A Dynamic Block-Level Execution Profiler

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 at Amherst, USA
E-mail: koren@ecs.umass.edu

Abstract

Most performance enhancing mechanisms in current processors, such as branch predictors or prefetchers, rely on program characteristics monitored at the granularity of single instructions. However, many of these characteristics can be obtained at the basic block-level instead. The coarser granularity allows a larger portion of the code to be examined, enabling a more accurate profiling and a detailed analysis of the different types of instructions executed within a block. Therefore, block-level analysis can be advantageous for performance enhancing mechanisms, as it allows us to look at how the instructions influence each other, and thus detect complex behavior patterns.

In this paper, we present the Dynamic Block-Level Execution Profiler (DBLEP), a basic block level online mechanism that profiles micro-architectural bottlenecks, such as delinquent memory loads, hard-to-predict branches and contention for functional units. DBLEP operates at the basic block level and provides information that can be used to reduce the impact of these bottlenecks. A prefetch dropping scheme and a memory controller policy were developed to use the code profiling information provided by DBLEP. By taking advantage of the high profiling accuracy, these mechanisms are able to improve the processor's performance by up to 18.6% (5.3% on average). We show that our mechanism's performance is comparable to mechanisms that work on single instruction granularity, using less hardware.

1. Introduction

Characterization of basic blocks is often used by various optimization techniques such as branch biasing [1] and value reuse [2]. The basic block granularity is especially useful as basic blocks represent portions of code that always end with conditional or unconditional branch instructions [3]. Thereby, a program's execution path is defined by sequences of basic block executions, enabling program phase characterization and dynamic optimizations. A recent example is the work of Kambadur *et al.* [4], which uses basic blocks to characterize the thread-level parallelism of an application in its different phases.

Current general-purpose processor designs only collect information at the instruction level. Although several research papers used basic block analysis, most performed it in software, even for hardware adaptations [5, 6]. One of the few techniques that actually performed basic block analysis at the hardware level is the rePlay framework [7]. It analyzes the executing code to perform online code optimization and stores the collected information in a trace cache for future use. However, no bottleneck profiling was performed. Block profiling is usually done in software due to the high complexity of detailed profiling and the required analysis. Still, profiling in hardware is worth investigating, as it can efficiently generate relevant information regarding the program's execution without pre-analysis or source code modification.

In this paper, we present our Dynamic Block-Level Execution Profiler (DBLEP). DBLEP is a general framework to characterize basic blocks according to the most relevant stalls occurring during the execution of the block, thus allowing improvement of the block's future executions. DBLEP has several advantages over other mechanisms. It adapts to program phase changes, as it dynamically keeps track of each basic block behavior. In addition, it requires less storage than using several instruction-granularity mechanisms, as it can aggregate the general behavior per block. DBLEP is capable of detecting different types of performance-related issues within a block, allowing it to provide information to a wide range of mechanisms, such as memory controllers or branch predictors. In this work, we focus on information gathered from stalls in the commit stage, but other data can be obtained in order to implement different mechanisms.

This paper is an extended version of our previous work which described our Block-Level Architecture Profiler (BLAP) [8]. BLAP showed the potential of basic block-level online profiling by using the profile to modify different memory controller designs. In this paper, we introduce an improved profiler DBLEP, and analyze the dynamic behavior during the execution of the application and the mechanism sensitivity to its detection methodology and accuracy. Furthermore, we study the the reasons behind DBLEP's performance improvements.
The main contributions of this paper are as follows:
Characterization Mechanism: We introduce DBLEP, an efficient mechanism to characterize applications at the basic block level during their execution.
Accuracy Enhancement: The accuracy of the proposed mechanism is further increased by using finite-state machines for the information collected, allowing unstable blocks to be re-characterized. This in turn increases the performance gains of the memory controller that relies on DBLEP's profiling.
Performance Analysis: DBLEP's performance improvements are analyzed, showing that it achieves reduced demand-based service time in the main memory, leading to overall fewer stall cycles while the processor waits for memory requests.

2. Correlating Microarchitectural Bottlenecks to Performance

In this section we explore the relationship between basic blocks and processor performance. We use here a relaxed definition of a basic block [2, 3]. A basic block is a code segment with a single point of entry and a single point of exit. Thus, every basic block ends either with a branch instruction, or an instruction that is the target of another branch instruction and therefore indicates the beginning of a new basic block. This enables mechanisms based on basic blocks to identify program phases automatically, as a program's phase is characterized by the blocks being executed [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. Moreover, since we use the Branch Target Buffer (BTB) to store the block's behavior (as explained in Section 4), we collect information only for blocks that are executed following a taken branch. The implications of this implementation are further discussed in Section 4.

To analyze the block behavior that can be monitored using our relaxed block definition and its correlation with performance, we calculated the Pearson Moment-Product correlation coefficients between execution events within a block (such as branch mispredictions) and the processor performance. We chose this correlation since it is invariant to the scale of the values used (which greatly varies between different levels of cache), unlike simple linear regression models [9].

2.1. Complete Execution Correlation

Correlation coefficients have a value between -1 and 1. The higher the absolute value, the stronger the correlation between the parameters. If the coefficient is negative, the parameters are inversely correlated (that is, when one increases, the other one decreases), while if it is positive, both values increase or decrease together. As an example, consider a correlation value of -0.95 between the number of integer multiply instructions and the Micro-ops Per Cycle (UopPC) (See IS benchmark in Table 1). Such a value means that, whenever the number of integer multiplications increases, the UopPC value will closely follow the change and will be lower, due to the inverse correlation. If instead, the correlation coefficient is 0.9, the UopPC value would be higher. A small value of -0.13 would indicate that the values are almost independent of each other.

The details of the configuration and benchmarks used can be found in Section 5. To calculate the correlation coefficients, we generated a trace of the execution. This trace contained the most important processor events relevant to execution performance measured in UopPCs: including L1 data cache (L1D) misses, L2 cache misses, Last-Level Cache (LLC) misses, branch mispredictions, and the number of instructions of a certain type, e.g., integer (INT), floating-point (FP), arithmetic-logic (ALU), multiply (MUL) and divide (DIV). Whenever a basic block finishes executing, we record the number of instructions the block has committed (therefore, the number of micro-ops), and the number of cycles it took to execute, in order to measure its performance. We then record how many of the above listed events happened during the execution of that block. For each parallel application from the NAS-NPB and SPEC-OMP2001 benchmark suites, we first created a list for each characteristic of the most frequently executed blocks which showed a large number of stalls for that characteristic, and calculated the correlation coefficient for each event considering these blocks occurrences in all threads.

The correlation results are summarized in Table 1. The highest correlation coefficients for each benchmark are marked in bold. We must note that the implications of the obtained correlation values are not always intuitively clear. Although a miss in

Table 1: Pearson moment-product correlation coefficients between events and performance (measured as microops per cycle) detected during basic block execution.

	Benchmark	INT			FP			L1D	L2	LLC	Branch
		ALU	MUL	DIV	ALU	MUL	DIV	misses	misses	misses	mispred.
NAS-NPB	BT	– 0.44	**– 0.75**	0.06	0.11	– 0.23	– 0.38	– 0.50	– 0.24	– 0.33	0.00
	CG	0.17	– 0.64	– 0.64	0.16	0.16	– 0.40	**– 0.69**	– 0.48	– 0.51	0.01
	FT	– 0.02	0.25	– 0.04	– 0.04	0.35	0.00	**– 0.45**	– 0.31	– 0.27	– 0.04
	IS	0.61	**– 0.95**	– 0.13	– 0.13	0.00	0.00	– 0.17	– 0.16	– 0.16	– 0.01
	LU	0.42	– 0.72	**– 0.91**	– 0.02	0.18	– 0.49	0.06	0.04	– 0.16	– 0.02
	MG	– 0.04	**– 0.42**	0.00	0.11	0.24	0.00	– 0.37	– 0.30	– 0.29	– 0.02
	SP	– 0.34	0.00	0.18	0.06	– 0.09	– 0.17	– 0.48	– 0.45	**– 0.50**	– 0.01
SPEC–OMP2001	Applu	0.29	0.00	– 0.36	0.27	0.00	– 0.02	**– 0.49**	– 0.48	– 0.41	– 0.01
	Apsi	**0.46**	0.00	0.04	– 0.09	– 0.09	– 0.07	– 0.11	– 0.15	– 0.15	– 0.09
	Fma3d	– 0.35	0.00	– 0.16	– 0.27	0.07	– 0.29	– 0.49	**– 0.57**	**– 0.57**	0.00
	Galgel	– 0.06	0.00	**– 0.30**	– 0.04	– 0.07	– 0.02	– 0.24	– 0.26	– 0.07	0.00
	Mgrid	– 0.27	0.00	**– 0.63**	– 0.60	– 0.58	0.00	– 0.30	– 0.30	– 0.29	– 0.01
	Swim	– 0.38	0.00	– 0.11	– 0.41	– 0.35	– 0.28	**– 0.70**	– 0.61	– 0.58	0.00
	Wupwise	0.06	0.00	0.04	0.14	0.00	– 0.16	**– 0.60**	– 0.51	– 0.48	0.00

the LLC means a main memory access, which will stall the processor, the number of requests that 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 strongly correlate with the performance differences between blocks in some benchmarks.

Analyzing the arithmetic-logic functional units we observe two different behaviors. Some benchmarks have a positive correlation value for certain arithmetic units, which implies that, in these benchmarks, the number of executed instructions in these arithmetic units positively correlates with higher performance. See, for example, the coefficients for INT ALU in *IS* and *Apsi*. In contrast, for the *BT* benchmark, the INT ALU, although it has only a single cycle latency, correlates negatively with the performance due to a long chain of dependent INT ALU instructions. The integer multiplication and division units have longer latencies and may aggravate the problem and result in a stronger negative correlation with performance. As an example, consider the coefficients for INT MUL and INT DIV for the *CG* and *LU* benchmarks. A similar observation can be made for floating-point units.

Although a branch misprediction in a 15-stage pipeline results in 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 (less than 1%) in the benchmarks.

2.2. Dynamic Execution Correlation

The analysis summarized in Table 1 points out the correlation coefficients for the most frequent blocks of each benchmark. However, applications change their behavior and exhibit dynamic changes in the observed correlation during their execution. To study the temporal behavior, we calculated the correlation of all blocks executed within intervals of 100,000 completed micro-ops.

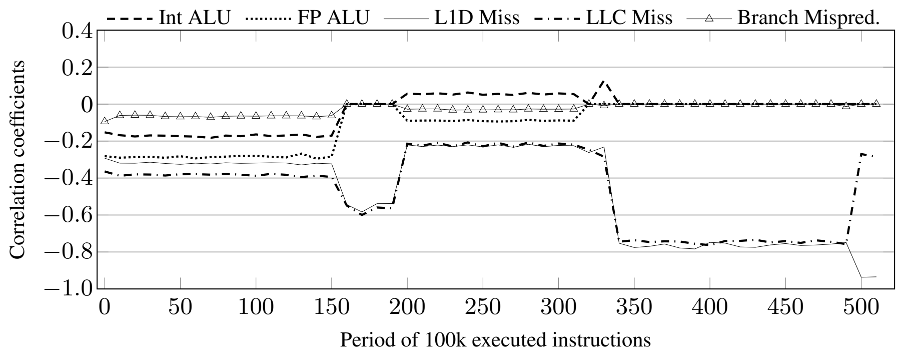

Figure 1: Dynamic behavior of Fma3d from SPEC-OMP2001.

Figure 1 presents the correlation values for five events, for every interval of 100,000 micro-ops during the execution of the *fma3d* benchmark. Not all the events that were listed in Table 1 were included in the figure for better clarity. This benchmark exhibits dynamic behavior with several stable phases. Cache misses have the largest (negative) correlation with degraded performance throughout the benchmark. In the first 150 intervals branch mispredictions have a small (but still larger than later on) correlation with degraded performance, likely due to the initialization of the branch prediction unit. There is also a limited correlation between degraded performance and the number of INT and FP ALU instructions. These correlations become negligible for the rest of the benchmark's execution. After the 500th interval there is a large disparity between the correlations of the LLC and L1D cache misses.

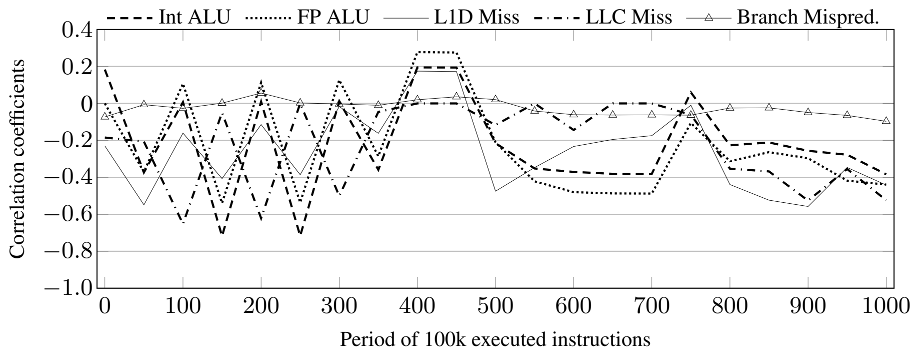

Figure 2: Dynamic behavior of LU from NAS-NPB.

The *LU* benchmark exhibits a different behavior, as shown in Figure 2. No clearly identifiable stable phases exist but we still observe that there are alternating time periods; in some of which the INT ALU has the highest correlation with degraded performance, while in other, either FP ALU or L1D misses show the largest correlation with performance. It is also interesting to note that in some periods the correlation coefficient between L1D cache misses reaches -0.56, while a positive correlation value of 0.06 was observed for the most frequent blocks in Table 1. The most important conclusion that can be drawn from these two figures is that applications exhibit a dynamically changing behavior during their execution and consequently, we should not determine statically which event has the highest negative impact on performance but instead, monitor in real time the application behavior to determine the event that currently has the largest impact on the processor performance.

2.3. Mechanism Design Options

Based on this study that has shown that memory accesses often have a high correlation with processor performance, we decided to first focus on reducing the memory access bottleneck. However, collecting the relevant statistics per block during execution, as well as aggregating a block behavior and identifying the event with the largest impact on performance is a non-trivial matter. There are three main challenges that are listed below.

First, the collected statistics have to 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 attempting to determine the one with the largest impact. Assuming a hit-under-miss policy with a MSHR implementation, 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 determine the bottleneck with the largest impact we cannot compare the observed value directly to the latency of, for example, the floating-point division unit, as we do not know whether there is any instruction that depends on the division result, or if it will eventually stall the commit stage.

Third, using hardware counters to profile a block is not simple as instructions from a consecutive block may alter the counts. When the last instruction of the monitored block has completed its execution and is ready to commit, we already have collected the required statistics and we should then reset the counters in order to gather statistics for the next block. However, instructions from the next block may have already executed and modified the counters, thus preventing us from accurately profiling each block.

To overcome these challenges, we analyze blocks in the commit stage. Instructions only cause bottlenecks in the pipeline if they cause the commit stage to stall. Thus, in order to compare the delays of different instructions we only look at how many cycles each micro-op of that 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 micro-ops, 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 hardware mechanism that can identify bottlenecks based on monitoring the commit stage stalls may prove to be beneficial. It has to be able to meaningfully characterize blocks, efficiently store the block profile and require an overall small logic overhead in the pipeline. As can be seen in the previous subsection, profiling during execution has the advantage of adapting to the dynamically changing behavior of applications.

Finally, the mechanism should be able to provide varied characterizations, so that multiple performance enhancement techniques can use the generated profile. Based on the correlation results and latencies associated with the functional units, we chose to record the following characteristics: *(1) None* to denote that the block has negligible stalls, *(2) Brch* to denote hard-to-predict branches, *(3) Mem* to denote commit stalls due to loads and *(4) FU* to denote commit stalls due to any functional unit. The number of options is kept small as we focus on differentiating blocks stalled by memory from the others and we want to ensure a small hardware overhead. 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 due to space constraints.

3. Related Work

Many mechanisms and tools have used the basic block granularity to characterize program phases [4, 7, 10, 11, 12], but most of them only perform static profiling. Our work seeks to study profiling with hardware-level information during execution. In this aspect, our work has conceptual similarities with Clarke *et al.* [13], whose work relies on hardware-level block detection to perform dynamic instruction translation in hardware. Their framework, *Liquid SIMD*, uses compiler-assisted generation of baseline code that contains only scalar instructions, and hardware support to implement new SIMD instructions. The technique consists of detecting basic blocks which are composed of scalar instruction sequences or loops, and then performing dynamic instruction translation generating new SIMD instructions to replace those blocks. Thereby, compilers do not need to be changed to adapt to the ever increasing number of new instructions in ISAs. The SIMD accelerator is effectively decoupled from the ISA, allowing instruction sets to seamlessly support multiple SIMD accelerator designs with no change to the instruction set, and thus support full compatibility. The detection concept is similar to our own, but we focus on characterization that can aid other mechanisms to improve system performance.

In order to use the profiled information, we have implemented several known techniques to lower the performance degradation due to memory access. One of these techniques also provides the basis for our mechanism. The Criticality Binary Prediction (CBP) scheme [14] observes by how many cycles each load instruction stalls the commit stage and assigns higher priority to the loads that stall. As it only keeps track of loads, it uses a small 64-bit tagless table per core, which is reset every $100,000$ cycles to adapt to the current program phase. The paper explores additional options, such as storing the number of stall cycles for more complex policies, but concludes that the base mechanism, which uses only 1 bit per table entry and all memory request buffer entries, offers the most efficient implementation. Our work, on the other hand, generalizes the idea of commit stalls monitoring, using it to characterize entire blocks.

Further, since we compare different instruction stalls in the same block, we indirectly infer which instructions are more important in each block and should be prioritized.

As discussed in detail in [8], previous techniques use block granularity profiling for value reuse, cache resizing and general profiling. Our paper is the first to use the detected block behavior to improve other hardware mechanisms that attempt to detect phase changes and bottlenecks during execution. We evaluate the CBP and the prefetch-aware DRAM controller (PADC) [15] techniques by integrating them with DBLEP and we show the resulting improvements in Section 6.

4. Block Level Architectural Profiler

In this section, we introduce DBLEP and present its hardware implementation. DBLEP consists of three parts: Behavior Detection, Behavior Labeling and Behavior Storage. We then describe additional schemes to further improve the profile information and reduce or avoid the potential impact of DBLEP on the critical path delay. The hardware overheads of the three stages are then discussed.

Figure 3 shows a simplified block diagram of the DBLEP implementation in an Out-of-Order (OoO) processor. DBLEP's basic idea is to identify the potential bottleneck in a block by detecting instruction stalls and storing the information in the BTB when the last instruction of the block commits.

4.1. Behavior Detection

With an in-order commit stage, instructions that take long to commit may stall the processor. We consider the instructions that stall the commit stage for the longest amount of time to be the instructions that characterize the block's performance degradation. DBLEP modifies the commit stage to collect profile information at the basic block level.

In Figure 4, we show a flowchart of the commit stage in a superscalar processor with the additional steps needed to implement our detection mechanism. Notice that the DBLEP implementation requires an in-order commit stage, which is widely used in current processors.

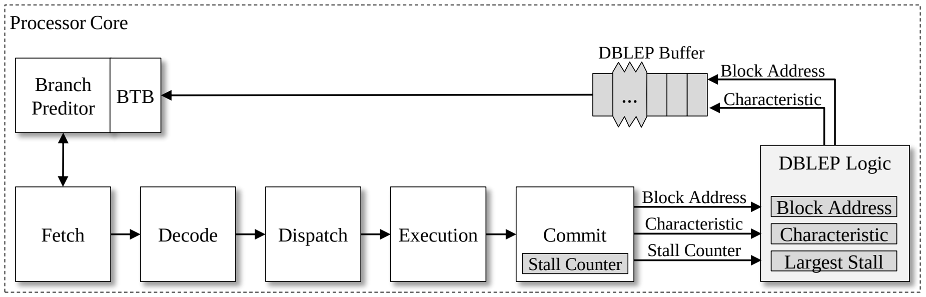

Figure 3: Overview of the operation of DBLEP in a superscalar processor. Parts in gray represent DBLEP's modifications or additions to the processor.

The commit stage checks whether the oldest instruction is ready to commit $\boxed{1}$. We add a *Stall Counter* that is incremented whenever an instruction stalls the commit stage $\boxed{2}$. When the instruction is ready to be committed, we check if it is a branch $\boxed{3}$, as a branch indicates the end of a block. If it is not a branch, we should also check whether it is the first instruction being committed in that cycle, as we only wish to identify instructions that stalled the commit stage $\boxed{4}$. Only the first instruction committed in each cycle could possibly have stalled the commit stage, so we only observe these instructions. Clearly, if it is not the first instruction being committed and it is not a branch, no action is necessary, as the instruction did not stall the commit stage. However, if it is the first instruction to commit, we compare its accumulated stall time to a register that holds the previous largest stall time of the current block $\boxed{5}$. If the stall counter is larger than the previous value register, we update the latter, set a *Characteristic* indicator to 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 store the block profile information. We first check whether the branch was correctly predicted $\boxed{7}$. As branch instructions do not stall the commit stage, we can only set the Characteristic indicator to *Brch* if the branch is not

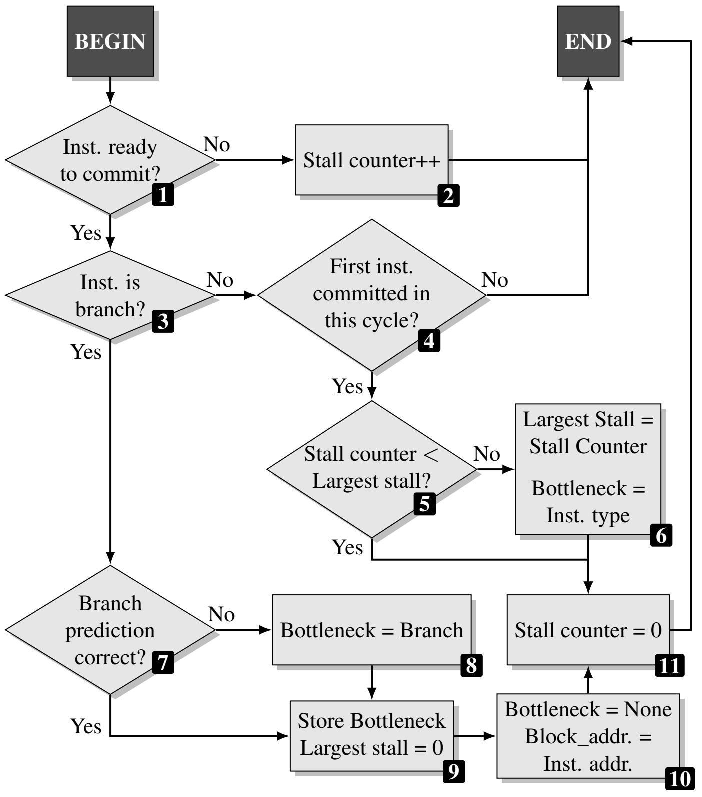
BEGIN
END
Inst. ready to commit?
1
No
Stall counter++
2
Yes
Inst. is branch?
3
No
First inst. committed in this cycle?
4
No
Yes
Yes
Stall counter < Largest stall?
5
No
Largest Stall = Stall Counter
Bottleneck = Inst. type
6
Yes
Branch prediction correct?
7
No
Bottleneck = Branch
8
Stall counter = 0
11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

Figure 4: Flow chart of the modified commit stage. Instruction types are limited to FP or MEM, INT instructions are not considered.

correctly predicted 8. We then store the *Characteristic* indicator in the BTB, using the value of the Block_Address register as index (the destination 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) 10. In other words, 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 9. We also reset the register that holds the largest stall count 9 and the Characteristic 10, followed by resetting the stall counter 11.

4.2. Behavior Storage

In order to use the profiled Characteristic, we store it for future use. Based on the results of the correlation study we have decided to use only 2 bits per block, allowing us to have four distinct Characteristics: (*None*, *Brch*, *Mem*, *FU*). 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 the Characteristic for each block targeted by a branch. This way, if a branch is taken, we can employ the Characteristic stored in the BTB entry as we know that it is the characteristic profiled for the next block. The register *Block_Address*, is responsible for indexing each block in the BTB. To prevent profiling characteristics from being affected by cold start effects in caches, prefetchers and branch predictors, we designed a stabilizer for DBLEP. This stabilizer deals with problems that may arise when a block has an unstable characteristic in its first executions. A 2-bit comparison and one stability bit are used per entry. The stability bit is set whenever we detect the same characteristic for the block twice in a row, by using a 2-bit comparison. If the bit is set, we consider the characteristic to be stable, and will forward the value whenever the block is accessed. Further changes in the block behavior will not overwrite the BTB entry, in order to avoid disabling the improvements that may have resulted from a bottleneck reduction.

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 have 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 a few, smaller internal basic blocks, 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 smaller blocks often represent an inner loop, they will be executed enough times to be characterized properly. If they do not, then they are likely not relevant.

We have experimented with two different implementations of DBLEP [8], the first using the BTB to store the relaxed blocks' profile and the second using a cache to store real basic block profiles, with a large enough cache to avoid conflicts and capacity misses in all benchmarks. Our experiments have shown that the large number of entries in the BTB is sufficient to store the basic blocks profile information for most of the benchmarks. We have also observed that the results obtained with the relaxed blocks are often better than those with basic blocks due to the greater focus on loop codes, which have more impact on the application's behavior. Thus, all the results in this paper were obtained using the BTB-based implementation of DBLEP.

4.3. 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 contents of this register are copied to a new field for every entry for that instruction, e.g., the Reorder Buffer (ROB) entries, until the content of the *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.

4.4. Block Behavior Stability

As memory access latency is variable, our block characterization is inherently unstable. The variability of the memory access latency is due to the cache hierarchy, and to the different data and access patterns associated with the different inputs to the benchmarks, which also affect branch prediction. There is no way to precisely predict how the input data will affect the average instruction stall.

Therefore, to keep up with the possible changes in the behavior of blocks, we have designed a simple extension of the mechanism. Previously, we trained each entry for a value, and once it stabilized, the block retained the same characteristic until it was evicted from the branch target buffer. We now added a 3-state finite-state machine per entry to allow characteristic changes during execution. The three states are: training, trained and labeled-training. Training means there there is still no stable characterization for the entry. If an entry is in the "training" state, its value represents the last characteristic detected for the block. Once we match a characteristic with the value already present in the entry, the state changes to "trained". Entries which reach the "trained" state will label their respective blocks with the entry value in the fetch stage. However, if a different characteristic is detected while an entry is in the "trained" state, we keep the current characteristic but change the state to "labeled-training". When an entry is in the "labeled-training" state, it will still label blocks as before. If the same characteristic is detected again, we upgrade it to "trained" but, if a different characteristic is detected, the entry is downgraded to "training" and will no longer label fetched blocks. This implies a "None" label to all fetched blocks for which we cannot provide characterization.

4.5. Critical Path Implications and Hardware Costs

The detection scheme of DBLEP requires the addition of several registers and counters and the logic necessary for their updates. In addition, the extra steps required by DBLEP could possibly lengthen the clock cycle time.

To prevent an increase in the clock cycle time we have taken the following steps. First, we make sure that the detection stage of DBLEP can be done without increasing the clock time. Updating the registers in stages 2, 4, 5, 6, 8, 10 and 11 can be split between two cycles or postponed altogether to the next cycle, as branches which finish the block will already get the updated value. Detection of branches in stage 3 can be done using the decoded opcode. Stage 5 is illustrative: we always consider only the first instruction committed in the cycle, since all other instructions block the commit stage for 0 cycles. Stage 7 is the only real concern during the detection process and can be implemented in two different ways. One way is to consult the branch prediction table, but this could require additional clock cycles due to the access and wire delay involved in obtaining the data. A more reasonable way is to have one extra bit for all ROB entries. When a branch is detected as mispredicted, we can set this bit for that specific branch, and later on, in the commit stage when retiring the instruction, we can recover this information through this single bit. This is likely already done for Intel's hardware counters [16].

Second, obtaining stall information in the commit stage and storing it in the BTB in the same cycle may still require a longer cycle time. To avoid this, we write to a buffer and create an additional pipeline stage used only for DBLEP. This extra stage stores the information obtained for the last block in the BTB and does not affect the processor's throughput. This stage ensures synchronization with the BTB reads performed by all instructions being fetched, so the written value is only valid after 3 cycles [16]. We conducted experiments to find out whether the buffer is large enough, and found that only 0.05% of all commit cycles in multiple workloads (SPEC-CPU 2006, NPB-OMP, SPEC-OMP 2001) contain more than one branch commit. Still, if two branches are committed in the same cycle, the second block can simply be discarded, as its stall value is zero. If, in the same cycle, the branch unit tries to write a target address in the BTB, we assign a higher priority to the branch unit and postpone DBLEP's write to the next cycle. The labeling step has no implication on the critical path as it requires only the forwarding of a few bits through the pipeline along with their respective instruction.

DBLEP's hardware costs can be divided into detection, storage and labeling. For each core, DBLEP requires a total storage of 2,142 bytes, plus three 2-bit multiplexers, 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 about 206,548 transistors. In an 8-core Sandy Bridge processor, this amounts to 1,652,384 transistors, out of 2.27 billion transistors, which constitutes an overhead of less than 0.08%. A detailed explanation of the hardware costs can be found in [8].

5. Evaluation Methodology

5.1. Simulation Environment and Workloads

To validate our mechanism, we used an in-house, cycle-accurate simulator [17]. It accurately simulates the micro-architecture, modeling all functional unit contentions, register dependencies, processor system restrictions (e.g., memory disambiguation), in addition to the cache architecture, DRAM and interconnections. In Table 2, we provide the parameters of the simulated system, whose configuration is based on Intel's 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 sets and seven applications from the SPEC-OMP2001 benchmark suite with the *ref* input sets.

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.

These application suites were selected to increase pressure on the memory system. Single-threaded benchmarks do not gain or lose performance in our simulations for any of the mechanisms we study in this paper. The reason for this will become clear in Section 7, where we show how our mechanism improves performance.

5.2. Evaluated Memory Controller Policies

Given the correlation coefficients presented in Section 2, we chose to improve the memory controller design since memory accesses were more frequently correlated with degraded performance. Following the proposals of Ghose *et al.* [14] and Lee *et al.* [15], we designed an improved memory controller that can use the profile information provided by DBLEP 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 assigns the highest priority to row hits (First Row), thus lowering the average memory wait time, and then serves 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.* [14] and the Prefetch-Aware DRAM Controller (PADC) from Lee *et al.* [15]. 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 in the original papers.

Table 2: 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;
	Decode and commit up to 5 instructions; Rename/dispatch/execute up to 5 μ instructions;
	18-entry fetch buffer, 28-entry decode buffer; 1 branch per fetch; 8 parallel in-flight branches;
	168-entry ROB; 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);
	MOB entries: 64-read, 36-write; 1-load and 1-store functional units (1-1 cycle);

Branch predictor	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; MSHR: 8-request, 10-write-back, 1-prefetch;
	LRU policy; 2-cycle; Stride prefetch: 1-degree, 16-strides table;

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

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

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

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;

We have, however, observed that as the memory pressure increases, the improvements achieved by these mechanisms and our implementations increase.

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 $100,000$ 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 two bits: a prefetch bit and an access bit. These bits track which prefetches were useful. By measuring the prefetch accuracy every $100,000$ cycles, PADC determines whether it should assign the same priority to prefetches and demands, or whether it should prioritize demands and drop prefetches based on the prefetch accuracy. The authors define four levels for their scheme. 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 have waited in the memory request buffer for more than $50,000$ cycles to be serviced. Between 10% and 30% accuracy, the mechanism drops those prefetches that waited for more 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 DBLEP-based mechanisms were implemented as follows. *DBLEP-CBP* is an adaptation of CBP using the basic block profile information provided by DBLEP. We give priority to blocks that DBLEP characterized as *Mem*, through the CBP memory controller policy. DBLEP-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 *DBLEP-PADC*, generated prefetches get DBLEP 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 on 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 eight 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 5 illustrates the request selection logic for different memory controller schemes. The information bits from the request are compared as a single number, by concatenating all the bits and considering the left-most bit as the most significant one. 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 DBLEP. Finally, row hit is 1 if the address of the request matches the currently open row.

In comparison to CBP, the first advantage of DBLEP-CBP is that we can consider other processor bottlenecks besides memory pressure. The second advantage of this

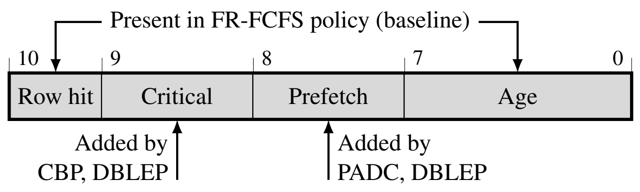

Figure 5: Request selection logic for different memory controller schemes.

approach is that it also provides information on branch mispredictions, which can result in further performance gains. We will not assign higher priority to loads that are followed by a mispredicted branch as doing so would not enhance the block performance. Thirdly, 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, DBLEP-PADC requires four times less storage than PADC by using the BTB to store the profile information. We also achieve better performance due to improved selection on when to drop prefetches and which prefetches to drop. We do so by prioritizing critical prefetches, and, as these get serviced faster and within the average demand time, prefetches to blocks which are not suffering of memory stalls are likely to be dropped first.

6. Experimental Results

In this section we present and compare the characterization accuracies of the BLAP and the DBLEP mechanisms. We then report the performance improvements achieved by BLAP and DBLEP.

6.1. Accuracy of BLAP and DBLEP

As observed in previous research, using the commit stall cycle count allows characterizing blocks that have critical loads and consequently, enables performance improvements through the use of priority and prefetch-dropping policies. However, as this is a heuristic based on past behavior to predict future executions, we can not expect an accuracy of 100%.

In order to assess and further improve the accuracy of our method, we modified BLAP to detect characteristics corresponding to instruction types we have shown in Section 2 and *None*. Therefore, we measure the accuracy for 10 characteristics (that is, *None*, 3 integer types, 3 floating-point types, load, store, and branch). We measure accuracy by comparing the characteristic that our mechanism has identified for each block to the stall type that was actually detected as the largest for that specific block execution. Figure 6 shows the accuracy results for the BLAP mechanism. Each bar represents all the blocks executed by the given benchmark. The *Accurate* portion represents block executions in which the largest stall characteristic provided by BLAP matched the characteristic with the largest stall for those executions, the *Inaccurate* portion represents the mismatch between the two, and the *Training* portion represents block executions in which BLAP did not yet detect the same characteristic twice in a row for that block.

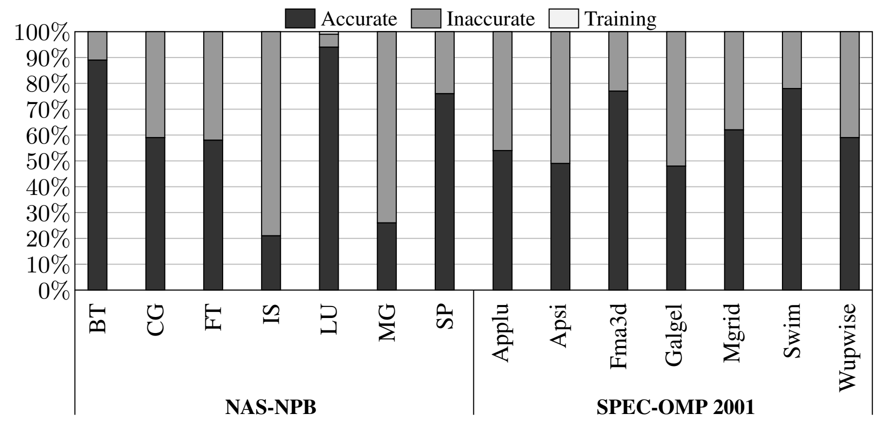

Figure 6: The precision of the BLAP mechanism when fixing a characteristic after two detections in a row.

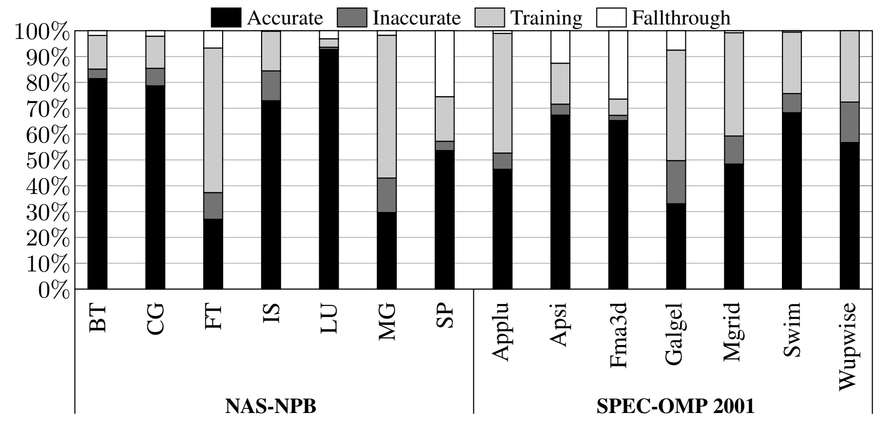

Figure 7: The accuracy of the DBLEP mechanism using a 3-state finite state machine for each entry.

To compare the accuracy of the mechanisms, we use the geometric mean of all benchmarks' accuracy. The mean of all benchmarks' accuracy is 56.4%, which is not satisfactory. Moreover, several benchmarks, such as *IS*, have a very low accuracy. This can happen due to different input patterns for the same block, different latencies due to different numbers of memory accesses by the different threads, or fall through block (block following a not taken branch) characterization influencing the taken branch characterization.

To improve the profiling accuracy compared to BLAP, DBLEP does not record fall through blocks or provide labels to them. Furthermore, to dynamically adapt to

Table 3: Classification accuracy of BLAP and DBLEP (in %).

Mech.	BT	CG	FT	IS	LU	MG	SP	Aplu	Apsi	Fma	Ggel	Grid	Swim	Wup	Mean
BLAP	89.0	59.0	58.0	21.0	94.4	26.0	76.0	54.0	49.0	77.0	48.0	62.0	78.0	59.0	56.4
DBLEP	83.8	86.3	31.5	67.5	96.0	51.6	74.4	42.7	77.3	71.0	36.2	57.5	67.5	58.4	64.7

different phases that depend on data input patterns and memory pressure, we have added a 3-state finite state machine for each entry. The accuracy results of DBLEP are shown in Figure 7, with block executions split into four categories. If we correctly (incorrectly) label a block, we indicate it as "accurate" ("inaccurate"). If a block is undergoing training and is not labeled, we indicate is as "training". Finally, if a block executes after a not taken branch, we consider it to be the fall through case, for which we have no prediction, and thus we label the block as "None".

We can observe that the number of blocks executions regarded as "training" is much higher than the number detected using BLAP. The resulting inaccuracy is also much lower, since the unstable behavior present in blocks was not detected by BLAP.

In Table 3, we can observe that when looking only at characterized blocks (accurate, inaccurate and training, thus dismissing fallthrough blocks), accuracy is higher (a mean of 64.7% with DBLEP against 56.4% using BLAP). These results show the unstable nature of blocks during execution, which can come from training other mechanisms (caching, branch prediction), or different program phases. However, the performance results presented below show larger performance improvement.

Taking *MG* as an example, the number of blocks that were found to still be under training due to characteristic changes went up from 9,558 to 33,472,725 instances, implying the mechanism cannot keep up with the behavior instability of the benchmark's blocks. Obviously, benchmarks with blocks that exhibit an unstable behavior that can not be accurately characterized using their past behavior, will never have a high characterisation accuracy. As we attempted to identify the instruction type that had the largest number of stalls for 10 types (*None*, 3 integer types, 3 floating-point types, load, store, and branch), characteristic changes are likely to happen upon small fluctuations of the number of stalled cycles for any of those types of instructions.

6.2. Performance Improvements with BLAP and DBLEP

Figure 8 shows the performance results for four different mechanisms for the NAS-NPB and SPEC-OMP2001 benchmarks, all using BLAP as base detection mechanism. In the figure, we show the speedup in terms of execution time for all benchmarks, normalized to the baseline. The first observation that we can make is that the mean gains of the two related mechanisms (CBP and PADC) are different from their own results, due to the use of different benchmarks, architectural parameters and simulators. Although the impact of our mechanism implementations are noticeable, as seen for the PADC mechanism in the *IS* benchmark, the average benchmark improvements remain low for all implementations.

In Figure 8, PADC shows the highest improvement, achieving 19.0% for the *IS* benchmark. We observe average performance improvements of 1.9% for CBP, 0.8% for BLAP-CBP, 3.1% for PADC and 3.9% for BLAP-PADC. In general, CBP achieves

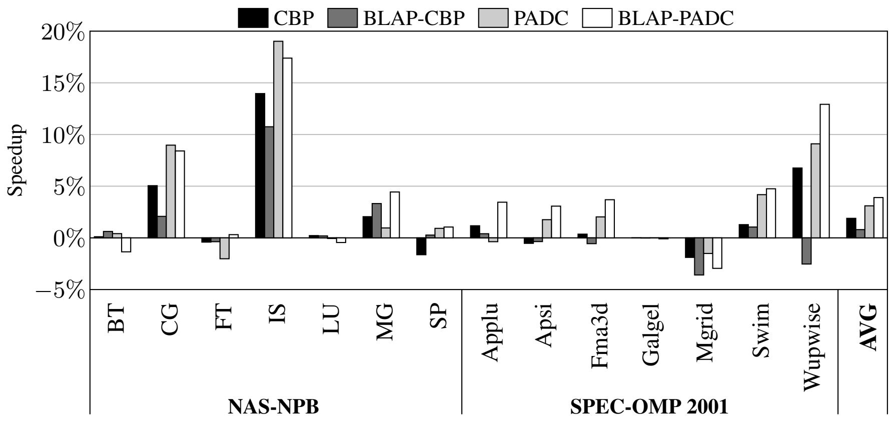

Figure 8: Performance results for NAS-NPB and SPEC-OMP2001, relative to the FR-FCFS baseline.

better results than BLAP-CBP. This is due to CBP's information being specifically designed for load instructions, while BLAP generates a more general profile. We conclude that instruction-granularity performs better than block granularity for this case.

BLAP-PADC outperforms PADC on average as we adapted it to perform in a flexible way, by using the average demand request time. Using BLAP profiling information, the mechanism is able to drop prefetches more aggressively, while still servicing important ones. This is because the prioritized prefetches come from critical, repetitive blocks. This also makes BLAP-PADC more likely to drop false-positive triggered prefetches, as they have low priority. Thereby, the BLAP framework can outperform previous works by combining their techniques with a reduced hardware overhead.

Figure 9 presents the performance results for the DBLEP mechanism, which uses a finite state machine for each branch target buffer entry (allowing to quickly adapt to new characteristics) and has no fall through block recording or labeling.

The first relevant observation is that the *BT* and *Mgrid* benchmarks no longer experience performance loss. The benchmarks that previously lost more performance, *LU* and *Galgel*, are now 0.77% and 0.00% faster than the baseline. *FT* now shows a negligible performance degradation, as our mechanism can barely train it. Furthermore, all the benchmarks that had previously shown performance improvements benefit from the higher accuracy. Two good examples are *SP*, which now shows a 6.0% performance improvement instead of the previous 1.0%, and *Wupwise*, which increased its performance improvement from 12.9% to 15.6%. The average performance improvement went from 3.9% to 5.3% with the implementation of a finite state machine per entry. Besides *FT*, only the *MG* benchmark had a diminished performance improvement.

The hardware cost for the finite state machine consists of 1 additional bit for every branch target buffer entry, therefore we need another 512 bits per core, resulting in a total extra storage of 4096 bits.

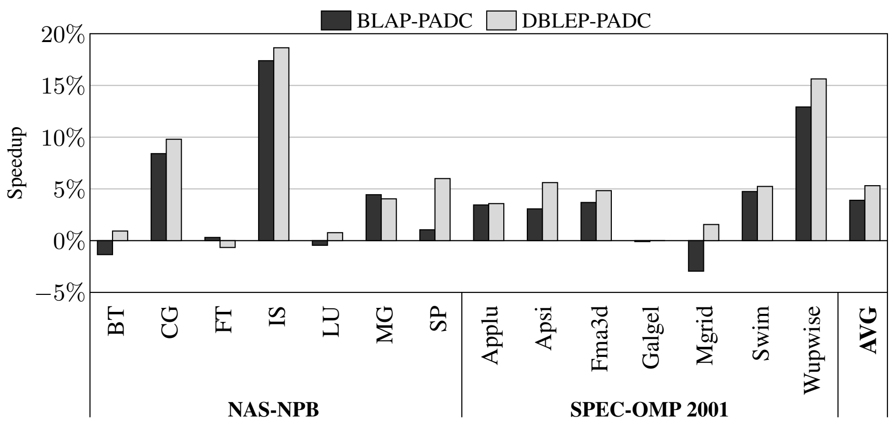

Figure 9: Performance results when settling on a characteristic after two detections in a row (BLAP) and with a finite state machine and no fall through block recording or labeling (DBLEP), relative to the FR-FCFS baseline.

7. Analysis of Performance Gains

Our mechanism is sensitive to several characteristics of the executing benchmark. We have observed that most prefetches made by the studied benchmarks are accurate with an average accuracy of 95.7%. Therefore, we are likely dropping prefetches that are useful, and yet achieving performance improvements. Hence, we can deduce that the average demand service time has been reduced. To analyze this in detail, we plot in Figure 10 three statistics of the Dynamic Block-Level Execution Profiler (DBLEP)-PADC's execution normalized to the FR-FCFS baseline for each benchmark: (1) read average time, which represents the average number of processor cycles that read requests require to be serviced in the main memory; (2) absolute number of demand read requests, which represents the number of demand read requests that were serviced in the main memory; and (3) total read service time, which is the product of the previous two and represents how many cycles the memory spent servicing demand reads. We can observe that all benchmarks had their average demand read time reduced, as the pressure in the memory was alleviated due to the prefetch dropping. An intuitive explanation of the above mentioned phenomenon is that prefetches may open different row buffers and cause row conflicts for demand reads, especially among threads competing for access to the same bank.

In Figure 10, we can see the characteristics of each benchmark relative to the baseline. We can also see that, as expected, by dropping useful prefetches, we have more cache misses and therefore, more demand requests to the main memory, which intuitively would lead to performance degradation. The third bar in Figure 10 shows that the total time consumed in memory has a slight correlation with our performance gains. The reason we still observe performance gains despite the fact that useful prefetches were dropped (in several of the benchmarks) is due to an interplay between the prefetch drop mechanism and the criticality mechanism. Consider, for example, the *CG* and *IS* benchmarks that show a direct reduction in the total amount of time spent per read

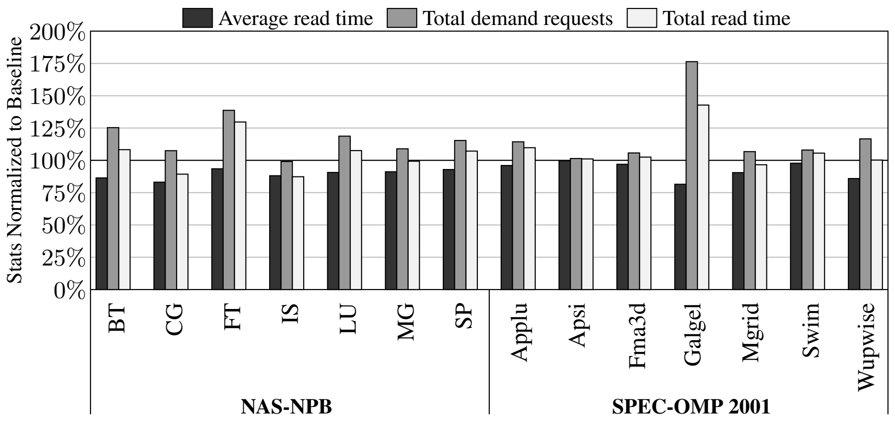

Figure 10: Average memory read latency, total number of demand requests and total memory read latency using our mechanism normalized to the baseline.

request. In contrast, *Galgel* does not achieve performance gains as, even though the average read time has reduced, the overall number of additional requests made the total time spent on reads quite large. *Galgel* has a relatively small memory footprint (less than 4MB), thus limiting the potential performance changes, and we can deduce that the absolute demand service time in the main memory is small compared to the total number of the benchmark's execution cycles. The benchmark *Wupwise* is an outlier which exemplifies the advantage of the reduced average read time. As the cores stall their instructions due to data dependencies, speeding up loads becomes critical for instruction-level parallelism, which is why, despite spending a larger amount of time using the memory, it still achieves improved performance due to the ILP obtained.

A direct way of validating this is by examining the average time each load stalls the commit stage. We compare our mechanism to the baseline in Figure 11 and observe that the combination of the prefetch drop and criticality mechanisms works in the same way of fairness-oriented mechanisms [19, 20], as criticality and prefetch-dropping indirectly enable all cores to get their share of demand requests serviced quickly. The benchmark *Apsi* is especifically interesting as its commit stall time is reduced over 8%, but the performance is only improved by 5%, due to other characteristics which quickly block the commit in place of the loads. This shows the unstable nature of blocks and the close competition of multiple characteristics for the largest stall value in each block.

8. Conclusions

Our results show that the proposed characterization is highly relevant for memory accesses. However, our findings indicate that as the behavior of blocks may change during the program's execution, we must be able to adapt the characterization of the block due to different block combinations in each program phase, as these combinations lead to different memory pressures and instruction mixes. By adapting our mechanism to dynamically characterize blocks and thus creating DBLEP, we were able to

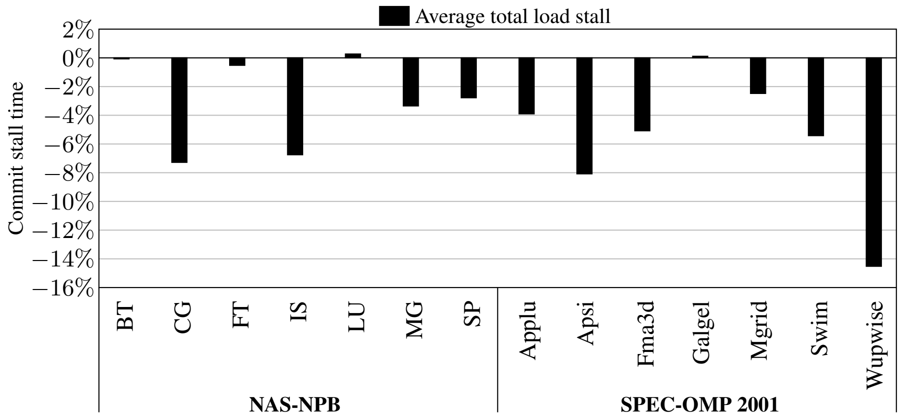

Figure 11: Average number of cycles in which the commit stage was stalled due to load instructions for our mechanism, compared to the baseline.

improve performance by 5.3% on average, when compared to the baseline FR-FCFS, with higher improvements than a static characterization mechanism. In this way, we have shown that a higher characterization accuracy also leads to better performance improvements, motivating future studies on how to further improve the characterization of blocks.

Another significant finding focuses on the way prefetches interact with parallel benchmarks. Similar to previous research on prefetches for multi-programmed workloads [21, 22], our results show that prefetches also degrade performance of parallel workloads on cores that share the prefetched data, as dropping prefetches results in better performance by ensuring reduced average demand service time, and thus indirectly balancing the service time that all cores experience for their memory accesses.

In the future, we intend to characterize blocks in regard to all instruction types and use this for complex events, such as data-dependent branches [23] and value reuse for predictable blocks [2]. This extension will require fundamental changes to the mechanism, involving not just detecting the highest characteristic stall, but characterizing blocks with multiple labels to indicate characteristics that stalled more than a certain threshold. Basic block detection at the commit stage can also be done in parallel with group commit [24] to enable the implementation of the aforementioned complex techniques based on basic block analysis.

Acknowledgment

This work was partially supported by CNPq and Capes.

References

[1] C. Lattner, V. Adve, Llvm: A compilation framework for lifelong program analysis & transformation, in: Code Generation and Optimization, 2004. CGO 2004. International Symposium on, IEEE, 2004, pp. 75–86.

[2] J. Huang, D. Lilja, Extending value reuse to basic blocks with compiler support, Trans. on Computers 49 (4) (2000) 331–347. doi:10.1109/12.844346.

[3] J. Cocke, Global common subexpression elimination, SIGPLAN Not. 5 (7).

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

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

[6] P. Ratanaworabhan, 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, S. S. Lumetta, replay: A hardware framework for dynamic optimization, Trans. on Computers 50 (6).

[8] F. B. Moreira, M. A. Z. Alves, M. Diener, P. O. A. Navaux, I. Koren, Profiling and reducing micro-architecture bottlenecks at the hardware level, in: Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on, 2014, pp. 222–229.

[9] R. Jain, D. Menasce, L. W. Dowdy, V. A. Almeida, C. U. Smith, L. G. Williams, The art of computer systems performance analysis: Techniques.

[10] T. Sherwood, E. Perelman, 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.

[11] G. Hamerly, E. Perelman, J. Lau, B. Calder, Simpoint 3.0: Faster and more flexible program phase analysis, Journal of Instruction Level Parallelism 7 (4).

[12] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, A. Karunanidhi, Pinpointing representative portions of large intel itanium programs with dynamic instrumentation, in: Int. Symp. on Microarchitecture (MICRO), 2004.

[13] N. Clark, A. Hormati, S. Yehia, S. Mahlke, K. Flautner, Liquid simd: Abstracting simd hardware using lightweight dynamic mapping, in: High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on, IEEE, 2007, pp. 216–227.

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

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

[16] Intel, Intel 64 and IA-32 Architectures Software Developer's Manual (2015).

[17] M. A. Z. Alves, M. Diener, F. B. Moreira, C. Villavieja, P. O. A. Navaux, Sinuca: A validated micro-architecture simulator, in: IEEE International Conference on High Performance Computing and Communications (HPCC), 2015.

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

[19] Y. Kim, M. Papamichael, O. Mutlu, M. Harchol-Balter, Thread cluster memory scheduling: Exploiting differences in memory access behavior, in: Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on, IEEE, 2010, pp. 65–76.

[20] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, O. Mutlu, Mise: Providing performance predictability and improving fairness in shared main memory systems, in: High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on, IEEE, 2013, pp. 639–650.

[21] E. Ebrahimi, O. Mutlu, C. J. Lee, Y. N. Patt, Coordinated control of multiple prefetchers in multi-core systems, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2009, pp. 316–326.

[22] E. Ebrahimi, C. J. Lee, O. Mutlu, Y. N. Patt, Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems, in: ACM Sigplan Notices, Vol. 45, ACM, 2010, pp. 335–346.

[23] M. U. Farooq, K. Khubaib, 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.

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