare the behavior for executions of different sizes, for different inputs. Next, a BBV comparison method is created, by treating each BBV as a fingerprint of the program slice observed. To generate this difference, a BBV is subtracted from another, and all absolute values are summed, generating a value between zero and two. Zero means the BBVs are identical, as there was no difference between their fingerprints, while two means the BBVs don't execute any block in common.

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 great 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 amount of instructions. They show that the behavior of selected program slices with near-identical BBV are similar, with statistics pertaining cache misses, branch mispredictions and overall type of instructions executed differing at maximum by 3%. This was further improved on SimPoints, and Pinpoints is basically a tool to use Pin [15], an instrumentation tool from Intel, 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 trace called a *frame*. A frame aggregates several basic blocks, as it ignores unconditional branches, and promotes easy to predict branches into assertions [16]. They also provide a scheme to replay a frame in case an assertion fires, which signals a misprediction. 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 [17]. Although the framework is proposed, 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 online optimizations using the frames collected. Currently, improving performance of the code itself is hard, as compiler technology is quite evolved. Because of this work, we changed our approach to try and characterize our fine-granularity blocks in a simple, so we can better inform other mechanisms, instead of aiming to provide complex analysis.

The recent work of Kambadur et al [3] also uses a simple profiling method they develop called Parallel Basic Block Vectors. It can be seen as an extension of Sherwood's work, as now each 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 with which frequency at which parallelism levels, clearly identifying sequential and parallel code blocks. They also identify the most critical regions of code in terms of performance, which one would hope to be executed at high thread counts. Several scenarios are then illustrated, showing where this analysis can be applied, such as in serial and parallel application partitions, or analysis of program features by degree of parallelism and parallelism hotspot.

The Criticality Binary Prediction (CBP) mechanism was actually the base for our mechanism. It observes how many cycles each load instruction stalls the commit stage and gives priority to the loads that do. As it only keeps track of the loads, it uses a small 64 bits tagless SRAM table per core, which is reset every 100000 cycles to adapt to program phase. It then gives priority to the loads found in the table. The actual paper explores more options, such as storing the number of stalled cycles for more complex policies, but overall using only 1 bit per table entry and for all structures proved to be the best trade-off, which is used in this paper. Our idea is a generalization of [8], used in a novelty way.

Following the work of rePlay and Ghose et al. [8], we take a new direction, by using the behavior detected to improve on existing hardware mechanisms that need to detect phase changes and bottlenecks online, namely Ghose's criticality binary prediction and Lee et al. [9] prefetch-aware DRAM controller. We intend to implement these works with our basic block framework, aiming to improve their performance or adapt them in a radically new way.

IV. Block Level Architectural Profiler

In this Section, we present the detailed implementation of Block-Level Architecture Profiler (BLAP) in hardware. We divide the implementation in three parts: Behavior Detection, Behavior Labeling and Behavior Storage. Afterwards, we explain potential critical path implications and how they were avoided. We then provide the hardware overhead cost calculation for these three stages and additional optimizations to improve profile information.

A. *Behavior Detection*

With an in-order commit stage, instructions that take long to commit may stall the whole processor, according to the latency and the ILP of the application. We consider that the instructions which stall the commit stage the longer in a block are the instructions that characterize a block's performance issues. We propose BLAP, which modifies the commit stage to obtain profile information on basic block level. Figure 1 shows an abstraction of the BLAP implementation inside an Out-of-Order (OoO) processor.

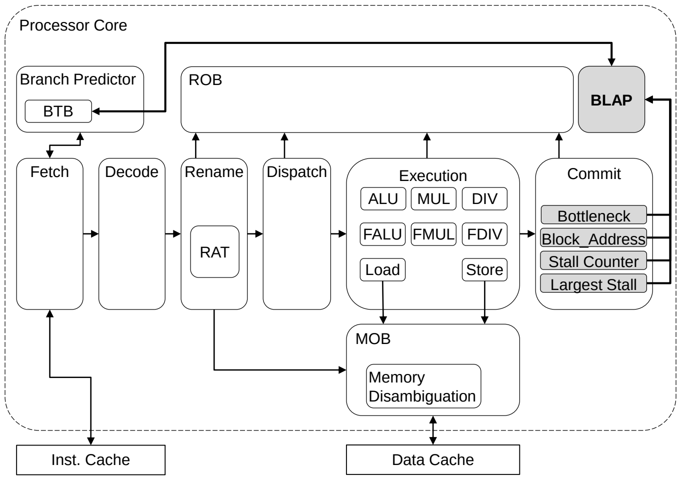

Fig. 1. BLAP implementation abstraction in a superscalar processor.

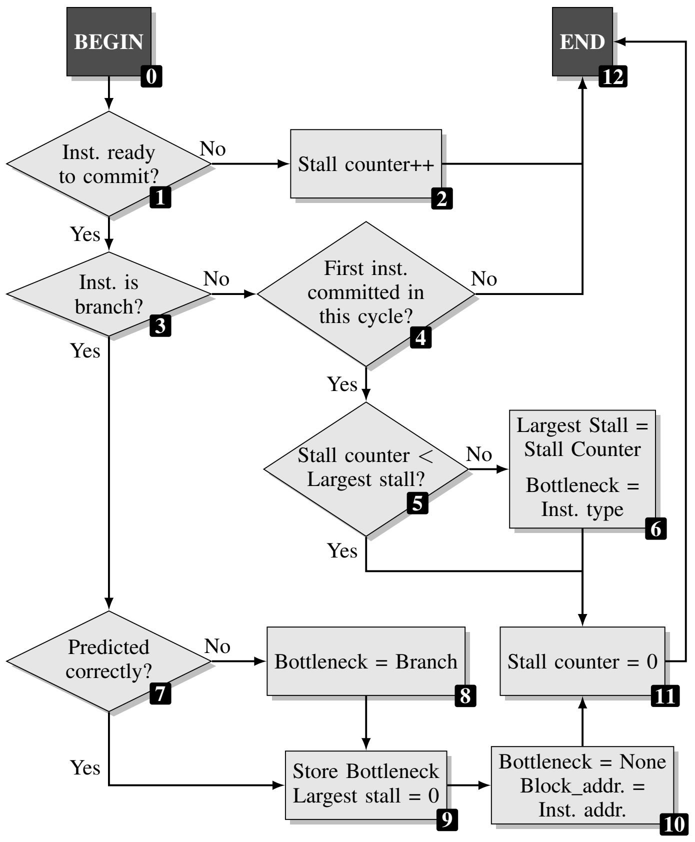

Fig. 2. Flow chart of commit stage modifications.

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 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 older instruction is ready to commit $\boxed{1}$. Whenever an instruction stalls the commit, counter *Stall Counter* is incremented $\boxed{2}$. In the case the instruction is ready to be committed, we must check if it is a branch $\boxed{3}$, as branches indicate the end of blocks. 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 it's 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 $\boxed{11}$.

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 directly access it's accuracy information. 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 storage used, 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) $\boxed{10}$. This way, in 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 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 such as when a block has an unstable characteristic in the beginning. It uses a saturated counter to record the amount of times the basic block was executed. When this counter saturates, the last detected characteristic is considered to be stable, and thus it is used as bottleneck for that block. Further changes in the block bottleneck will not overwrite the BTB entry, in order to avoid disabling the improvements that lead to the bottleneck change.

B. Behavior Storage

In order to use the profile, we must store it for future use. Accordingly to the correlation results, we use 2 bits per block, expressing 4 characteristics (*None*, *Brch*, *Mem*, *FP*). The number of characteristics can be incremented by using more bits per entry, for future extensions. As the BTB contains all the conditional and unconditional branch targets, instead of using a new cache, we can extend the BTB to store characteristics for each block targeted by each branch. That way, if a branch is taken, we can load the characteristic in its entry as we know 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 instruction 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 the block.

D. Critical Path Implications

Detection: The detection scheme of BLAP implies additional hardware to update the mechanism's registers. There is a corner case which occurs when two branches commit in the same cycle. This means 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 processor's performance, as it is not in the critical path.

Storage: 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: Labeling has no implications on critical path, requiring only additional information bits going through the pipeline along their respective instructions.

E. Hardware Costs and Design Considerations

To implement BLAP, we can divide the hardware costs into three parts: detection, storage and labeling. As can be seen in Figure 2, detection requires two 8 bit counters, Stall and Largest Stall. It also requires an 8 bit adder and a 8 bit comparison 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 prediction was a misprediction, we add 1 bit for each reorder buffer entry. For storage, we require an additional BTB write port to write the extra BTB bits. We reuse all tags and logic from the BTB. The extension is composed of 2 bits for characteristics and 2 bits saturating counter to stabilize each entry.

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 from the fetch buffer, decode buffer and ROB. The calculated costs in terms of hardware are shown in Table II. For each core, BLAP requires the total storage of 2142 bytes, plus 2 bit multiplexer, 8 bit adder and 8 bit comparison.

V. Evaluation Methodology

A. Simulation Environment and Workloads

To validate our mechanism, we used a cycle-accurate in-house simulator. It accurately simulates the microarchitecture modeling all functional unit contention, register dependency, processor system restrictions, besides 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 microarchitecture. We used two parallel workload suites for evaluation, running with 8 threads. 7 applications from the NAS-NPB workload suite, with the *A* input size. 7 applications from the the SPEC-OMP2001 workload suite with *ref* input size. Each thread executes 150 million instructions on average in the trace. The

TABLE II. HARDWARE COSTS OF BLAP

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); 2-bit Bottleneck reg. (BLAP);
	64-bit Block Addr. reg. (Commit); 64-bit Block Addr. reg. (BLAP);
	1-bit branch prediction information per ROB entry (168 entries in total);
	2-bit multiplexer, uses branch prediction information to update BLAP;
	Total of 316 bits for detection;

Storage	2-bit Bottleneck reg. per BTB entry (4096 entries in total);
	2-bit Stabilizer counter per BTB entry (4096 entries in total);
	1 additional BTB write port;
	Total of 2048 bytes of storage;

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

overall trace of each application represents one parallel time step from each algorithm. The applications uses OpenMP and were compiled with gcc 4.6.3, with the *-O3* options.

B. Evaluated Memory Controller Policies

Given the correlation coefficients presented in Section II, we chose to improve memory controller because of the high correlation the memory accesses have 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 give 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 [18] policy, which gives priority to row hits (First Row), thus lowering average memory wait time, and then priority to older accesses (First-Come, First-Serve). In order to compare our solutions 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].

TABLE III. SIMULATED ARCHITECTURAL PARAMETERS.

	Baseline Configuration

Processor Cores	8 cores OoO @ 2.0 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 Pred.	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;

L1-D 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;

L1-I 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; 8 KB row buffer per bank, Open-row first;
	1.5 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 bits tagless SRAM table per core, which is reset every 100000 cycles to adapt to program phase. It then gives priority to the loads found in this internal table. 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, which is used 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 for prefetches and demands, or whether it should prioritize demands and drop prefetches given the pollution present by using 4 values defined by the authors 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%, drop any prefetch which waits for more than 100 cycles to be serviced.

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

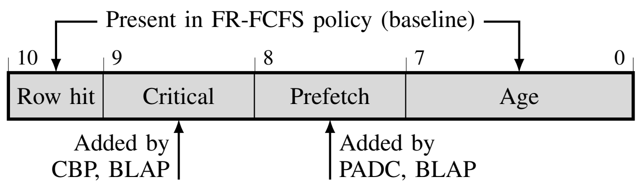

Fig. 3. Request selection logic for different memory controller mechanisms.

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 which makes us gain performance is that as we also have information of branch mispredictions. We will not give priority to loads that are followed by a mispredicted branch. Doing so would not help the block performance, which is why branch prediction is given the highest value. 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 extra criticality information bit.

Compared to PADC, BLAP-PADC-8L required four times less storage by using the BTB to store the profile information.

VI. Experimental Results

Figure 4 presents results for different mechanisms running NAS-NPB and SPEC-OMP2001 benchmarks. In the Figure, we show total execution time for all benchmarks, normalized to the baseline. The first observation that must be made 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 effect of the mechanism implementation is noticeable, as seen in the IS benchmark, the benchmark average results 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 obtained better results than BLAP-CBP. This is due to CBP information being specifically designed for load instructions, while BLAP profiles in 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 inclinable to drop false-positive triggered prefetches, as they are not prioritized and left to be dropped.

In order to stress the memory controller mechanisms and their profiling methods, Figure 5 shows results with an increased stream prefetcher agressivity 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 perfomed better than in the baseline experiment as it only gives priority to demand requests. Thus, improving performance by only servicing prefetches when there are no critical demand requests. For the reasons mentioned for the previous test, BLAP-CBP also improves, but not reaching the same level as CBP.

PADC obtains 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. This way, 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 aggressiveness on prefetch dropping.

Figure 6 shows results for BLAP-PADC-8L, comparing the BLAP mechanism implemented with the BTB and implemented with a large cache, which is large enough to avoid any conflict and capacity misses in all benchmarks. Moreover, it is

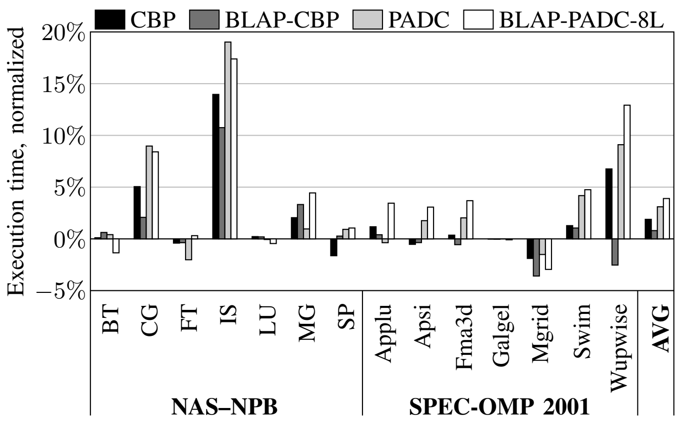

Fig. 4. Performance results for NAS-NPB and SPEC-OMP2001, relative to FR-FCFS baseline.

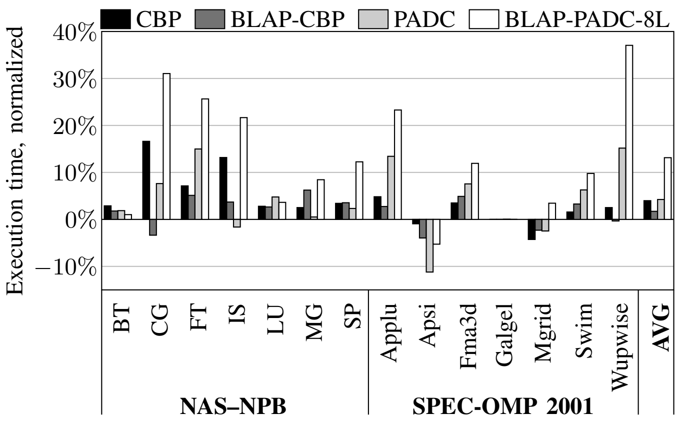

Fig. 5. Performance results for NAS-NPB and SPEC-OMP2001 with increased aggressivity prefetcher, relative to FR-FCFS baseline.

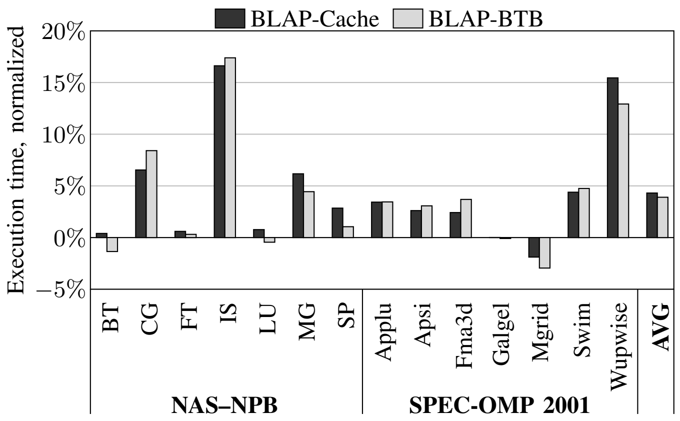

Fig. 6. Performance Results comparison between BTB and a large cache, relative to FR-FCFS baseline.

able to differentiate and store blocks targeted by branches as well as fall-through blocks.

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

VII. Conclusions

Our results show that basic block granularity can be more relevant than 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 higher memory pressure due to higher prefetch aggressivity.

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

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] D. Ansaloni, S. Kell, Y. Zheng, L. Bulej, W. Binder, and P. Tůma, "Enabling modularity and re-use in dynamic program analysis tools for the java virtual machine," in *ECOOP 2013 – Object-Oriented Programming*, 2013, vol. 7920.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[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.