UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FRANCIS BIRCK MOREIRA

Profiling and Reducing Micro-Architecture Bottlenecks at the Hardware Level

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Dr. Philippe Olivier Alexandre Navaux
Advisor

Porto Alegre, August 2014

CIP – CATALOGING-IN-PUBLICATION

Moreira, Francis Birck

Profiling and Reducing Micro-Architecture Bottlenecks at the Hardware Level / Francis Birck Moreira. – Porto Alegre: PPGC da UFRGS, 2014.

75 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2014. Advisor: Philippe Olivier Alexandre Navaux.

1. System Architecture. 2. Program Profiling. 3. Hardware Design. I. Navaux, Philippe Olivier Alexandre. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof^a. Carlos Alexandre Neto
Pró-Reitor de Coordenação Acadêmica: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora Adjunta de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecário-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“I do not know what I may appear to the world, but to myself I seem to have been only like a boy playing on the sea-shore,
and diverting myself in now and then finding a smoother pebble or a prettier shell than ordinary, whilst the great ocean of truth lay all undiscovered before me.”

— Sir Isaac Newton

ACKNOWLEDGMENTS

Well, I should as well make this the longest chapter of this dissertation. First, I would like to thank my parents for, well, everything. There is simply too much in this “everything” to be detailed, so let us keep ourselves at that. Second, I must thank Laércio Lima Pilla for enduring my introverted, awkward presence while living together for three and a half years, and for still being by far a better friend than I should hope to have. Thanks to him, I was able to make friends and become somewhat more human again. I’ll also use this space to thank Francieli Zanon Boito, his girlfriend, for all the guaca mole, and for becoming a friend with who I am usually able to identify with myself. In this same line, I must thank my cousin Herberth Birck Fröhlich, for living with me for four years and a half, and always being able to make me laugh in the simplest of ways. Thanks to his influence, I became someone who could even, at times, be considered fun and extrovert.

I am grateful to Philippe Olivier Alexandre Navaux, for the opportunity in the first place to engage in research. And when speaking of research, a special acknowledgment must go to Marco Antonio Zanata Alves for his endless help. Ever since I started researching, Marco has mentored me, taught me, laughed with me, corrected me, argued with me, corrected me again, and so on, until I eventually came up with something decent. This work would simply not exist were it not for Marco pushing me into MsC and always being there for me.

There are many more who I should thank, such as everyone in my laboratory, as everyone always helped: Vicente Cruz, Matthias Diener, Eduardo Cruz, Eduardo Roloff and others. I should also thank all the people who eventually became friends with me, and, somehow, changed me and my ways of thinking. Eight years ago I would not find it possible, but this number of friends became large enough that I should not list them here. Thank you all.

BLAP: Um Caracterizador de Blocos Básicos de Arquitetura

RESUMO

A maior parte dos mecanismos em processadores superescalares atuais usam granularidade de instrução para criar ou caracterizar especulações, tais como predição de desvios ou prefetchers. No entanto, muitas das características das instruções podem ser obtidas ao analisar uma granularidade mais grossa, o bloco básico de código, aumentando a quantidade de código coberta em um espaço similar de armazenamento. Adicionalmente, códigos podem ser analisados mais precisamente e prover uma variedade maior de informação ao observar diferentes tipos de instruções e suas relações. Devido a estas vantagens, a análise no nível de blocos pode fornecer mais oportunidades para mecanismos que necessitam desta informação. Por exemplo, é possível integrar informações de desvios mal previstos e acessos a memória para gerar informações mais precisas de quais acessos a memória oferecem melhor desempenho ao serem priorizados.

Nesta tese propomos o Block-Level Architecture Profiler (BLAP) (Block Level Architecture Profiler), um mecanismo em hardware que caracteriza gargalos no nível microarquitetural, tal como loads delinquentes, desvios de difícil previsão e contenção nas unidades funcionais. O BLAP trabalha no nível de bloco básico, apenas detectando e fornecendo informações que podem ser usada para otimizar tais gargalos. Um mecanismo para a remoção de prefetches e uma política de controlador de memória DRAM foram criados para usar a informação criada pelo BLAP e demonstrar seu potencial. Juntos, estes mecanismos são capazes de melhorar o desempenho do sistema em até 17.39% (3.9% em média). Nosso método mostrou também ganhos médios de 13.14% quando avaliado com uma pressão na memória mais alta devido a prefetchers mais agressivos.

Palavras-chave: Arquitetura de Sistemas, Perfil de Programas, Design de Hardware.

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. Moreover, the code can be profiled more accurately and provide a higher variety of information by analyzing different instruction types inside a block. Because of these advantages, block-level analysis can offer more opportunities for mechanisms that use this information. For example, it is possible to integrate information about branch prediction and memory accesses to provide precise information for speculative mechanisms, increasing accuracy and performance.

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

Keywords: System Architecture, Program Profiling, Hardware Design.

LIST OF FIGURES

2.1 Example of code presenting the classical definition of basic blocks (BBL) and our relaxed definition (RBL) 24
2.2 Statistics for the most relevant blocks of the *libquantum* benchmark, and their relationship with performance measured in IPC 25

4.1 Basic block characteristic distribution. Every block receives only one characteristic, the most relevant one according to the hardware counters 36
4.2 Basic block characteristic distribution. Every block receives only one characteristic, the most relevant one according to the hardware counters. Now statistics are properly attributed to each block. 37
4.3 Basic block characteristic distribution. Every block receives one characteristic, the most relevant one according to the sum of register true dependencies delays per type 39
4.4 Basic block characteristic distribution. Every block receives only one characteristic, the most relevant one according to commit stage delays 40

5.1 Overview of the operation of BLAP in a superscalar processor. Parts in gray represent BLAP's modifications or additions to the processor. 43
5.2 Flow chart of additional commit stage events. 45
5.3 Modeled performance improvements correctly predicting 25%, 50%, 75% and 100% of branch instructions in blocks characterized as "Brch". 50
5.4 Modeled performance improvements solving 25%, 50%, 75% and 100% of the load instructions of blocks characterized as "memory load". 50
5.5 Modeled performance improvements solving 25%, 50%, 75% and 100% of the load instructions of blocks characterized as "memory load", when the mechanism *only* characterizes loads. 51

6.1 Request selection logic for different memory controller mechanisms. 56

7.1 Performance results for NAS-NPB and SPEC-OMP2001, relative to FR-FCFS baseline. 57
7.2 Performance results for NAS-NPB and SPEC-OMP2001 with increased aggressivity prefetcher, relative to FR-FCFS baseline. 58
7.3 Performance results for NAS-NPB and SPEC-OMP2001 with conservative aggressivity prefetcher, relative to FR-FCFS baseline. . . . 59
7.4 Performance results comparison between using the branch target buffer and a large cache, relative to FR-FCFS baseline. 60

7.5 Performance results comparison using the same memory controller, but different informations, from BLAP and CBP respectively, relative to FR-FCFS baseline. 60
7.6 Mechanism performance, normalized to baseline configurations with different memory latencies 61
7.7 Mechanism performance with BLAP-PADC-8L using Branch Target Buffer (BTB), normalized to baseline configurations with 16 cores . . 62
7.8 Mechanism performance, normalized to baseline configurations with different memory latencies 62

LIST OF TABLES

2.1 Pearson moment-product correlation coefficients of absolute statistics per block and performance in IPC for sequential benchmarks. 27

2.2 Pearson moment-product correlation coefficients of absolute statistics per block and performance in IPC for parallel benchmarks. 28

5.1 Hardware Costs of BLAP 49

6.1 Baseline simulated architectural parameters. 54

LIST OF ABBREVIATIONS AND ACRONYMS

BBL	Basic Block
BBV	Basic Block Vector
BHT	Branch History Table
BLAP	Block Level Architectural Profiler
BTB	Branch Target Buffer
CBP	Criticality Binary Prediction
CPU	Core Processing Unit
FR-FCFS	First Row - First Come First Serve
GPU	Graphic Processing Unit
ILP	Instruction-Level Parallelism
IPC	Instructions per Cycle
ISA	Instruction Set Architecture
LLC	Last Level Cache
LRU	Least Recently Used
MOB	Memory Reorder Buffer
MSHR	Miss Status Handle Register
OBP	Online Behavior Predictor
PADC	Prefetch-Aware DRAM Controller
RBL	Relaxed Block
ROB	Reorder Buffer
TLP	Thread-Level Parallelism
x86	Intel's ISA x86

CONTENTS

1 INTRODUCTION . . . 19
1.1 Introduction . . . 19
1.2 Contributions . . . 20
1.3 Organization . . . 21

2 BACKGROUND . . . 23
2.1 Basic Block and Relaxed Blocks . . . 23
2.2 Basic Block Characteristics and Performance . . . 24
2.3 Correlation between Characteristics and Performance . . . 25

3 ANALYSIS OF THE STATE-OF-THE-ART . . . 29
3.1 Code Behavior Detection and Use . . . 30
3.2 Basic Block and Phases Use Cases . . . 31
3.3 Hardware Design Opportunities . . . 31
3.4 Summary of the State of Art . . . 33

4 BLOCK CHARACTERIZATION . . . 35
4.1 Introduction . . . 35
4.2 Hardware Counter Classification . . . 35
4.3 Register Dependence Latency Classification . . . 37
4.4 Stall Commit Classification . . . 38

5 BLOCK LEVEL ARCHITECTURAL PROFILER . . . 43
5.1 Behavior Detection . . . 43
5.2 Behavior Storage . . . 46
5.3 Behavior Labeling . . . 46
5.4 Critical Path Implications . . . 47
5.4.1 Profile Stability . . . 47
5.4.2 Hardware Costs . . . 48
5.5 Evaluating BLAP Precision . . . 48

6 EVALUATION METHODOLOGY . . . 53
6.1 Simulation Environment and Metrics . . . 53
6.2 Evaluated Memory Controller Policies . . . 53

7 EXPERIMENTAL RESULTS . . . 57
7.1 Mechanism Exploration . . . 57
7.2 Design Space Exploration . . . 60
7.2.1 Memory Latency . . . 61

7.2.2 Cores Number 61
7.2.3 Cache Size 61

8 CONCLUSIONS AND FUTURE WORK 63
8.1 Contributions 63
8.2 Future Work 64

REFERENCES 65

9 APPENDIX - PORTUGUESE SUMMARY 69
9.1 Introdução 69
9.2 Detecção de Blocos Básicos 70
9.3 BLAP: Proposta de Detecção de Blocos Básicos 72
9.3.1 Detecção 72
9.3.2 Armazenamento 73
9.3.3 Rotulamento 73
9.3.4 Implicações no caminho Crítico 73
9.3.5 Custos de Hardware 73
9.4 Resultados 74
9.5 Conclusões 75

1 INTRODUCTION

Currently, industry has reached the limits of Instruction Level Parallelism (ILP) for the available computing system models, which feature superscalar out-of-order execution. Extensive research has enabled compilers to optimize a program's fundamental building blocks (also known as basic blocks) for specific architectures, mostly through profiling with specific data inputs. However, a dissonance is still present in the code optimization, as compilers cannot leverage the hardware state information of different architectures, which leads to the concepts present in this thesis.

1.1 Introduction

Characterization of basic blocks is an important, recurring technique, used for automatic optimization of several types. Software tools such as Vtune (REINDERS, 2005) allow manual analysis to detect performance improvement opportunities, such as rewriting code to avoid high cache miss rates for specific basic blocks, known as hotspots. The basic block granularity is especially useful (COCKE, 1970) as basic blocks represent portions of code that always end with conditional or unconditional branch instructions. A program's execution path is therefore defined by basic block execution sequences, enabling a program phase characterization and dynamic optimization. A recent example is the work of Kambadur et al. (KAMBADUR; TANG; KIM, 2012), which uses basic blocks to characterize the thread-level parallelism of an application in its different phases.

General-purpose processor designs (YUFFE et al., 2011) only collect information at the instruction level. Although several research papers used basic block analysis, most did so using a software approach, even for hardware adaptations (PANAIT; SASTURKAR; WONG, 2004; RATANAWORABHAN; BURTSCHER, 2008). One of the few techniques that actually performed basic block analysis at the hardware level was the rePlay framework (PATEL; LUMETTA, 2001). It analyzes the code to perform dynamic code optimization which is stored in a trace cache for future use, although no bottleneck profiling is performed. Another example is (CLARK et al., 2007), which uses post-commit analysis to make dynamic translation from scalar instructions to SIMD instructions (given offline step to convert SIMD to scalar instructions), although using a single function (detecting branch-and-link instructions) as scope to detect scalar instructions for translation.

Block profiling is usually done in software due to the high complexity of detailed profiling and analysis required. Nevertheless, profiling in hardware is interesting as it can leverage current hardware state information to efficiently generate relevant information of a program's execution, requiring no pre-analysis or source code modification. Recent trends in hardware development also point to the relation of instructions within blocks for performance improvements (AFRAM; ZENG; GHOSE, 2013; FAROOQ; KHUBAIB;

JOHN, 2013). The main limitation of basic block analysis is on what kind of analysis and characterization can be performed in hardware. It must be relevant enough to ensure performance improvements, and simple enough for effective trade-off.

With the mainstream acceptance of prefetchers (ZHUANG; LEE, 2007) and the recent inclusion of the memory controller into the processor chip (RIXNER et al., 2000), there is space for research using basic block information to provide information to such mechanisms. Sherwood et al. (2001) shows that there is a strong correlation between the characteristics found in the most relevant blocks of each application and the overall application characteristic. Based on this correlation, a basic block classification has the potential to provide relevant information regarding the application's performance. As basic block granularity automatically adapts to program phase changes, there is no need for additional control and time to adapt to changing phases. Additionally, the granularity allows for higher coverage with less hardware, opposite to designing with instruction granularity, such as in Ghose et al. (2013). Finally, any ideas that have been succesfully implemented using simple basic block classification in software could be enabled to be implemented in hardware.

1.2 Contributions

In this thesis, we propose BLAP. BLAP characterizes basic blocks according to the most relevant delays occurring per block, thus allowing improvement of future executions of these blocks. BLAP has several advantages over other mechanisms. It adapts to program phase changes, as it dynamically keeps track of basic blocks. It requires less storage than mechanisms which use instruction granularity, as we aggregate the behavior per block. We are able to use the BTB to efficiently store this information, as it retains the initial address of each block targeted by a branch. BLAP is capable of detecting different types of performance-related issues within a block, thus being able to provide information to a wide range of mechanisms.

In order to show the potential of BLAP, we explore the use of its profiling information to design an improved memory controller. Compared with the instruction-granularity information used by Ghose et al. (2013) and Lee et al. (2008), our mechanism achieves better performance with a scalable hardware overhead. Moreover, BLAP's basic implementation can be extended to provide detailed information regarding a wide range of bottlenecks at low hardware costs. To the best of our knowledge, no previous research has profiled basic blocks in hardware. Moreover, we present an integration between BLAP and other mechanisms, in order to show the usefulness of the profiled information. The main contributions of this thesis are the following:

Characterization Mechanism: We propose BLAP, an efficient detection mechanism capable of characterizing applications at the basic block level during their execution. The mechanism's highlights are the ability to store information using a BTB extension, therefore adding minimal overhead. It detects a varied amount of characteristics, and provides information to all instructions fetched in each core.

Mechanisms Integration: We integrated BLAP with mechanisms that improve memory performance by adapting them to use the profile information and by creating a new mechanism that relies on BLAP.

New Memory controller: In order to adapt existing concepts in a format that could use BLAP's information, we have created a new, aggressive prefetch-dropping memory controller.

1.3 Organization

The main objective of this work is to propose and study a hardware mechanism capable of detecting the blocks that build a program and characterizing their behavior. Such characterization can make it possible to improve the processor performance through its use by other mechanisms, such as prefetchers or priority policies. In Chapter 3, we cite the related work done in the area of basic block classification, and recent works that are used for concept implementation and performance comparison. In Chapter 4, we overview several methods to characterize blocks accordingly to their characteristics. In Chapter 5, the mechanism is detailed, in order to estimate the area overhead and feasibility of implementation. In Chapter 6, we describe the simulation environment and methodology, presenting preliminary results showing BLAP's potential. Chapter 7 details related work on memory access, how they can be adapted to use BLAP, and compares all the implementations. Chapter 8 finishes the thesis with our conclusions and suggestions for future work.

2 BACKGROUND

In this section, we explore the relationship between blocks and performance. We actually use a relaxed definition of a basic block (ANSALONI et al., 2013; COCKE, 1970; HUANG; LILJA, 2000) for our mechanism.

2.1 Basic Block and Relaxed Blocks

A basic block is a stretch of code with a single point of entry and a single point of exit. Thus, every basic block ends either with a branch instruction, or before an instruction targeted by a branch. This enables mechanisms based on basic blocks to automatically adapt to the program phase, as a program's phase is characterized by the blocks being used (RATANAWORABHAN; BURTSCHER, 2008). However, our low overhead hardware implementation allows multiple entry points, as it is not possible to efficiently detect the beginning of a block which was not targeted by a branch without changing the instruction set functionality. In Figure 2.1, we can see an example of what might happen during a code execution.

In Figure 2.1, we can see a simple matrix multiplication with fixed dimensions, both in C code as well as in AT&T assembly, for visualization of how a basic block is seen by the hardware. The assembly is composed of 5 real basic blocks: initialization (1st BBL), external loop - internal loop initialization (2nd BBL), internal loop (3rd BBL), external loop - increment and check (4th BBL) and return (5th BBL). These are divided in the real basic block, as for example, the first instruction of 2nd BBL is the target of the *jne 400508 <matmul+0x8>* instruction in address 400540, and being a new point of entry, must denote the end of 1st BBL and the beginning of a new BBL, 2nd BBL.

In the rightmost brackets in red, we can see how a hardware mechanism can detect a relaxed definition of basic blocks, noted by *RBL*. As the hardware does not know about new entry points ahead of time, it will detect the relaxed blocks in the following order. First, it will aggregate BBLs 1st, 2nd, and 3rd in the 1st RBL. Once instruction *jne 400518 <matmul+0x18>* at address 400533 jumps back to the beginning of the 3rd BBL, we will detect the 2nd RBL, which is the internal loop. This allows a relaxed definition to still obtain frequently repeated loop code. Once the instruction fails to jump and execution reaches the end of the first external loop, we will detect the 3rd RBL, similar to the 4th BBL. However, we will now create a 4th RBL that aggregates the 2nd BBL and 3rd BBL, as we cannot see the entry point between them. Finally, once the external loop is over, the 5th BBL is equal to the 5th RBL, ending the function.

Although there is an overlap between the characteristics detected for each code portion, as the information is to be stored into the BTB, we have precise information for each block following a branch. In the example, the 1st RBL is only going to execute once,

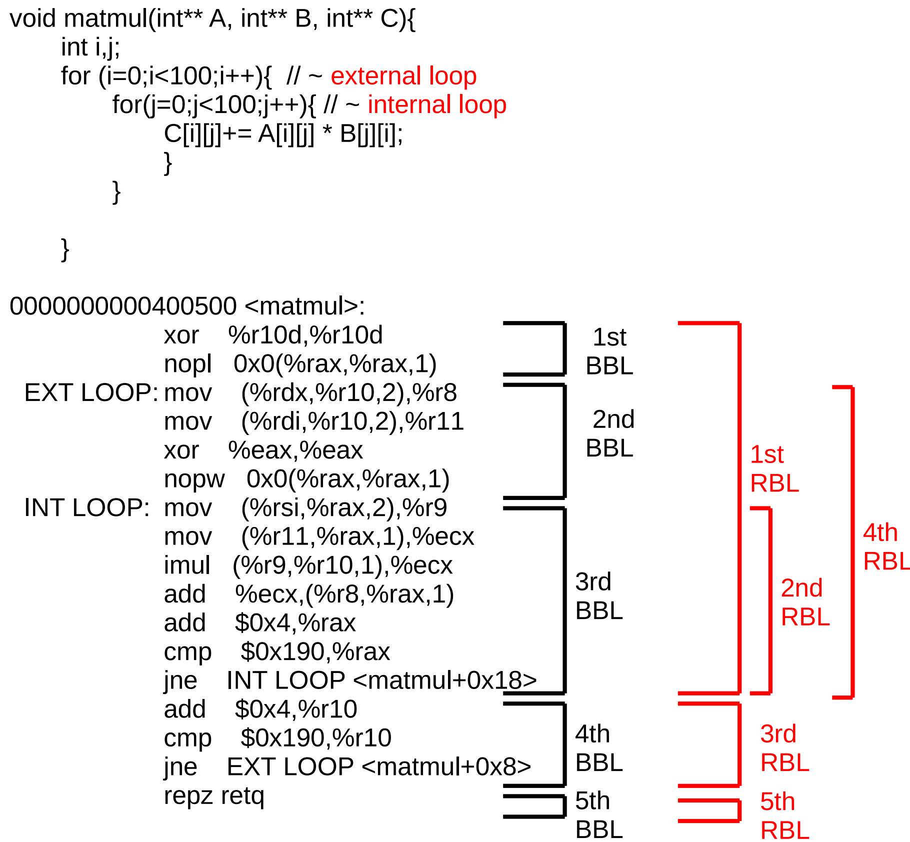

Figure 2.1: Example of code presenting the classical definition of basic blocks (BBL) and our relaxed definition (RBL)

while the 4th RBL is going to execute 100 times, and the 2nd RBL, the code that is truly repetitive, will execute 10000 times.

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 as the repetition of blocks begins after taken branches. Since we cannot recognize branch targets unless their respective branch occurred, we are breaking the definition of basic block, as we will likely record blocks with overlapping information. These blocks will aggregate behavior from all the instructions of the few, smaller real basic blocks inside them, and thus will not be characterized separately. However, the smaller basic blocks will be correctly characterized once they are targeted by a branch, thus obtaining their correct starting address. As in most cases smaller blocks represent conditions inside loops, they will be executed enough times to be characterized. If they do not, then they are likely not relevant.

2.2 Basic Block Characteristics and Performance

To motivate the behavior that can be observed per block and their correlation with performance, in Figure 2.2 we can see the behavior detection of the benchmark *libquantum* of the SPEC-CPU 2006 workload obtained through simulation. In the Figure 2.2, there are five bars showing the statistics of each block. These statistics all refer to the instructions inside the block (e.g. the branch misprediction happened for the branch within the block). The blocks are ordered in terms of number of executions, from most executed (1)

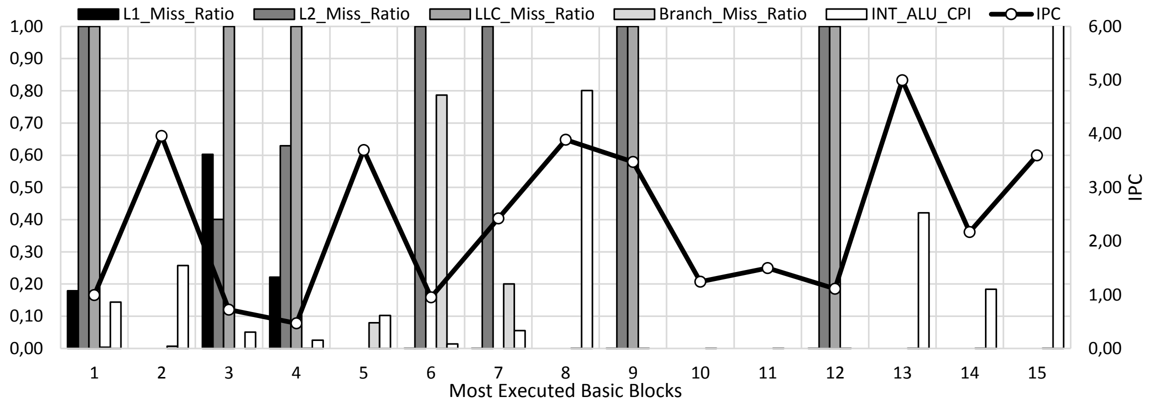

Figure 2.2: Statistics for the most relevant blocks of the *libquantum* benchmark, and their relationship with performance measured in IPC

to least executed (15).

The characteristics of the block clearly affect the performance on the majority of the blocks. As examples of blocks without characteristics, there are blocks 10 and 11. Likely, these blocks have different performances due to different code dependencies. In blocks 2, 8, 13, 14, and 15, contention in the ALU integer unit is present. Although contention should represent a problem for performance, we can notice that the blocks with this contention present higher performance than the average. This happens due to integer operations being simple and fast, and as the modeled architecture has 3 integer units, if there is contention, then it means that at least 3 instructions are being finished each cycle.

Blocks 3 and 4 show the worst performance. They also show bars representing a high data rate miss in all cache levels. In block 6, a similar effect occurs with branch mispredictions. If the architecture presented a better prefetcher, reducing the data miss rate, we could obtain improved performance for blocks 1, 3, 4 and 12. Alternatively, a prefetcher could use information regarding the poor performance of these blocks in order to prioritize them. This is the main motivation for our work, which is to improve or enable other mechanisms through observation of the block behavior.

To further motivate the behavior that can be observed per block and their correlation with performance, we used statistical correlation.

2.3 Correlation between Characteristics and Performance

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) with performance, using the Pearson Moment-Product Correlation Coefficient. This is a generalization of the linear regression model, 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 (i.e. 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 6. To calculate the correlation, we generated a profile of the execution. This profile contained the most important processor events relevant for execution performance (WALL, 1993)

for each block: L1 data cache misses, L2 cache misses, Last-Level Cache (LLC) misses, branch mispredictions, number of floating point arithmetic-logic instructions, and number of floating point division instructions. Whenever a basic block finished executing, we recorded the number of instructions the block contained and how many cycles it took to execute, in order to measure its performance. We then recorded how many of the events happened during the execution of that block. For fourteen parallel application from the NAS-NPB and SPEC-OMP2001 benchmark suites, each correlation coefficient was calculated considering blocks from all the threads together.

The correlation results are shown in 2.1 and Tables 2.2. The Tables show sequential benchmarks (SPEC-CPU2006) and parallel benchmarks (NAS-NPB and SPEC-OMP2001), respectively. The highest correlation coefficients for each benchmark are marked in bold. Looking at the cache misses correlation coefficients, we can observe a diminishing correlation as we go from smaller and faster to slower and larger caches. Although a miss in the LLC means a main memory access, which is likely to stall the processor, the number of accesses the LLC receives is small, because most accesses are serviced by higher level caches. Therefore, although a single LLC miss has a considerable impact on the final performance, it happens much less frequently than L1 and L2 misses, such that it does not correlate highly with the performance differences between blocks. We have noticed that, due to the large number of varied block executions, all correlation values are low. Were we to first partition the blocks into specific block types and run correlations individually, we would find much higher values. However, this analysis still serves the purpose of showing, for the entire program, which is the most relevant bottleneck of the program. A low correlation value will simply mean that the problem does not present many gains for the entire benchmark execution, but if we solved the specific problem of each block, we would certainly improve the overall execution.

Although a branch misprediction in a pipeline with 15 stages results in flush latency and a large number of stalled cycles, for parallel benchmarks, the other instruction types correlations seem to significantly diminish the branch instruction correlation. We found this low correlation coefficient happens due to a low branch misprediction rate, smaller than 1% in most of the parallel benchmarks. In the sequential benchmarks, we can observe higher branch misprediction correlation coefficient values for several benchmarks, denoting that the small memory pressure of a single thread alleviates the issues with memory for many benchmarks allowing branch misprediction to have a larger impact on program performance.

Floating points instructions per block correlate well on a few benchmarks. For parallel benchmarks, we can observe that for Apsi and Mgrid, floating point ALU instruction count is the statistic that correlates the most with degraded performance. In sequential benchmarks, the same happens for astar and hmmer.

Following this analysis, we seek to improve the memory access bottleneck. Therefore, in the next Chapter, we overview the related work that led us into our research and the state of the art regarding use of block profiling and memory controller improvements.

Table 2.1: Pearson moment-product correlation coefficients of absolute statistics per block and performance in IPC for sequential benchmarks.

Benchmark	**L1D Misses**	**L2 Misses**	**LLC Misses**	**Branch Mispred.**	**FP ALU Inst.**	**FP DIV Inst.**
astar	-0.185	-0.186	-0.066	**-0.209**	-0.003	0.000
bwaves	-0.026	-0.006	-0.002	-0.015	**-0.168**	-0.088
bzip2	-0.073	-0.117	-0.011	**-0.308**	0.000	0.000
cactusADM	-0.349	**-0.452**	-0.230	0.038	-0.036	0.063
calculix	-0.092	-0.051	-0.016	**-0.253**	0.096	-0.010
dealII	-0.090	-0.073	-0.014	**-0.158**	0.036	0.000
gamess	-0.052	-0.015	-0.004	**-0.076**	-0.062	-0.025
gcc	**-0.371**	-0.341	-0.157	-0.011	-0.225	-0.135
GemsFDTD	-0.187	**-0.216**	-0.089	-0.185	0.118	0.000
gobmk	-0.009	-0.015	-0.011	**-0.343**	-0.006	-0.003
gromacs	-0.223	-0.068	-0.008	-0.201	0.001	**-0.333**
h264	-0.065	-0.063	-0.013	**-0.129**	0.006	-0.002
hmmer	-0.077	-0.058	-0.006	-0.048	**-0.359**	0.002
lbm	-0.200	**-0.651**	-0.148	-0.031	-0.219	-0.087
leslie3d	**-0.413**	-0.356	-0.143	-0.049	-0.174	-0.325
libquantum	**-0.517**	-0.374	-0.078	-0.143	0.000	0.000
mcf	**-0.374**	-0.309	-0.064	-0.305	0.000	0.000
milc	-0.312	-0.299	**-0.245**	-0.004	0.028	0.000
namd	-0.007	-0.019	-0.002	**-0.068**	-0.033	-0.016
omnetpp	-0.224	**-0.236**	-0.051	-0.151	-0.132	-0.002
perlbench	**-0.140**	-0.095	-0.010	-0.108	0.002	-0.001
povray	-0.102	-0.022	-0.003	**-0.128**	-0.029	-0.113
sjeng	-0.056	-0.051	-0.018	**-0.235**	0.000	0.000
soplex	**-0.242**	-0.204	-0.072	-0.170	-0.066	-0.152
sphinx3	**-0.270**	-0.177	-0.028	-0.023	0.097	-0.035
tonto	-0.041	-0.035	-0.003	**-0.086**	-0.085	-0.041
wrf	**-0.161**	-0.128	-0.056	-0.031	-0.141	-0.114
xalancbmk	**-0.254**	-0.087	-0.033	-0.113	-0.025	-0.005
zeusmp	-0.116	-0.110	-0.034	-0.090	-0.112	**-0.164**

Table 2.2: Pearson moment-product correlation coefficients of absolute statistics per block and performance in IPC for parallel benchmarks.

	Benchmark	L1D Misses	L2 Misses	LLC Misses	Branch Mis-pred.	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

3 ANALYSIS OF THE STATE-OF-THE-ART

The common goal of basic block profiling is to enhance code execution by obtaining knowledge about its behavior in its different phases. Most approaches are done in software, through use of instrumentation tools to obtain profile information during the program's execution (LUK et al., 2005). Such method has an overhead due to additional instructions, and the very pollution generated by this overhead in registers and caches reduces the reliability of performance measurements done by such approach. Nevertheless, this approach can still correctly approximate the behavior of program phases. Its main problem is the need to execute a program multiple times with the same input, which makes any improvement achieved for single executions negligible.

State-of-the-art compilers can also use input sets to simulate a program execution and obtain profiling information, thus optimizing the code after some executions (LATTNER; ADVE, 2004). This approach can obtain performance improvements given a training input that is generic enough, i.e. which fully explores the code's execution paths. However, the bias over the necessarily exploring all execution paths may degrade performance for several input loads which only stress specific program execution paths. Alas, the need for recompilation among different architectures and systems always counts as a major factor in industry decisions to adopt a new technology.

Finally, the option of detecting basic blocks in hardware has intuitively been considered difficult to use. Most of the works which seek behavior characterization and reuse in hardware use coarser granularity for the sets of instructions analyzed in order to achieve performance gains. Such coarse granularity misses on opportunities for performance gain due to loss of specific blocks behavior information that is averaged among several blocks (PADMANABHA et al., 2013). This argument can also be used against basic blocks: it is a coarse granularity when compared to instructions. Nevertheless, individual instruction types are quite different, and cannot be compared with each other for any other purpose than hardware design analysis. Moreover, basic blocks have been proven to be a better granularity mostly due to code layout pattern detection (COCKE, 1970). As basic blocks identify the code units of repetition due to their direct relation with branches, they are intuitively good for the behavior analysis of different execution path phases.

The remainder of this section describes the works which serve as base to understand the potential usage of the mechanism, and then exemplifies hardware design mechanisms that can be improved with it.

3.1 Code Behavior Detection and Use

Sherwood et al. (2001) is the precursor to SimPoints (HAMERLY et al., 2005) and other works, such as Pinpoints (PATIL et al., 2004). The authors characterize the behavior of entire programs based on the analysis of basic block execution distribution. The concept of a *Basic Block Vector* (BBV) is first introduced to characterize a program. A basic block vector contains execution counts for all basic blocks, normalized by the total number of basic block executions. Therefore, the vector gives the execution frequency of each basic block proportionally to the entire program slice observed. The authors are then able to compare the behavior of executions of different sizes, for different inputs. In order to do so, a BBV comparison method is created by treating each BBV as a fingerprint of the observed program slice. To compare them, 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 do not execute any block in common.

With this comparison, the authors show a variety of features of their technique. They are able to identify the different phases of a program, such as the initialization phase of a program, due to the considerable difference between the BBV obtained for the first 100 million instructions and the BBV of the entire program. With the BBV of the entire program, they are also able to identify or create BBVs that have identical fingerprints, but a much smaller number of instructions. They show that the behavior of selected program slices with similar BBV are practically the same, with statistics pertaining to cache misses, branch mispredictions and overall type of instructions executed differing at most by 3%. This was further improved in SimPoints, which can use the Pin instrumentation tool (LUK et al., 2005) 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 (PATEL; LUMETTA, 2001) has a similar concept to our work. In this work, the authors use an extended definition of a block called a *frame*. A frame aggregates several basic blocks, as it ignores unconditional branches, and promotes easily predictable branches into assertions (PATEL; EVERS; PATT, 1998). They also provide a scheme to replay a frame in case an assertion fires, which signals a misprediction of an easily predictable branch which was promoted. In this way, they achieve a coarse granularity, enabling dynamic code optimizations during execution, and alongside the rollback mechanism, the opportunity for aggressive speculative techniques, such as value prediction and value reuse (PILLA et al., 2004). Although the framework is described, it is not explored in the paper. Frames intuitively have few opportunities for value reuse and value prediction, as they are coarse enough to capture several executions of loops, and seem to represent distinct phases of data progression in programs. The authors only show manual optimizations of single frame examples, and do not show any mechanism that can make automatic run time optimizations using the frames collected. In our work, we characterize application behavior on a finer granularity, so we can better inform other mechanisms.

The recent work of Kambadur et al (2012) also uses a simple profiling method called Parallel Basic Block Vectors. It can be seen as an extension of Sherwood's work, as now each entry in the BBV contains n positions, each representing the degree of parallelism at which the basic block is executed. 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 how frequently every basic block execute at each parallelism level, clearly identifying sequential and parallel code blocks. They also identify the most critical regions of code in terms of performance. Several scenarios are illustrated to show how this analysis can be applied, such as serial and parallel application partitioning, or analysis of program features by degree of parallelism and parallelism hotspots.

3.2 Basic Block and Phases Use Cases

In Panait et al (2004), the authors statically classify load instructions based on several heuristics. These include operations used for calculation of target address, registers used in address calculation and execution frequency. The work defines the *delinquency* of each load, pinpointing the instructions that are responsible for most of the data cache misses during execution. With static analysis, the authors are able to select only 10% of the total load instructions as responsible for more than 90% of the level 1 data cache misses. Using basic block analysis alongside the compiler, they reduce this number to 1.3% of all load instructions, responsible for 82% of all level 1 data cache misses. In this way, the efficiency of basic block analysis is shown by identifying blocks with loads that do not fit prefetch patterns or have overall low temporal or spatial locality.

Ratanaworabhan et al (2008) presents a new concept to detect program phase transition. Their phase transition is defined by critical basic block transitions. A critical basic block transition occurs when a rarely executed basic block is executed, signaling the change of behavior in a branch instruction. This usually signals a change of program phase, and thus different behavior. By observing critical basic block transitions, the work aggregates code into ten million instruction phases, and offers insight into different methods to detect critical block transitions, reusing the concept of BBV from Sherwood et al. to find the rarely executed basic blocks. It then uses phase information to adapt cache sizes to each phase. This shows the effectiveness of dynamic cache sizes when compared to optimal static size selection, as it reduces average cache size by 15%.

The work of Padmanabha et al (2013) proposes a predictive trace-based switching controller, which predicts an upcoming phase change in a program and preemptively migrates execution to an appropriate core in order to reduce energy consumption. In reality, the work uses a single heterogeneous core composed of a single frontend and 2 back ends. Each back end is based on the different organizations found in ARM's big.LITTLE architecture (PETER GREENHALGH, 2011), and what the authors describe as execution migration is actually selecting which back end will be used. Therefore, the only overhead is the communication of the register file's contents for correct context execution. With such a low overhead, migrations can be done often. Thereby the authors explore fine switching granularities, where they observe that these finer granularities offer more opportunities for energy savings. At the granularity of 300 instruction, the average time spent on the Little back end constitutes 28% of the execution, while targeting maximum performance degradation of 5%. This leads to an increased energy savings of 15% in comparison to running only on Big, representing claims of 60% improvement over existing techniques.

3.3 Hardware Design Opportunities

The main targets of current processor design research are branch prediction and memory access, as these issues are responsible for the greatest bottlenecks in modern super-

scalar designs. Although many other design points are becoming increasingly important, such as energy consumption, and others have been explored to some extent, such as value reuse (PILLA et al., 2004), branch prediction and memory access remain constant problems across decades. Therefore, it is of our interest to obtain information on the state-of-the-art mechanisms used to treat or alleviate these issues. We focus on memory access as it has the highest correlation with performance.

One of the methods available to improve memory access is prefetching. Prefetching is a technique that improves memory access by predicting future accesses and making requests ahead of time, generating better memory throughput and increased instructions per cycle due to lower average memory wait time. However, not all memory patterns are easily detectable, making the technique sometimes useless. Moreover, the technique can degrade performance by incorrectly predicting a pattern and generating useless prefetches. This results in cache pollution and unnecessarily increased memory pressure. To alleviate these aspects, Srinath et al (2007) create a mechanism to control prefetching aggressiveness. Every cache line is increased by 2 bits to characterize which lines were prefetched from main memory and which lines were touched. At the eviction of a line, if a line was prefetched and not touched, it falls under the case of pollution. With heuristics and mechanisms to insert prefetches on the last positions of a LRU policy (JALEEL et al., 2010), the authors propose tests using an adaptive stream prefetcher (HUR; LIN, 2006), which uses feedback from the cache lines to change the prefetcher aggressiveness configuration. Let it be noted that the technique can be used for any kind of prefetcher based on configurable parameters. In these results, performance of benchmarks from the SPEC-CPU 2000 benchmark suite increases by 6.5% on average when compared to the best performing static prefetcher configuration, while consuming 18.7% less memory bandwidth. Compared to a prefetcher using the same bandwidth, feedback directed prefetching provides 13.6% improved performance.

The work is superseded by Lee et al. (2008), who implements the same methodology to obtain prefetcher pollution data, but to avoid the slow pace of aggressiveness change, instead drops prefetches when detecting high pollution levels. This technique, known as Prefetch-Aware DRAM Controller (PADC), is able to obtain better improvements with a specific memory controller policy, with smaller hardware overhead. We used Lee's paper concepts to create a new memory controller using BLAP's information, and so we offer greater detail on their implementation in Chapter 6.

These works, along with Zhuang et al. (2007), ensure that no prefetcher ever degrades performance by reducing aggressiveness, even disabling all prefetch requests if necessary. These papers are crucial for prefetchers, as the increased memory pressure coming from multiple cores has been a main point of consideration regarding whether prefetchers should or should not be used. Generally, their performance improvement outweighs their performance degradation, even for large numbers of cores. These works use an interval to detect prefetching pattern, a fixed period of 8192 level 2 cache useful block evictions which is detected by the access bit of the lines. This is a valid approach to detect a program's memory phase change, but it does not detect a program's execution phase change.

In Ghose et al. (2013), the authors work on a memory controller policy capable of prioritizing critical loads. To identify critical loads, they develop a mechanism that stores information over the stalled cycles each load instruction generated in the commit stage. This characterizes how long each load takes, and with this several policies are possible. They explore binary policy, giving priority to loads which are currently stalling the commit stage and using a prediction table to give priority to loads that have stalled it before.

Moreover, they explore the usage of the cycles that a load stalls by creating four more policies. As we used this work for comparison, more details are given in Chapter 6.

The work's idea is solid, but the methodology shows some flaws. Prioritizing loads ahead of time ends up giving priority to loads that are already in the cache, or are easily predicted by prefetcher. Moreover, the aliasing of load PC addresses in the tagless table can lead to prioritization of irrelevant loads.

The authors used a processor frequency predicted to be present in the future, with a 4.27 Ghz frequency, while modeling a current DDR3 main memory at 1066 Mhz frequency. Therefore the memory pressure becomes much higher and the relevance of reordering loads in the memory controller also higher than it should be. We consider this to be a valid assumption, as the access times of memory accesses are likely to increase in the future. We use this idea in our work, as it can be implemented in coarse granularity by giving priority to all loads in a block. Additionally, we keep track of block behavior and ensure that a block has stable behavior before changing priorities. We also make a deeper study around the benefits of such approach in the presence of prefetchers.

3.4 Summary of the State of Art

Overall, most of the existing works in hardware to improve performance or energy consumption use coarse granularity phases. They do this given several constraints of their own implementations, such as migration overhead or detection overhead.

However, most works do not take into account that the branch target buffer structure contains information regarding all branch targets. These are the instructions that begin basic blocks. Detecting basic block execution is possible given relaxation of the definitions, and as seen in the related work, it should provide good granularity to detect program behavior.

Following the work of (PATEL; LUMETTA, 2001), we take a new direction by using the behavior detected to adapt existing hardware mechanisms, namely (GHOSE; LEE; MARTÍNEZ, 2013) and (LEE et al., 2008). In the next Chapter, we present all detection mechanisms that led from related work to the final mechanism, presented in Chapter 5.

4 BLOCK CHARACTERIZATION

Basic block profiling is a frequently used technique in compiler and post-compilation steps. It aims to characterize basic blocks in some aspect (LATTNER; ADVE, 2004). In this Chapter, we explore methods to perform block profiling at the hardware level.

4.1 Introduction

Our proposed method to perform block profiling in hardware builds an evolving profile of a program's code. Thereby, it is possible to improve frequently used blocks by characterizing them and improving their performance in future executions. In the following sections, the profiles generated by the mechanism in this work record the types of instructions that are responsible for the largest delays in each block. However, the basic concept of the mechanism can be extended to implement a variety of techniques based on block profiling. In this section, we base ourselves on the related work to discuss common techniques to characterize code behavior, and how we can adapt these techniques to the basic block case. This section shows a progression towards accuracy and simplicity, in order to achieve a useful technique which could be implemented in hardware. This section only cares about behavior detection. Behavior storage and its information use are explored in later sections.

4.2 Hardware Counter Classification

Currently, Intel processors come with hardware counters available to enable performance profiling and analysis, used by tools such as Vtune (REINDERS, 2005). These counters keep track of several events during processor execution, such as cache misses, operations per type, to the point of detailing even bus transactions. Such statistics intuitively correlate to performance, and they can be used to provide insight on what issues might result in suboptimal performance of a code snippet.

However, to do so at the basic block level raises a few issues. The first issue is identifying and delimiting blocks. As our relaxed block definition uses branches and jumps as blocks' last instructions, we can identify branch and jump commits to mark block endings. We then store the hardware counter values and reset them, so they start collecting information for the next block. If we assign a weight to each counter in order to properly characterize each instruction given its performance overhead, we get the block distribution seen in Figure 4.1. The figure shows the execution of a simulation that prints, at every block end, which weighted characteristic had the highest value for that block. We assigned the following weights for hardware counters: 8 for instruction cache misses, 8

for data cache misses, 32 for level 2 cache misses, 200 for last level cache misses and 120 for branch mispredictions. All functional unit stalls were already counted as cycles, and thus had weight 1. In all of our tests we eliminated statistics that did not show to have a relevant number of occurrences or delays, such as integer multiplication, integer division, and instruction cache misses. Their occurrences accounted for less than 1% of all blocks. "Others" denote none of the registered characteristics were found in the block, so any stalls result from register real dependencies or were simply not detected.

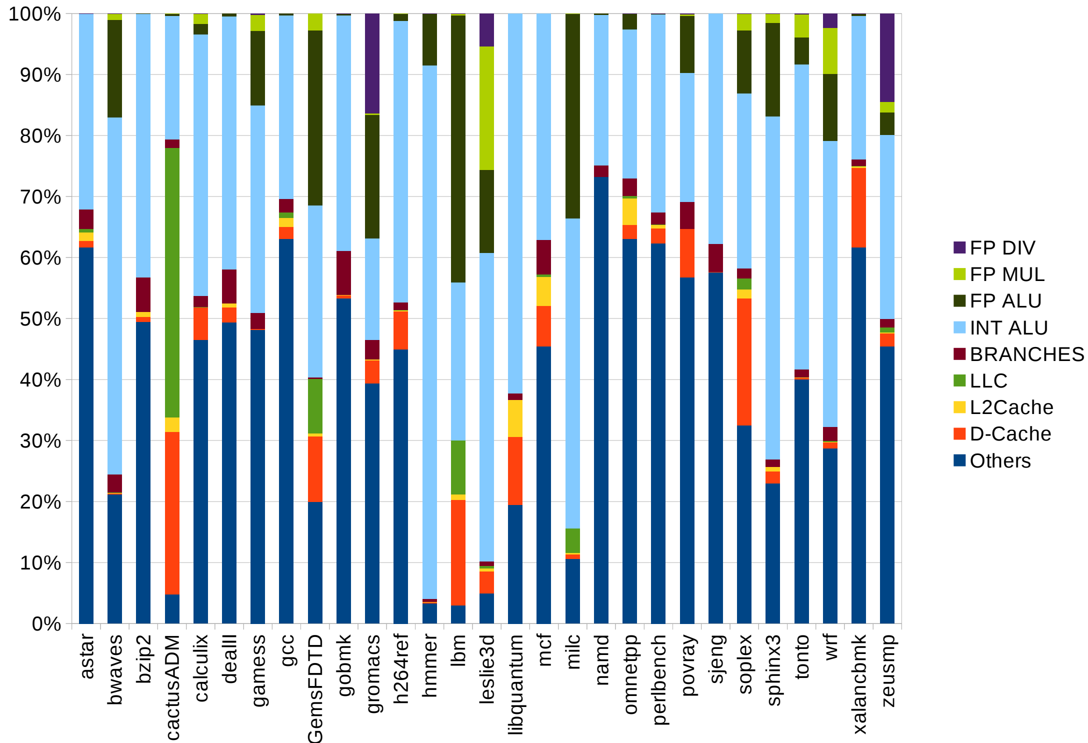

Figure 4.1: Basic block characteristic distribution. Every block receives only one characteristic, the most relevant one according to the hardware counters

These results show us the second issue. As we are using superscalar architectures, the branch commit contains all the information of what was processed since the previous branch commit; the in-order commit stage ensures that. However, since new instructions do not wait for a branch commit to happen, they can enter execution after branch "A", change some of the statistics (such as data cache accesses), and be stored as information of the block that finishes at the branch "A", i.e. the *previous* block information. Therefore, we now have information that does not belong to this block, resulting in its pollution. This also means the next block will be lacking this information, and it may also grab information from the block ahead of it. We denote this phenomenon as *information skew*.

This would not be a problem if all statistics collected to characterize a block were obtained within two branch instruction commits. As the skew happens for every block, we would have a fixed skew for every block, and a fixed characteristic skew. However, statistics such as level 1 data cache miss and last level cache miss take different times to be recorded. This generates variable pollution for each block, which is the main challenge in implementing a real mechanism.

To avoid this, in our simulation we created a table to store statistics per in-flight branch. Whenever a new statistic must be recorded, we search the reorder buffer for the first conditional branch instruction that would come after it, and increment the statistics for that branch entry. Then, when the branch commits, we get the statistics from this table, ensuring no information skew occurs. Due to the large size and difficulty of hardware implementation, we believe that creating such a table in hardware and checking for ordering would offer a mediocre tradeoff. Nevertheless, the characteristics of this distribution are shown in Figure 4.2.

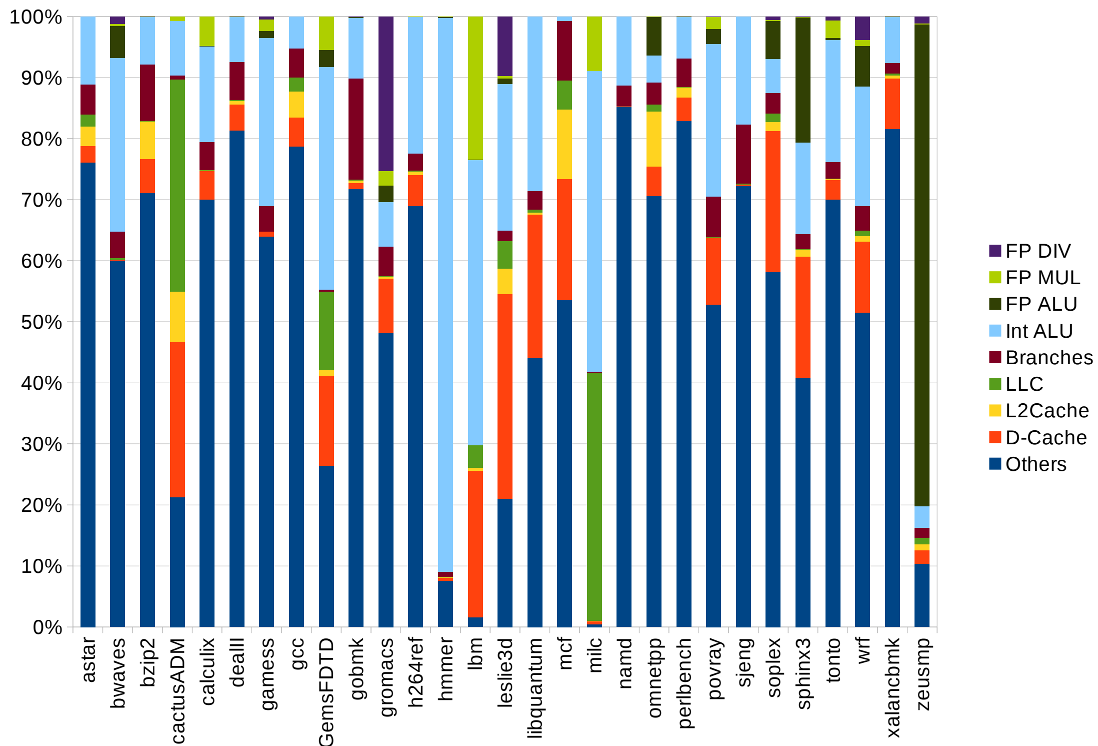

Figure 4.2: Basic block characteristic distribution. Every block receives only one characteristic, the most relevant one according to the hardware counters. Now statistics are properly attributed to each block.

Although there were several changes in the characteristics attributed to each block, the main problem with both classifications is that many blocks do not receive any characterization. In order to achieve characterization for more blocks, we must use information of greater detail, which led us to seek the latencies generated by register dependencies.

4.3 Register Dependence Latency Classification

The basic, initial number of delayed cycles caused by a single instruction A is the one measured by how many cycles another instruction B, which is dependent of instruction A's result, was delayed. To implement this mechanism we observe the latencies generated by each register dependence (we do not take into account structural hazards, i.e. two unrelated instructions which need the same functional unit). A table is kept with an entry for each characteristic. When an instruction is blocked due to register dependencies, we

count how many cycles the instruction takes to receive the register value which it needs. The instruction A, which was using the register, is accused of generating the delay, and thus the characteristic of such instruction type's entry is incremented by that number of cycles.

Although the accused instruction A might have suffered with delays from other instruction C, instruction C would also be accused for such delays, as it was stalling A first. This warrants we find the culprits for the largest stalls in the critical path of the application. We can observe the resulting basic block distribution in Figure 4.3. As we do not know the level of cache that was able to provide the correct line for loads and writes (considering caches with write-allocate policy), we measure the number of cycles delayed and classify them into the three levels. If the delay is smaller than the minimum number of cycles necessary to access to the level 1 cache, we classify the delays as memory loads and memory stores, representing latencies on the load-store queue.

An odd observation is that memory stores are taking up a portion of the benchmarks, such as in the *dealII* benchmark. In Sandy Bridge's pipeline, once a write has been sent to memory, there is no need to wait for registers, as they do not store values at any register. After some research, we found out that Pin considers the *CALL* instruction as store, as the instruction writes values to the stack and increases its size. However, it also generates register dependence for the PC and stack pointer registers, therefore generating dependence for other instructions. Even with the prediction of the PC value in the branch target buffer, instructions which require the stack pointer, such as the instructions that must obtain the parameters of the function in the stack, would theoretically still be stalled. To remediate this, Intel uses a "Stack Engine" in the front-end, which keeps a copy of the stack pointer updated to free dependencies for every *CALL*, *RET*, *POP*, *PUSH*, and other instructions which implicitly use the stack pointer.

This mechanism was modeled in simulation, but deemed too complex for hardware implementation, as in hardware there is no efficient way to trace back which instruction generated the delay. Even if we built such a mechanism, to assign instruction types to registers in order to check for dependence delays, this would likely generate larger delays per stages in the middle of the back end, which is not desirable.

Alas, the number of blocks characterized as *ALU integer* is large likely due to minimal stalls characterizing the block, covering the area that was presented as *others* or *none* in the previous distributions. Nevertheless, integer operations do not cause any real harm to performance, as the unit's latency is minimal. Even though register dependencies delay instructions, the overall throughput can still be high, as long as the hindrances do not stall the commit stage, which takes us to the next idea.

4.4 Stall Commit Classification

Rather than checking for indirect and intermediary effects on performance, a simpler way is to observe the delays generated by the instructions at the final point: the commit stage. If an instruction delays the commit stage, it means that it will affect throughput, as the commit stage is in-order, and being blocked by one instruction means blocking all instructions that could be committed in the same cycle. This is a rather simple way to observe delays, as we do not see the indirect delays that register dependencies generate. If instruction B was greatly delayed by instruction A, instruction B will stall the commit stage and we will accuse its delay. However, our intuition is that instruction A will likely stall the commit stage before instruction B does, and for a longer time. The resulting

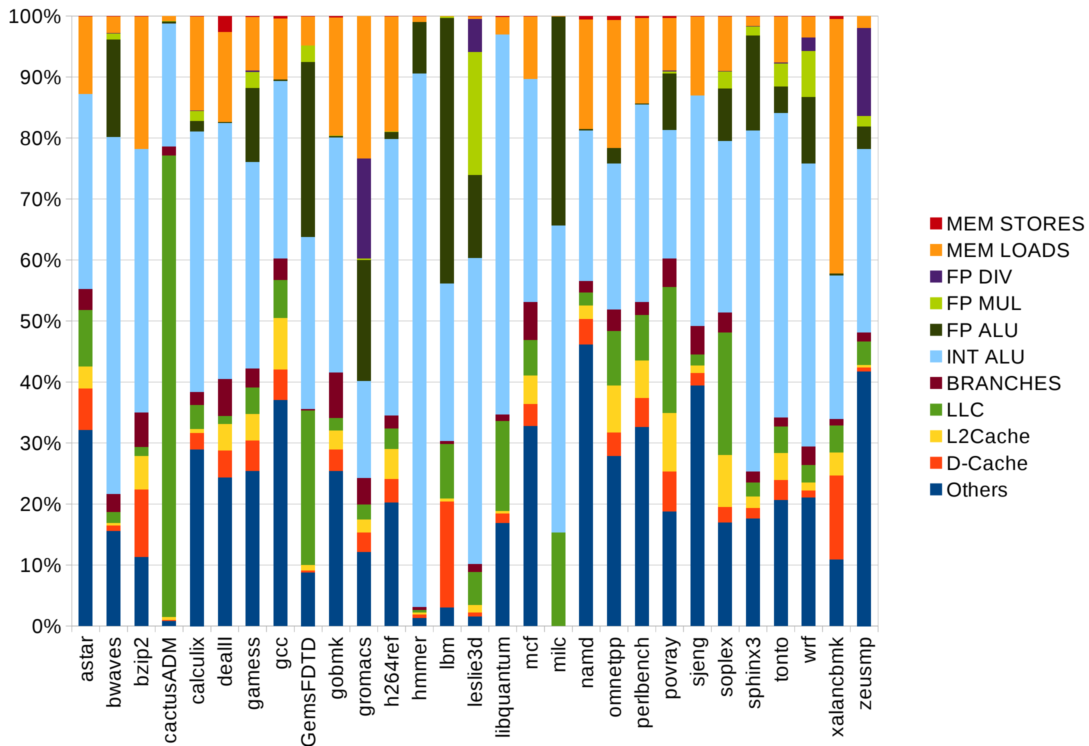

Figure 4.3: Basic block characteristic distribution. Every block receives one characteristic, the most relevant one according to the sum of register true dependencies delays per type

block characteristic distribution can be seen in Figure 4.4.

As seen in this Chapter, obtaining detailed hardware counter statistics per block during execution is a complex matter. As we aim to aggregate behavior and uniquely identify blocks, within a reduced storage size, using statistics gives us three challenges. First, a statistic must show a direct impact on performance. While cache misses are intuitively correct in expressing delinquent loads (PANAIT; SASTURKAR; WONG, 2004), current architectures are usually tolerant to L1D misses due to high ILP, which provides enough computation to overlap the cache access latency. That is, in most cases, L1D misses stall the processor for a small number of cycles.

Second, different hardware events cannot be directly compared. When a level 1 data cache miss occurs, we know that it will take at least the level 1 data cache access time plus level 2 cache 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 level 2. Even such a large latency could be hidden in the presence of a branch misprediction. If we want to find which 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 even 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, are ready to commit, and we

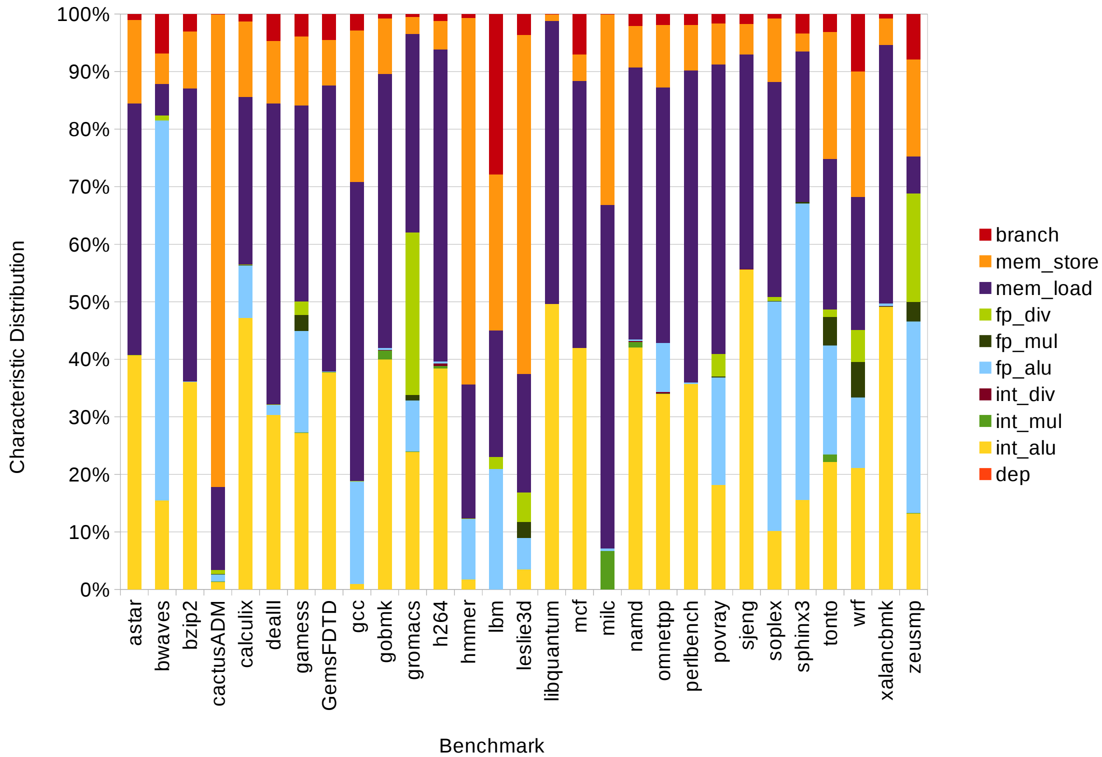

Figure 4.4: Basic block characteristic distribution. Every block receives only one characteristic, the most relevant one according to commit stage delays

obtained all their statistics, 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 opted to 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 at the commit stage stalls. We can directly compare the number of commit stage stalled cycles 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).

Overall, this is the better choice because it is simple to implement in hardware. As the operation code of the instructions can be obtained from their reorder buffer entries when they are being retired, no critical path increase is generated, as a simple wire extension can give us those signals. All the needed hardware for detection can be counted in a few registers to detect how many cycles each instruction stalls the commit stage. When using register dependencies or isolated hardware counters, we need intermediary storage to isolate the information per block, which is expensive with large reorder buffers and high ILP, making these techniques unfeasible. Furthermore, this technique only measures meaningful stalled cycles, as the throughput of the processor is usually observed by the number of instructions retired in the commit stage every cycle.

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 store 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 multiple characterizations, so multiple mechanisms can use the profile. In accordance with the correlation results and the characteristic distribution, we chose to record the following characteristics: *None* to denote that the block presents no problems, *Brch* to denote branches hard to predict, *Mem* to denote commit stalls due to loads and *FP* to denote commit stalls due to any floating point unit. We have chosen to use the *Brch* characteristic due to more tests which show the relevance of the characteristic influence in block executions.

5 BLOCK LEVEL ARCHITECTURAL PROFILER

In this Chapter, we show the detailed implementation of BLAP(Block Level Architectural Profiler) in hardware. BLAP consists of three parts: Behavior Detection, Behavior Labeling and Behavior Storage. After explaining each, we discuss potential critical path implications and how they can be avoided. We then list the hardware overhead costs these three stages and describe additions to further improve the profile information.

5.1 Behavior Detection

Modern superscalar processors use an in-order commit stage, so they can avoid memory access speculation (CRISTAL et al., 2004). This enables reading the decoded in-order stream of instructions of a program by looking at the retired instructions each cycle. With the instructions decoded, we can observe which instructions are conditional branches,

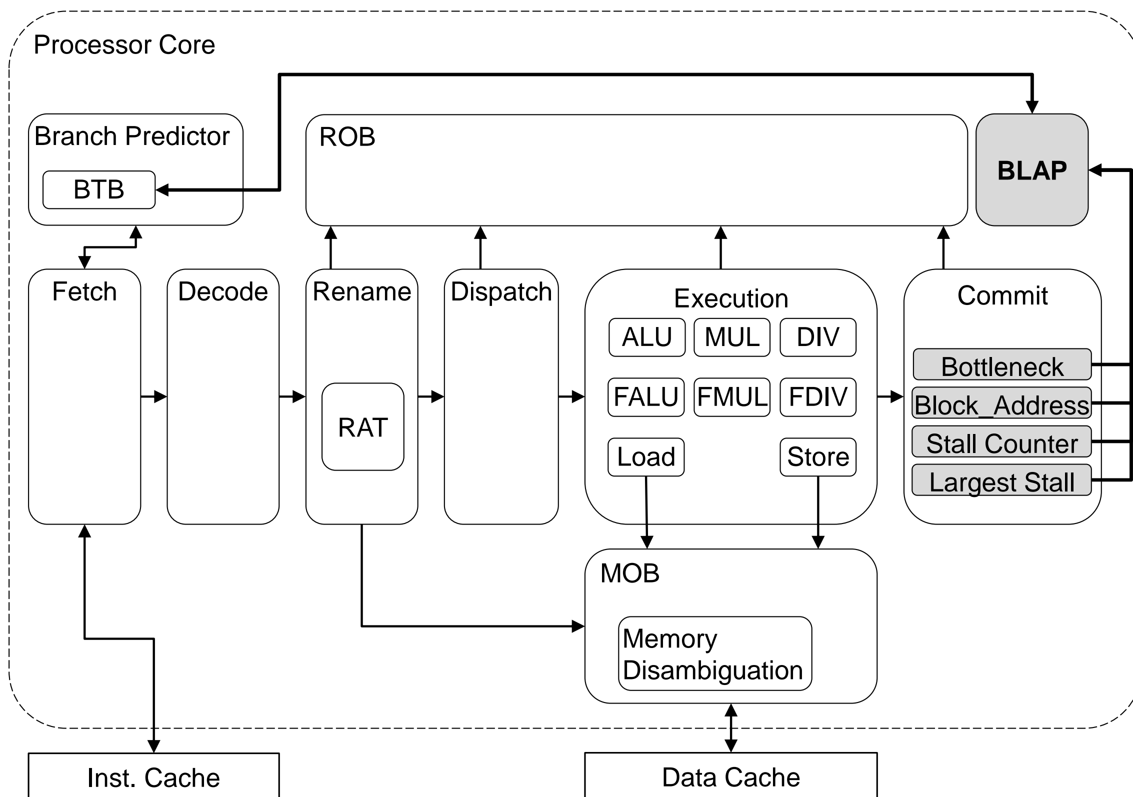

Figure 5.1: Overview of the operation of BLAP in a superscalar processor. Parts in gray represent BLAP's modifications or additions to the processor.

thus allowing us to detect ends of basic blocks. Whenever the branch target buffer is accessed by a branch, we can observe the beginning of a new basic block as branch targets are stored in the BTB.

To better illustrate the hardware needed for our mechanism, in Figure 5.1 we can see a common abstraction of a modern superscalar design with our mechanism additions in gray. The fetch, decode, rename and commit stages are all in-order. For every branch instruction, the fetch stage consults the branch predictor to speculate on the correct path to follow. This forces the fetch stage to be in-order, as the instruction path is conditioned by the branches present in the instruction flow.

The following substages which compose the decode and rename stages all perform simple, single-cycle operations on the instructions. The rename stage is in-order, as it needs to be aware of instruction order to keep track of real dependencies and eliminate false dependencies. The rename stage also passes instructions to the reorder buffer, while it keeps track of the register dependencies between these instructions. Thereby, the reorder buffer keeps all ordered instruction information, so the out-of-order stages can execute correctly.

The next stage, dispatch, begins out-of-order execution. It will dispatch any instruction to an available functional unit of the instruction type. Given different program behaviors, this means that a busy functional unit will delay its instruction type, while another instruction can be scheduled to another functional unit type in that cycle, thus resulting in the reorder of the instructions. Moreover, different functional units take different times to execute, which also affects the execution order.

The commit stage is responsible for retiring instructions from the reorder buffer in order. This stage is executed in order to ensure *precise exceptions*, which means that whenever an instruction generates an exception, every instruction previous to it must have finished, and nothing after it can have finished execution. In this way the software state is not changed by incorrect execution of instructions that could be affected by the instruction that generated an exception.

Further, an in-order commit stage also simplifies the rollback mechanism necessary for branch mispredictions when using speculative execution. As an example, let's consider that a branch prediction speculative path contains an instruction *I1* that changes a register value. With an in-order commit stage, whenever a branch's prediction is detected as a misprediction, the necessary steps to continue the correct execution are to flush the reorder buffer positions that come after the branch, flush all in-order stages of the front end (fetch, decode and rename), and then start fetching instructions from the correct datapath. In this way, the *I1* instruction will be flushed, and since it never committed its results, register values would not be changed. If this instruction was allowed to commit its results before the branch, a misprediction would imply that the branch would need to keep every register state, so it would know which registers were changed by the speculated instruction and would be able to recover their values.

In Figure 5.2, we show a flowchart of the additional events needed to implement our detection mechanism in the commit stage of a superscalar processor. BLAP implementation requires an in-order commit stage, which is widely used in current commercial processors. The Figure accounts events that must happen for all instructions, but a real implementation will only look for the reorder buffer head instruction and the first branch instruction it finds, as can be seen in the sequence of conditionals **3** and **4**.

In the commit stage, we must check whether the oldest instruction, at the reorder buffer head, is ready to commit **1**. Whenever an instruction stalls the commit, a *Stall*

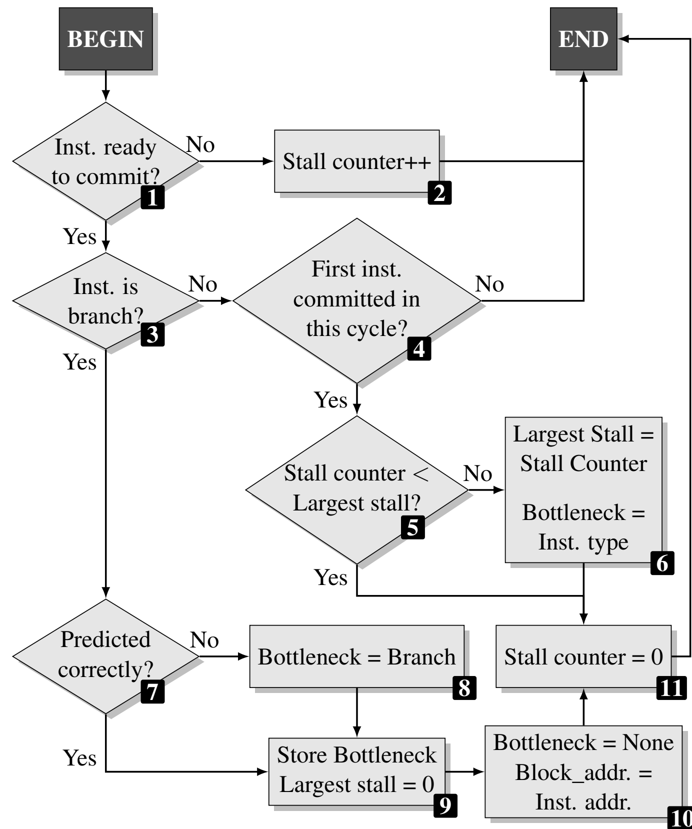

Figure 5.2: Flow chart of additional commit stage events.

Counter is incremented 2. When the instruction is ready to be committed, we must check if it is a branch 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 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 does it end a block. However, if this is the first instruction, we must compare its accumulated stall time with the previous largest stall time of the current block 5. If the stall is larger, we update the register keeping track of the bottleneck with the current instruction type 6, update the largest stall with the value of stall cycles present in the stall counter, and reset the stall counter 11. Otherwise, we skip the update and reset the stall counter at 11.

If the instruction is a branch (which can be obtained in the same cycle by extending the signals present in the reorder buffer entries to our scheme), we must store the block information. First, we check whether the branch was correctly predicted 7. This can be done by checking the branch history table (which takes 2-3 cycles) or carrying this information in the reorder buffer entries (done within the same cycle), but additional hardware, which is likely already present in the processor commit stage to detect mispredictions. Obtaining the signal from a reorder buffer entry only requires a wire extension from the signal (picoseconds), while selection requires more logic and time, although the combinational logic used by the commit stage to select the reorder buffer entries can simply be extended to obtain the extra bit signals which denote instruction type and whether a branch misprediction occurred. As branch instructions do not stall the commit stage, the only way to characterize a block as "Brch" is to directly access its accuracy information.

If the branch is not correctly predicted, we change the block's bottleneck type to

Brch **8**. We then store the bottleneck type in the BTB, using the value of the Block_Address register as index (the address of the branch that started the block) and store the instruction address of the current branch instruction in the Block_Address (as we are starting a new block) **10**. Thereby, 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 counters related to largest stall **9** and bottleneck **10**, followed by resetting the stall counter **11**.

In our experiments, the average relaxed block size varies between 5 and 10 instructions, while we have a commit width of 5 instructions.

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

5.2 Behavior Storage

In order to use the profile, we must store it for future use. Based on the correlation results, we use 2 bits per block, expressing 4 characteristics (*None*, *Brch*, *Mem*, *FP*). These characteristics were chosen because they either have shown the highest correlation coefficients or they had the greatest impact in terms of stalled cycles in our experiments. The number of characteristics can be incremented by using more bits per entry, to allow 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 every block targeted by a conditional or unconditional branch. In this way, when the branch target is predicted by the BTB, we also load the bottleneck characteristic of that entry as we know it is the profiled characteristic of the block about to be fetched. 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. Mislabeling can occur, as "not taken" branches will load the characteristic from a predicted taken path, when such mispredictions happen. As blocks with mispredictions after the training phase are rare (less than 1% of all predictions), we accept this mislabeling, since the mechanisms which use this information are all speculative and do not generate wrong execution.

5.3 Behavior Labeling

To use our profile information, we followed an approach that allows implementation of multiple mechanisms. When a branch is predicted at the fetch stage, we access the BTB using the address of the branch instruction. The characteristic is loaded into a new register called *Block Characteristic*.

The information of this register is copied to a new field for every entry of that instruction (e.g. the ROB) until the content of *Block Characteristic* is updated by the next block profile information. Thus, the fetch buffer's entries, the decode buffer's entries, and the ROB entries are all augmented by 2 bits to store the characteristic pertaining to the block.

5.4 Critical Path Implications

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

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. When there is conflict for the BTB write port, target address writes coming from branch execution units are given priority through the use of multiplexers to the BTB write port. The BTB has an additional bit to signal which part of the entry will be written(regular branch writing its target address or BLAP information). If new values from blocks would come to this new stage, we do not overwrite the current BLAP information waiting for the write port, as these new values are from blocks that likely were stalled for a small number of cycles.

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

5.4.1 Profile Stability

The behavior of a basic block is static given enough repetitions, but the scheme shown so far updates the characteristic every execution. Alas, the characteristic of a block changes quickly in the first executions, as caching, prefetching and branch prediction mechanisms get trained. To ensure that a block's characteristics are stable, we add 2 bits to each BTB entry. These bits count to 4 to ignore the first 4 executions, allowing time for a block behavior to become stable. Once the counter saturates, we register a block characteristic and do not change it until the block is evicted.

We were led to implement behavior stabilization because of implementation details detected in our tests. For instance, when increasing the aggressiveness of a prefetcher for a block detected with long load stalls, two cases could happen. The first case happened when the block did not actually have a *Mem* characteristic, and it should stabilize as another characteristic. In the second execution, we would increase prefetcher aggressiveness, and performance could be degraded due to too much memory bandwidth used, or improved. If performance was improved, there was no way to tell whether it happened due to caching, prefetching or increased aggressiveness on prefetching, so we would end up keeping a prefetcher more aggressive than it should be and degrade performance for other blocks. The second case, which happened when the block actually had memory problems, generated more problems. If we improved memory access via prefetcher aggressiveness or load priority bits, and the characteristic of the block was kept as *Mem*, that would be fine. But even though the block's stable characteristic was of long load latency, our implemented mechanisms would improve performance enough so that the detected

characteristic would change. This generated a "tic-toc" effect. Whenever we got a *Mem* characteristic on block A, we improved performance of block A via memory mechanisms, and got a different characteristic for it, such as *FPDiv*. Then, on the next execution of block A, as its characteristic was detected as *FPDiv*, the changes on mechanisms to improve memory performance would not be triggered, degrading performance back to characteristic *Mem*. The characteristics and the mechanism activation would keep alternating, when it is desirable that the changes made to improve memory access remained active the entire time.

5.4.2 Hardware Costs

To implement BLAP, we can divide the hardware costs into three parts: detection, storage and labeling. As can be seen in Figure 5.2, detection requires two 8-bit counters, Stall and Largest Stall. It also requires an 8-bit adder and a 8-bit comparator for these registers. We use two 2-bit Bottleneck registers, and two Block Address registers, to pipeline the actual write to the BTB with an additional BLAP stage. To determine whether a branch prediction was a misprediction, we add 1 bit for each reorder buffer entry (which may already be there to detect mispredictions and generate rollbacks in the commit stage). All entries and options of the datapath require two 2 bit 2-input multiplexers and one 8 bit 2-input multiplexer.

For storage, we modify the BTB write port to write the extra BTB bits. The value in BLAP's write buffer waits until no branch instruction has a branch target address to write in the same cycle, and one extra bit indicates the entry bits to be written (branch target address or BLAP information). If another block information coming from BLAP would overwrite BLAP's buffer while it waits for a write opportunity, we ignore the second block information as the stall value is likely low for such conflict to happen. Optionally, we can add another write port to avoid this issue. Thereby, we reuse all tags and logic from the BTB. The extension is composed of 2 bits for characteristics and a 2-bit saturating counter to stabilize each entry.

We store the labels in a Block Characteristic register when we obtain the information bits from the BTB in the fetch stage. Every following instruction of the block must copy this information, so we must add these bits to the entries of all structures. We add 2 bits to every entry of the fetch buffer, decode buffer and ROB. The calculated costs in terms of hardware are shown in Table 5.1. For each core, BLAP requires the total storage of 2142 bytes, plus combinational logic of three 2 bit multiplexers, one 8 bit multiplexer, one 8 bit adder and one 8 bit comparator. This is a significant amount of bytes, but using it as cache storage or extra branch target buffer targets would yield less than 1% performance improvements. The total area overhead per core is of 206164 transistors. In Sandy Bridge, that means 1236984 total transistors. As Sandy Bridge has 6 cores, Graphic Processing Unit (GPU) and large caches, if we consider the core areas to only take 25% out of the 2.27 billion transistors, our area overhead corresponds to less than 0.22% of the total cores area.

5.5 Evaluating BLAP Precision

In order to evaluate BLAP's precision and performance improvement potential, we performed a collection of experiments for the "Brch" and "Mem" characteristics. First, we tested baseline execution and upper bound improvements execution for each characteristic. The upper bound changes for every characteristic, as it is modeled upon eliminating

Table 5.1: Hardware Costs of BLAP

Mechanism Portion	Resources
Detection	8-bit Stall counter; 8-bit Largest Stall counter; 8-bit adder for Stall counter; 8-bit comparator (Stall counter with Largest Stall); 1 2-input 2-bit multiplexer, uses branch prediction information to update BLAP; 1 2-input 2-bit multiplexer, bottleneck evaluation and selection; 1 2-input 8-bit multiplexer, largest stall evaluation and selection;
	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); **Total of 316 bits for detection;**
Storage	1 2-input 2-bit multiplexer (selects regular BTB write or BLAP write);
	2-bit Bottleneck reg. per BTB entry (4096 entries in total); 2-bit Stabilizer counter per BTB entry (4096 entries in total); 1 4-bit write buffer; **Total of 2048 bytes of storage;**
Labeling	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;**

the latencies of the characteristic. Memory accesses are all solved within the LLC cache (eliminating the major latency of accessing main memory), while branches are all correctly predicted. The configurations used to obtain these results are detailed in Chapter 6.

After we defined these results, we ran tests eliminating these latencies only for blocks found within BLAP, and only for a certain amount of them. For these results, we used a 4096 lines cache. Our objective here is to first show BLAP's accuracy and performance improvement potential. We will use the BTB later on, with mechanisms defined for performance improvement. In the next figures, each bar shows the performance improvements compared to the baseline, with progression when modeling constraint elimination from 25% of the blocks detected by BLAP to 100%, with leaps of 25%. The rightmost bar shows the upper bound, achieved if we eliminated the latency for all instructions of that type.

In Figure 5.3, branch performance progression is shown. Following the correlation, performance is greatly dependent on correct branch prediction. If we take into account the figures shown in previous chapters, we know that branches represent a small amount of all blocks characterized, since the amount of branches mispredicted is relatively small. Yet, correctly predicting that small amount of branches yields great improvements, and being able to improve any of the branches detected by BLAP allows for a good amount of performance improvement. The issue is, as the progression shows, that BLAP has a hard time finding all the correct blocks. As branch prediction hit rate does not stabilize like memory access time, BLAP's stabilization scheme makes it impossible to detect unpredictable branches. An example would be a branch that alternates between correct and incorrect prediction back and forth, without repeating a correct or incorrect prediction.

In Figure 5.4, last level cache performance progression is shown. Here we also notice performance potential for the cache misses found within BLAP, as expected from avoiding

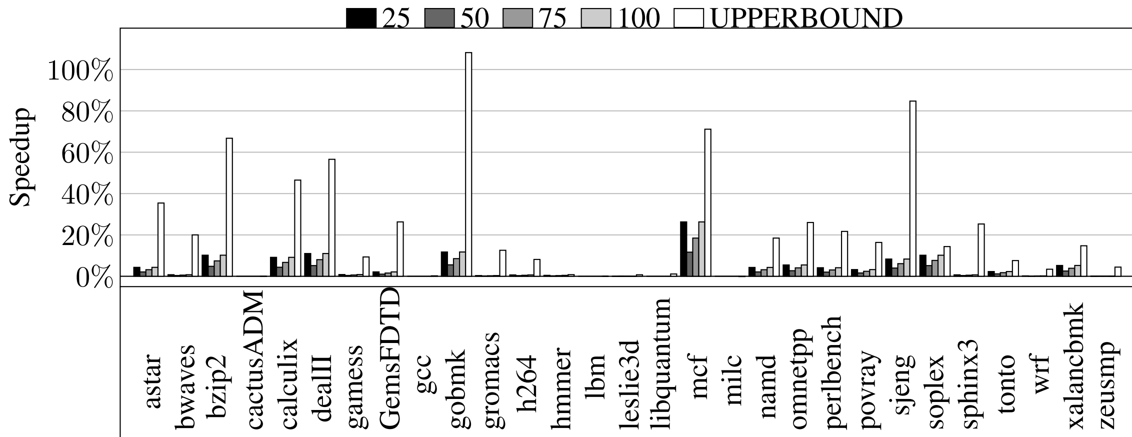

Figure 5.3: Modeled performance improvements correctly predicting 25%, 50%, 75% and 100% of branch instructions in blocks characterized as "Brch".

accesses to the main memory. We can notice that the cache performance is still distant from the upper bound. This happens due to characteristic aggregation. For instance, if a block has a large latency of 160 cycles for a load, but a misprediction also happens, the load instruction should not matter. Likely, its stall effect on general performance will be diminished due to the lack of instructions ready for execution and committing. In the cases of floating point functional units, these could hide small latency loads, which would hit L2 cache level and still be tolerated.

Thus, in Figure 5.5, we redo the experiment characterizing only for cache characteristic or no characteristic. Here, we can see small increases in performance, thanks to the elimination of branch characteristic in blocks that had last level cache misses as well.

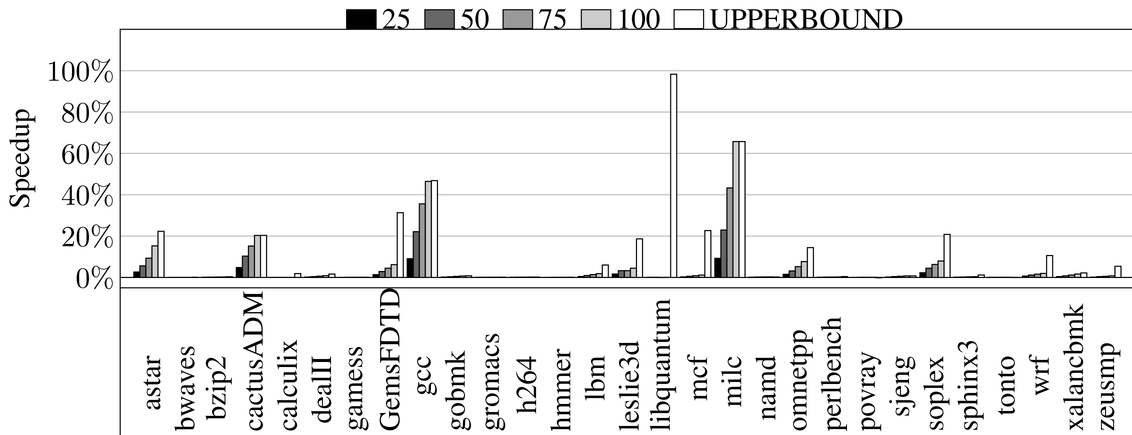

Figure 5.4: Modeled performance improvements solving 25%, 50%, 75% and 100% of the load instructions of blocks characterized as "memory load".

With these results, we can see a lot of potential for the memory bottleneck. In the next Chapter, we detail the architecture used and the related work that was considered as state of art to improve the memory controller.

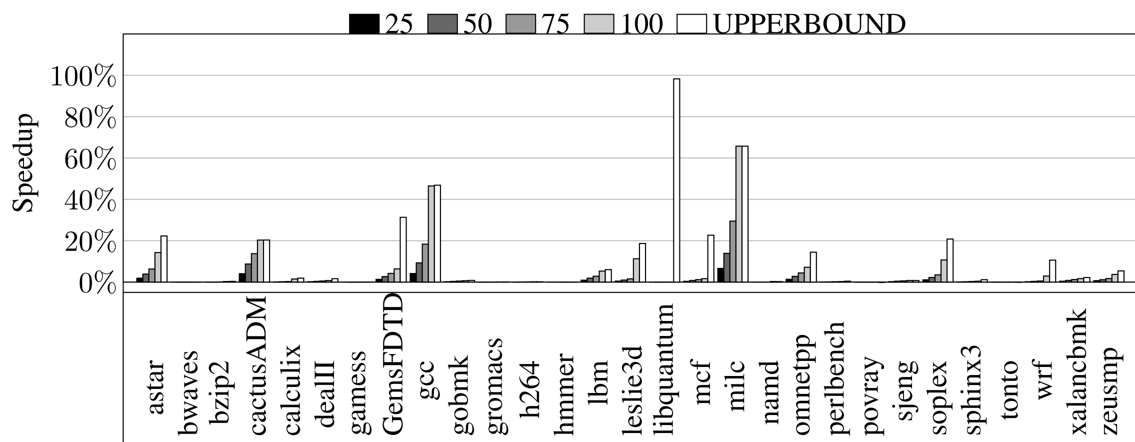

Figure 5.5: Modeled performance improvements solving 25%, 50%, 75% and 100% of the load instructions of blocks characterized as "memory load", when the mechanism *only* characterizes loads.

6 EVALUATION METHODOLOGY

In this Chapter we detail the simulation environment and the mechanisms implemented for the 7 Chapter. The related work detailed here has been implemented in all details described in their respective papers. The major difference is the architecture configuration used, as we simulate a realistic scenario, the Sandy Bridge architecture, as a baseline.

6.1 Simulation Environment and Metrics

To validate our mechanism, we used a cycle-accurate in-house simulator (ALVES, 2014). It accurately simulates the micro-architecture, modeling all functional unit contention, register dependency, processor and system restrictions, cache architecture, DRAM memory and interconnections. In Table 6.1, we specify the details of the simulated system, which has a configuration based on the Intel Sandy Bridge micro-architecture.

We used two parallel workload sets for evaluation: (i) seven applications from the NAS-NPB workload suite, with the *A* input size, and (ii) seven applications from the the SPEC-OMP2001 workload suite with *ref* input size. On average each thread's execution trace contains 150 million instructions. Overall, the trace of each application represents one parallel time step from each algorithm. The applications use the *OpenMP* parallelization library and were compiled with *gcc* 4.6.3, with the *-O3* options. We only used the parallel benchmarks for these tests as single-threaded benchmarks do not generate enough memory pressure to provide performance improvement opportunities.

As we use parallel benchmarks for all our performance tests, we measure performance in terms of total number of execution cycles until the program finishes. This means that individual thread behavior may vary during execution. We show speedups calculated to express the proportion of total execution time reduced in each application in comparison to the baseline.

6.2 Evaluated Memory Controller Policies

Given the correlation coefficients presented in Section 2, we have chosen to improve the memory controller because of the high correlation that memory accesses have with performance. Considering the proposals of Ghose et al. (2013) and Lee et al. (2008), 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 to the application's critical path. The baseline for the memory controller policies is the FR-FCFS (RIXNER et al., 2000) 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 Criticality Binary Prediction (CBP) from Ghose et al. (2013) and the PADC from Lee et al. (2008).

The *CBP* mechanism gives priority to the load instructions that stall the commit stage.

Table 6.1: Baseline simulated architectural parameters.

Component	**Configuration**
Processor Cores	8 cores OoO @ 2.66 GHz, 32 nm; in-order front-end and commit; 16 stages (3-fetch, 3-decode, 3-rename, 2-dispatch, 3-commit stages); 16 B fetch block size (up to 6 instructions); Decode and commit up to 5 instructions; Rename/dispatch/execute up to 5 μ instructions; 18-entry fetch buffer, 28-entry decode buffer; 3-alu, 1-multiplication and 1-division integer units (1-3-32 cycle); 1-alu, 1-multiplication and 1-division floating-point units (3-5-10 cycle); 1-load and 1-store functional units (1-1 cycle); MOB entries: 64-read, 36-write; 168-entry ROB;
Branch Predictor	1 branch per fetch; 8 parallel in-flight branches; 4 K-entry 4-way set-assoc., LRU policy BTB; Two-Level PAs 2-bit; 16 K-entry BHT;
L1D Cache	32 KB, 8-way, 64 B line size; LRU policy; 2-cycle; MSHR: 8-request, 10-write-back, 1-prefetch; Stride prefetch: 1-degree, 16-strides table;
L1I Cache	32 KB, 8-way, 64 B line size; LRU policy; 2-cycle; MSHR: 8-request, 1-prefetch; Stride prefetch: 1-degree, 16-strides table;
L2 Cache	Private 256 KB, 8-way, 64 B line size; LRU policy; MSHR: 4-request, 6-write-back, 4-prefetch; 4-cycle; Stream prefetch: 4-degree, 64-dist., 64-streams;
L3 Cache	Shared 16 MB (8-banks), 2 MB per bank; MOESI coherence protocol; 16-way, 64 B line size; LRU policy; 6-cycle; Inclusive; MSHR: 8-request, 12-write-back; Bi-directional ring interconnection;
DRAM and Bus	On-chip DRAM ctrl., 8 banks/channel; 4-channels; DDR3 1333 MHz; 8 burst length; 4 KB row buffer per bank, Open-row first; 4.0 core-to-bus frequency ratio; 9-CAS, 9-RP, 9-RCD & 28-RAS cycles; MSHR: 128-request, 64-write-back, 32-prefetch;

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: *BlockCount*, *LastStall*, *MaxStall* and *TotalStall*. *BlockCount* counts how many times a load blocked the commit stage, regardless of how many cycles it blocked the commit stage. *LastStall* counts how many cycles the commit stage was stalled by the last load. *MaxStall* counts the maximum amount of cycles the commit stage was stalled by a single load execution. *TotalStall* counts the total amount of cycles the commit stage was stalled by a load. This information is then added in the memory packages to define priority levels, instead of the simpler 1-bit binary policy. The paper reports speedups of 6.5% for the basic binary policy, 8.7% for *BlockCount*, no tangible benefit for *LastStall* when compared to binary policy, 9.3% for *MaxStall* and meager improvements over *MaxStall* with *TotalStall*. We implemented only the binary policy, as the other policies demand too many bits of added bandwidth and storage to the memory controller.

For the *PADC* mechanism, each cache line is extended by adding 2 bits, a prefetch bit and an access bit, in the same way of (SRINATH et al., 2007). These bits track which prefetches were useful. By measuring prefetch accuracy every 100000 cycles, PADC decides whether it should give the same priority to prefetches and demands, or whether it should prioritize demand requests and drop prefetches given the pollution present. It uses 4 prefetch accuracy threshold values and 4 corresponding thresholds for the number of cycles which the prefetch waited for service to drop it, 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 longer than 50000 cycles to be serviced. Between 10% and 30% accuracy, the mechanism drops prefetches which waited longer than 300 cycles to be serviced. If accuracy is lower than 10%, it drops any prefetch which waits 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. Therefore, BLAP-CBP uses the following priority set of rules:

1. Give priority to critical row hits;
2. Give priority to non-critical row hits;
3. Give priority to critical non-row hits;
4. Give priority to non-critical, non-row hits.

In *BLAP-PADC-8L*, 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 which combines *CBP* and *PADC*. 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 level rules are:

1. Give priority to demand critical row hits;
2. Give priority to prefetch critical row hits;

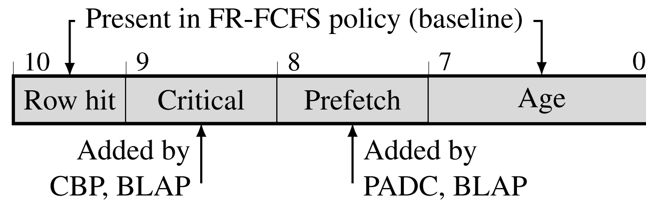

Figure 6.1: Request selection logic for different memory controller mechanisms.

3. Give priority to demand non-critical row hits;
4. Give priority to prefetch non-critical row hits;
5. Give priority to demand critical non-row hits;
6. Give priority to prefetch critical non-row hits;
7. Give priority to demand non-critical, non-row hits;
8. Give priority to prefetch non-critical, non-row hits.

Figure 6.1 illustrates each input bit used by the selection logic for different memory controller mechanisms. The mechanisms compare the information bits from the request as a single number, by concatenating all bits and considering the leftmost 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, in order to give priority to demand requests. The critical bit is the information fed either by CBP or BLAP. Finally, row hit is 1 if the address of the request matches the currently open row.

In comparison to CBP, the first advantage of BLAP-CBP is that we can exploit other processor bottlenecks beyond memory pressure. The second advantage of this characterization that yields performance gains is that it also provides information on branch mispredictions. We will not give priority to loads that are followed by a mispredicted branch. Doing so would not help the block performance, which is why branch prediction is given the highest value. The objective in doing so is to reduce the number of critical loads to the loads which can actually improve performance if prioritized, which is not the case of loads followed by mispredicted branches. 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 the CBP's table. Both implementations require the same amount of hardware in the memory controller and MSHR structures to pass the information bit that indicates critical requests.

Compared to the PADC, BLAP-PADC-8L required four times less storage by using the BTB to store the profile information. We cannot offer a straightforward adaptation, as BLAP does not have prefetch accuracy information. Rather, we used the average demand request wait time to make a new prefetch dropping mechanism, and used BLAP information to avoid evicting prefetches triggered by relevant blocks. In this sense, the memory controller used for BLAP-PADC-8L is entirely new, as it mixes the priority from CBP, extends it by applying it to prefetches, and uses the concept of prefetch dropping to discard irrelevant prefetches, thus alleviating the pressure on the memory request buffer and average memory service time. In the next Chapter, we evaluate all of these mechanisms using the configuration detailed in this Chapter.

7 EXPERIMENTAL RESULTS

In this chapter we discuss the two main mechanism implementations used to achieve performance improvements using the characteristics detected per block. We chose these mechanisms because they are sensitive to the information BLAP collects, have shown good prospects for performance improvement, and were recent works representing novelty in the computer architecture field. Although we detect several characteristics, these mechanisms only use the "Mem" characteristic.

After we reach a satisfactory mechanism, we perform a design space exploration to evaluate how the mechanism behaves with different architectural parameters. In all our evaluations, we only used parallel benchmarks, as the memory pressures present in single-threaded sequential benchmarks were too small to provide any improvement opportunity.

7.1 Mechanism Exploration

Figure 7.1 presents results for different mechanisms running NAS-NPB and SPEC-OMP2001 benchmarks. In the Figure, we show speedup relative to the baseline for all benchmarks, meaning higher values mean better performance. The first observation is that the average gains of both related work are different from the ones found in their work, due to different benchmarks, architectural parameters and simulators. Although the effect of the mechanism implementation is noticeable, as seen in the IS benchmark, the average benchmark speedup is low.

For this experiment, PADC offers the highest improvement, achieving 19.02% for IS. We have average performance improvements of 1.89% with CBP, 0.80% with BLAP-

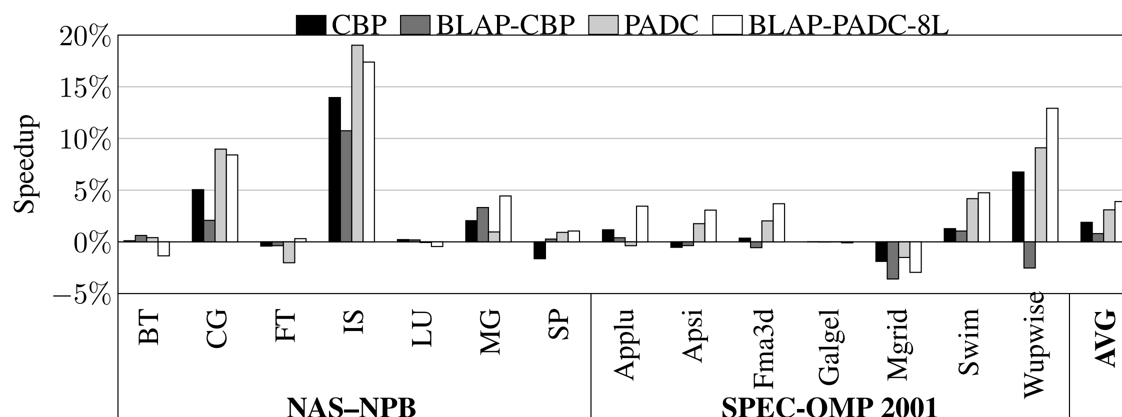

Figure 7.1: Performance results for NAS-NPB and SPEC-OMP2001, relative to FR-FCFS baseline.

CBP, 3.10% with PADC and 3.90% with 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 using the average demand request time. Although the average time could also be used for PADC, there is no information on the original paper on which thresholds would better benefit from such value, and we would rather replicate the original work in every aspect for comparison. Using BLAP information, the mechanism is able to drop prefetches more aggressively while still servicing important prefetches. This is because the prioritized prefetches come mostly from critical, repetitive blocks, which got characterized as "Mem". This makes BLAP-PADC-8L inclinable to drop false-positive triggered prefetches, as they do not receive priority (low number of executions of their block may not allow characterization) and are thus left to be dropped (since the priority of non-critical prefetches is below average demand request time for closed rows).

In order to stress the memory controller mechanisms and their profiling methods, we used a stream prefetcher with increased agressivity. A stream prefetcher normally looks for cache misses within a range (search distance), and, if the sequential misses fall within this distance, we allocate a stream. If any cache access falls within this stream initial address and *m* cache lines ahead(where *m* is the *prefetch distance* parameter), we prefetch *n* cache lines (where *n* is the *prefetch degree* parameter) starting from address *prefetch triggering address + (prefetch distance * cache line size)*, then attributing starting address with the value of the request that triggered the prefetches. In our baseline, we used prefetch degree 4 and prefetch distance 64, which is already considered aggressive. It is considered aggressive because of the high number of packages prefetched and the large area of cache lines detected. This configuration was used since the current related work all use this same aggressiveness. During validation, Sandy Bridge seems to use prefetch degree 2 and prefetch distance 32 for its L2 cache stream prefetcher, which seems realistic for current architectures.

In the next experiment, Figure 7.2 shows results for CBP, PADC, BLAP-CBP and BLAP-PADC-8L with prefetch degree 8 and prefetch distance 128. In this experiment BLAP-PADC-8L offers the highest improvement, achieving 37.05% for Wupwise. We have average improvements of 3.99% with CBP, 1.72% with BLAP-CBP, 4.24% with PADC and 13.14% with BLAP-PADC-8L.

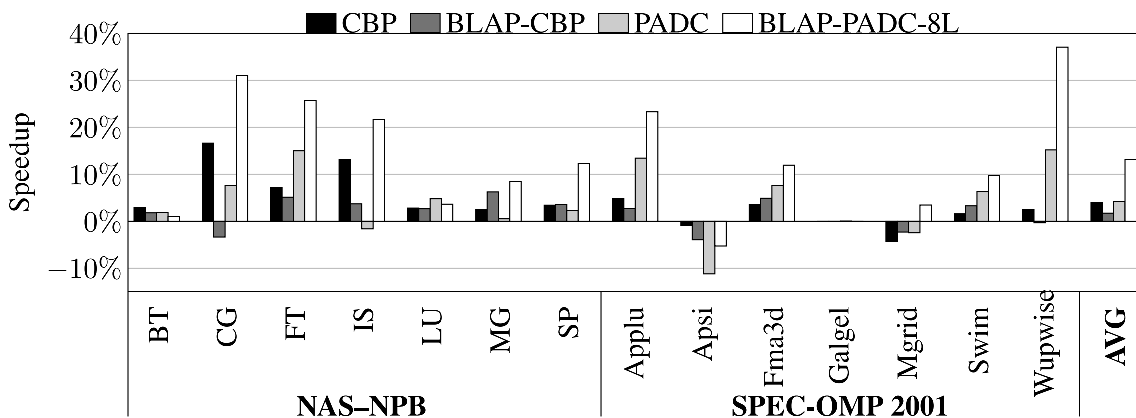

Figure 7.2: Performance results for NAS-NPB and SPEC-OMP2001 with increased aggressivity prefetcher, relative to FR-FCFS baseline.

CBP performed 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 in the previous test, BLAP-CBP also improves, but it does not reach the same improvements 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. We have no correct way of determining these parameters, as they are not discussed in the original paper, which leads to unfairness in this comparison. 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 aggressive prefetch dropping.

In the opposite direction of the previous Figure, Figure 7.3 shows results with a conservative stream prefetcher aggressiveness for CBP, PADC, BLAP-CBP and BLAP-PADC-8L, with prefetch degree 2 and prefetch distance 32. By conservative, we mean values that are deemed safe and do not generate performance degradation due to cache pollution. These values are used for current architectures. This experiment shows that all mechanisms offer little improvement, as there is not enough memory pressure. The highest improvement of 3.88% with BLAP in the benchmark MG does not suffice to make the results considerable, as all averages are within 1% of the baseline performance.

We can observe that the prefetch dropping mechanisms hurt performance, as they are likely too aggressive when dropping prefetches. In such a conservative prefetcher aggressiveness configuration, prefetches are unlikely to generate significant pollution in the memory controller MSHRs and caches.

Figure 7.4 shows speedup 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 within all benchmarks. Moreover, it is able to differentiate and store blocks targeted by branches and fall-through blocks by storing all addresses that come after a branch.

Comparing the BTB to the large cache implementation, we can notice similar performance improvements over the baseline. This shows that the large entry number in the BTB is sufficient to keep the profile information for most benchmarks. This has been further tested by using different BTB sizes, and while a smaller BTB does not show the

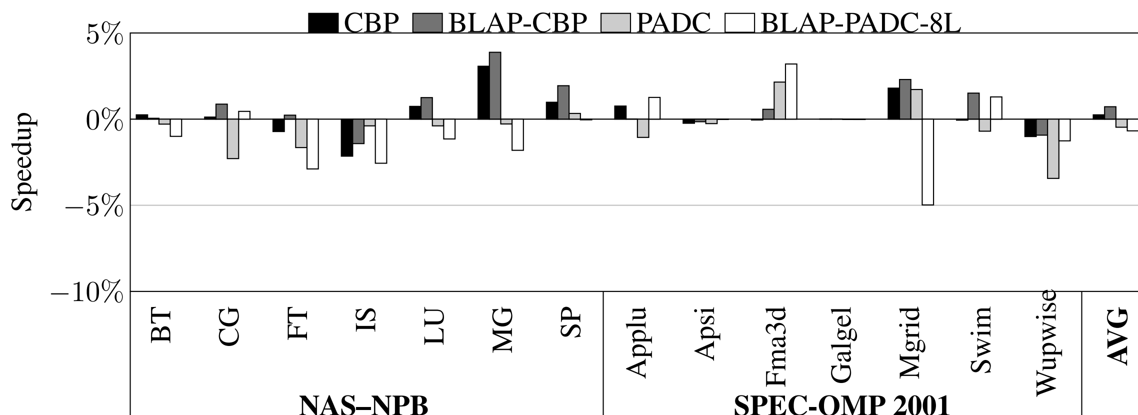

Figure 7.3: Performance results for NAS-NPB and SPEC-OMP2001 with conservative aggressivity prefetcher, relative to FR-FCFS baseline.

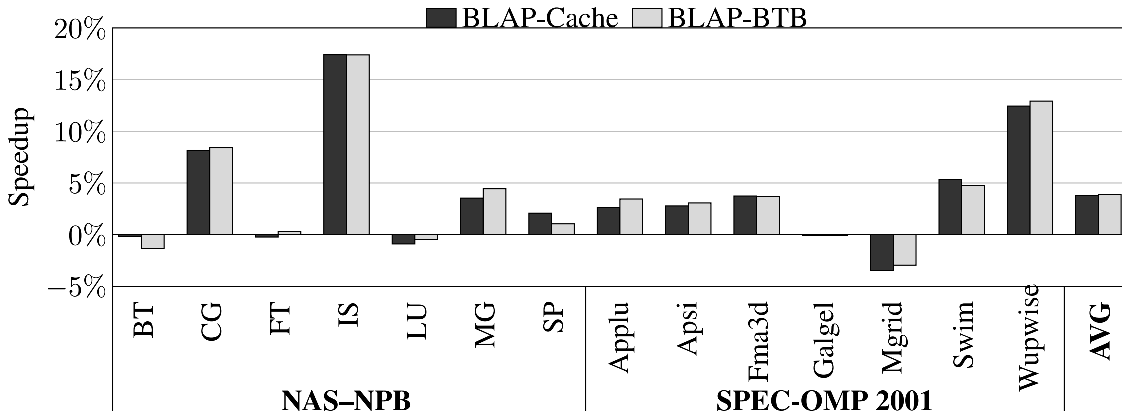

Figure 7.4: Performance results comparison between using the branch target buffer and a large cache, relative to FR-FCFS baseline.

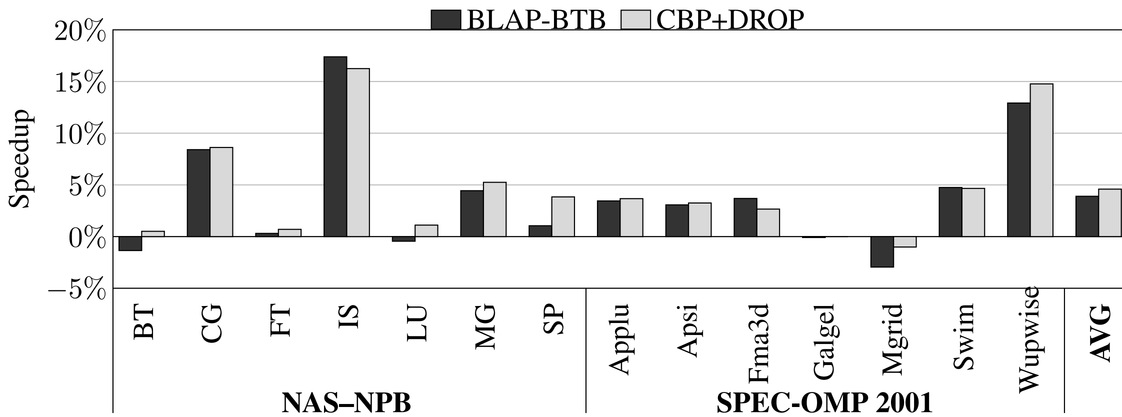

Figure 7.5: Performance results comparison using the same memory controller, but different informations, from BLAP and CBP respectively, relative to FR-FCFS baseline.

same performance improvements, a larger BTB was not able to show any improvement at all, registering the same hit ratio as the size currently in use. This can be attributed to the use of traces that are not composed of the entire program, but rather one application step. However, one application step should contain references to most of the branches that have repeated accesses. There are cases where the BTB implementation outperforms the large cache, as giving priority only to repeatedly executed blocks (i.e. blocks targeted by a branch in a loop) instead of priority to all blocks helps to differentiate the truly important blocks in an application.

Figure 7.5 shows speedup results comparing the performance of the memory controller and prefetch dropping policies when fed with information from BLAP and CBP. Since CBP is able to detect single-instruction granularity and change the priority of instructions as soon as they stall the first time in the processor, requiring no training time, its characterization of loads is slightly better than BLAP's. However, CBP is unable to detect other characteristics, which gives BLAP an edge for future work and extensions.

7.2 Design Space Exploration

In order to explore the scalability of the mechanism for future systems design and its general behavior, we have performed three tests to explore the characteristics of the

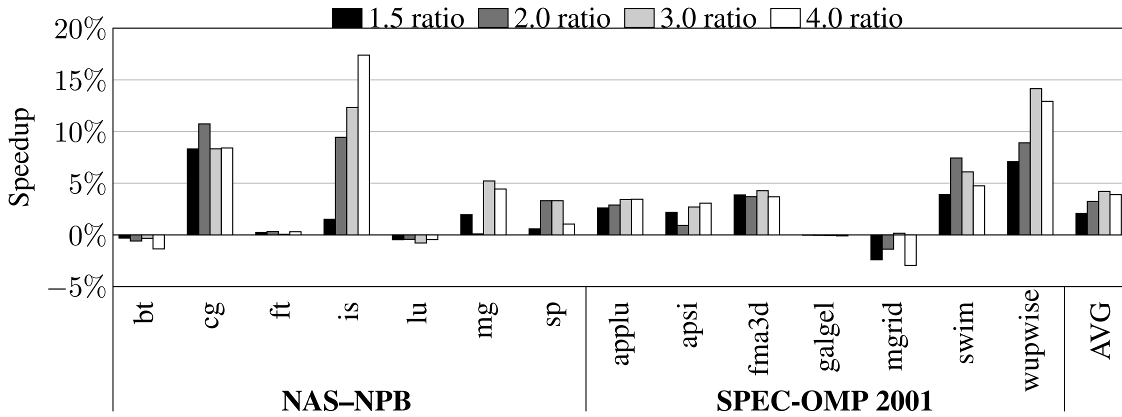

Figure 7.6: Mechanism performance, normalized to baseline configurations with different memory latencies

mechanism under different conditions.

7.2.1 Memory Latency

One of the most important aspects of the mechanism provided is related to the memory latency present in the system. In Figure 7.6 we have varied the core-to-bus ratio, i.e. the ratio between the core cycle period and the memory bus cycle period, from lower levels to the configuration used for our tests.

We can clearly observe that, with lower memory latencies, BLAP-PADC-8L offers diminishing advantages, as reordering requests and prefetch pollution have less impact in faster memories. However, future memories are likely to have a larger latency when compared to processors. The DRAM we currently use in Sandy Bridge has a core-to-bus period ratio of 3.0, which is not far from the ratio used for the results. The results actually show that a ratio of 3.0 yields better speedup results, although we never tuned the parameters of the initial configuration to obtain the best results. With higher frequency processors, running at 3.2GHz, this ratio would actually be 5.0, which shows that the performance is still scaling. However, memory pressure is much more likely to become an issue due to the number of cores generating requests with the scaling of core numbers than the actual core-to-bus ratio.

7.2.2 Cores Number

In order to test how the mechanism scales when using a larger number of cores we have scaled our system to 16 cores. We still use a bidirectional ring, but now we have 16 last level cache banks of 2MB each. In Figure 7.7 we can see that we obtain an average of 6.04% speedup in relation to the baseline configuration with 16 cores, which shows that our mechanism's performance scales with the larger number of requests per memory controller. This happens because the priorization of critical requests and alleviating prefetcher pollution becomes increasingly important with more pressure on the memory and more requests to consider when ordering request priorities.

7.2.3 Cache Size

Evaluation of memory-related mechanisms are always impacted by the cache hierarchy present in the system. To evaluate this, in Figure 7.8 we vary the cache size of our baseline system.

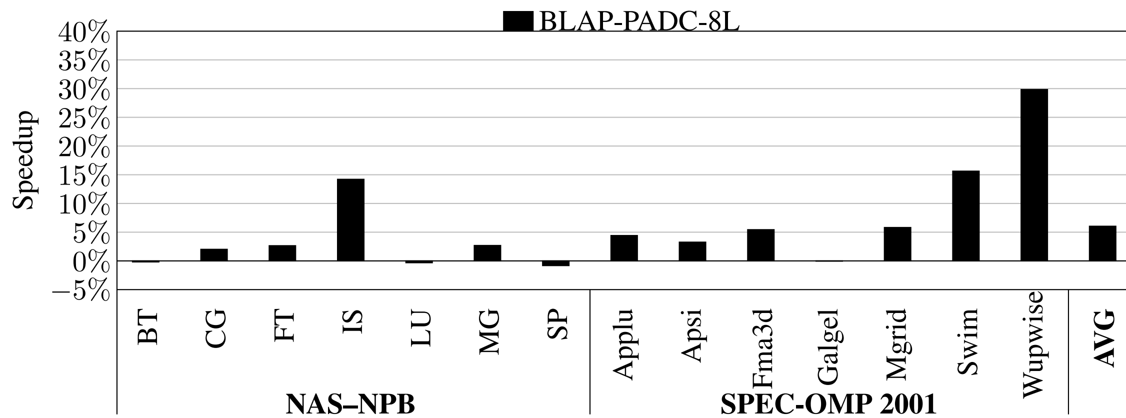

Figure 7.7: Mechanism performance with BLAP-PADC-8L using BTB, normalized to baseline configurations with 16 cores

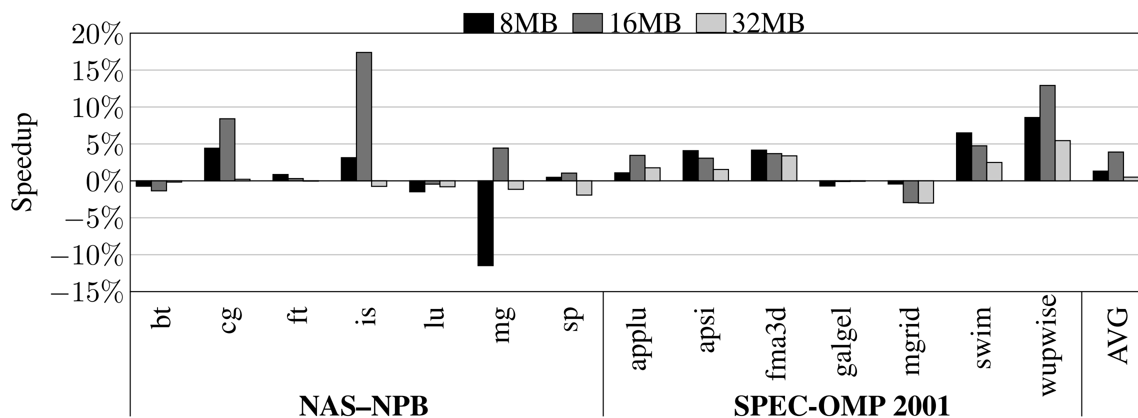

Figure 7.8: Mechanism performance, normalized to baseline configurations with different memory latencies

In the Figure we can observe a larger cache will filter more accesses, reducing memory pressure and consequentially reducing the performance improvement opportunities for our mechanism. On the other hand, a smaller cache has the opposite effect.

8 CONCLUSIONS AND FUTURE WORK

In this master thesis, we have presented BLAP, a mechanism capable of characterizing blocks to better inform other mechanisms about basic block behavior. Our main objective was to provide a mechanism that could aggregate behavior to minimize storage overhead and have a simple, efficient detection, in order to compare basic block information versus instruction information. BLAP has shown several advantages. 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 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.

8.1 Contributions

Characterization Mechanism: We proposed BLAP, an efficient detection mechanism capable of characterizing applications at the basic block level during their execution. We have detailed its implementation avoiding any critical path changes within reasonable hardware overhead.

Low Overhead Profile Storage: By using the BTB, our mechanism requires negligible storage to keep information about the relevant characteristics of each basic block.

New Memory Controller: Using a combination of related works, we were able to design a new memory controller that can outperform both related work using BLAP's characterization. It works by dropping prefetches deemed useless or late, by servicing demand requests and useful prefetches first. Additionally, we perform design space exploration to show that this memory controller scales well with future memory pressures.

Our results show that basic block granularity can be just as relevant as single instruction granularity for memory accesses. The findings indicate that as basic blocks naturally track a program's phase progression, we are able to more accurately adapt to different memory pressures that occur in different program phases. We were able to improve performance by 3.9% on average (up to 17.39%), 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, and scales well for a larger number of cores. This master thesis generated two publications: "Influência das Características de Processadores e Aplicações no Nível de Blocos Básicos" in WSCAD (Workshop de Sistemas Computacionais de Alto Desempenho) in 2013, and "Profiling and Reducing Micro-Architecture Bottlenecks at the

Hardware Level" in SBAC-PAD (Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho) 2014.

8.2 Future Work

For future work, there are several ways to build on this work. First, exploring BLAP's information for other instruction types among those selected should be the first concern. For branches, it might be possible to increment the base detection to work on data-dependent branches, adjust Farooq et al. (2013) to work using our mechanism information. For floating point units, it should be possible to work on instruction reordering when these are a block's problem, as compiler usually gives priority to memory instructions when ordering them. Additionally, value reuse and energy consumption research are viable ways to use the base concept of the mechanism.

Second, the concept should be tested with different architectures, such as ARM and MIPS, and configurations, varying core number and core complexity. The scheme shown in this work should not be implemented in simple cores, but large cores used in heterogeneous systems should have improved performance with the memory controller options shown here.

Third, there are still optimizations that can be done within the mechanism itself. Stabilizing behavior can likely be improved, or perhaps skipped altogether. Devising a minimum number of stall cycles comparison to ensure that characterized blocks have a relevant stall is also an interesting alternative. Finally, the cases in which the mechanism degrades performance should also be studied, to enable controlling and shutting down the mechanism action for such blocks.

REFERENCES

AFRAM, F.; ZENG, H.; GHOSE, K. A group-commit mechanism for ROB-based processors implementing the X86 ISA. In: HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA2013), 2013 IEEE 19TH INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2013. p.47–58.

ALVES, M. **Increasing Energy Efficiency of Processor Caches via Line Usage Predictors**. 2014. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul.

ANSALONI, D. et al. Enabling modularity and re-use in dynamic program analysis tools for the Java virtual machine. In: **ECOOP 2013–Object-Oriented Programming**. [S.l.]: Springer, 2013. p.352–377.

CLARK, N. et al. Liquid SIMD: abstracting simd hardware using lightweight dynamic mapping. In: HIGH PERFORMANCE COMPUTER ARCHITECTURE, 2007. HPCA 2007. IEEE 13TH INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2007. p.216–227.

COCKE, J. Global common subexpression elimination. **SIGPLAN Not.**, New York, NY, USA, v.5, n.7, p.20–24, July 1970.

CRISTAL, A. et al. Out-of-order commit processors. In: SOFTWARE, IEE PROCEEDINGS-. **Anais...** [S.l.: s.n.], 2004. p.48–59.

FAROOQ, M. U.; KHUBAIB, K.; JOHN, L. K. Store-Load-Branch (SLB) predictor: a compiler assisted branch prediction for data dependent branches. In: HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA2013), 2013 IEEE 19TH INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2013. p.59–70.

GHOSE, S.; LEE, H.; MARTÍNEZ, J. F. Improving Memory Scheduling via Processor-side Load Criticality Information. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 40., New York, NY, USA. **Proceedings...** ACM, 2013. p.84–95. (ISCA '13).

HAMERLY, G. et al. Simpoint 3.0: faster and more flexible program phase analysis. **Journal of Instruction Level Parallelism**, [S.l.], v.7, n.4, p.1–28, 2005.

HUANG, J.; LILJA, D. Extending value reuse to basic blocks with compiler support. **Computers, IEEE Transactions on**, [S.l.], v.49, n.4, p.331–347, 2000.

HUR, I.; LIN, C. Memory prefetching using adaptive stream detection. In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 39. **Proceedings...** [S.l.: s.n.], 2006. p.397–408.

JALEEL, A. et al. High performance cache replacement using re-reference interval prediction (RRIP). In: ACM SIGARCH COMPUTER ARCHITECTURE NEWS. **Anais...** [S.l.: s.n.], 2010. v.38, n.3, p.60–71.

KAMBADUR, M.; TANG, K.; KIM, M. A. Harmony: collection and analysis of parallel block vectors. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 39., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2012. p.452–463. (ISCA '12).

LATTNER, C.; ADVE, V. LLVM: a compilation framework for lifelong program analysis & transformation. In: CODE GENERATION AND OPTIMIZATION, 2004. CGO 2004. INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2004. p.75–86.

LEE, C. J. et al. Prefetch-aware DRAM controllers. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 41. **Proceedings...** [S.l.: s.n.], 2008. p.200–209.

LUK, C.-K. et al. Pin: building customized program analysis tools with dynamic instrumentation. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2005., New York, NY, USA. **Proceedings...** ACM, 2005. p.190–200. (PLDI '05).

PADMANABHA, S. et al. Trace based phase prediction for tightly-coupled heterogeneous cores. In: ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 46. **Proceedings...** [S.l.: s.n.], 2013. p.445–456.

PANAIT, V.-M.; SASTURKAR, A.; WONG, W.-F. Static identification of delinquent loads. In: CODE GENERATION AND OPTIMIZATION, 2004. CGO 2004. INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2004. p.303–314.

PATEL, S. J.; EVERS, M.; PATT, Y. N. Improving trace cache effectiveness with branch promotion and trace packing. **ACM SIGARCH Computer Architecture News**, [S.l.], v.26, n.3, p.262–271, 1998.

PATEL, S. J.; LUMETTA, S. S. rePLay: a hardware framework for dynamic optimization. **Computers, IEEE Transactions on**, [S.l.], v.50, n.6, p.590–608, 2001.

PATIL, H. et al. Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. In: MICROARCHITECTURE, 2004. MICRO-37 2004. 37TH INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2004. p.81–92.

PETER GREENHALGH, A. **Big. LITTLE Processing with ARM Cortex™-A15 & Cortex-A7**. [S.l.]: Sep, 2011.

PILLA, M. L. et al. Value predictors for reuse through speculation on traces. In: COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 2004. SBAC-PAD 2004. 16TH SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2004. p.48–55.

RATANAWORABHAN, P.; BURTSCHER, M. Program phase detection based on critical basic block transitions. In: PERFORMANCE ANALYSIS OF SYSTEMS AND SOFTWARE, 2008. ISPASS 2008. IEEE INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2008. p.11–21.

REINDERS, J. **VTune performance analyzer essentials**. [S.l.]: Intel Press, 2005.

RIXNER, S. et al. **Memory access scheduling**. [S.l.]: ACM, 2000. v.28, n.2.

ROTENBERG, E.; BENNETT, S.; SMITH, J. E. Trace cache: a low latency approach to high bandwidth instruction fetching. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 29. **Proceedings...** [S.l.: s.n.], 1996. p.24–35.

SHERWOOD, T.; PERELMAN, E.; CALDER, B. Basic block distribution analysis to find periodic behavior and simulation points in applications. In: PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, 2001. PROCEEDINGS. 2001 INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2001. p.3–14.

SRINATH, S. et al. Feedback directed prefetching: improving the performance and bandwidth-efficiency of hardware prefetchers. In: HIGH PERFORMANCE COMPUTER ARCHITECTURE, 2007. HPCA 2007. IEEE 13TH INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2007. p.63–74.

WALL, D. W. **WRL Research Report 93/6**. [S.l.]: DEC Western Research Laborator, 1993.

YUFFE, M. et al. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In: SOLID-STATE CIRCUITS CONFERENCE DIGEST OF TECHNICAL PAPERS (ISSCC). **Anais...** [S.l.: s.n.], 2011.

ZHUANG, X.; LEE, H.-H. Reducing cache pollution via dynamic data prefetch filtering. **Computers, IEEE Transactions on**, [S.l.], v.56, n.1, p.18–31, 2007.

9 APPENDIX - PORTUGUESE SUMMARY

In this chapter, we present a summary of this master thesis in the portuguese language, as required by the PPGC Graduate Program in Computing. Neste capítulo, é apresentado um resumo desta dissertação de mestrado na língua portuguesa, como requerido pelo Programa de Pós-Graduação em Computação.

9.1 Introdução

A caracterização de blocos básicos é uma técnica importante e recorrente, usada para vários tipos de otimizações automáticas. Ferramentas em software, tal como o Vtune (REINDERS, 2005), permitem análises manuais para detectar oportunidades de ganhos de desempenho, tal como reescrita de código para evitar uma alta taxa de falta de dados nas caches ou uma alta taxa de predições de desvio erradas. A granularidade de blocos básicos é especialmente útil (COCKE, 1970) pois blocos básicos representam porções de código que sempre acabam em desvios. Portanto, o comportamento e as fases de um programa são definidos pela seqüência de blocos básicos executados.

No entanto, não existe nenhum uso claro de blocos básicos em computadores de propósito geral atuais (YUFFE et al., 2011). O único uso realizado poderia ser considerado o da cache de traços (ROTENBERG; BENNETT; SMITH, 1996), e nenhuma caracterização é feita em cima do que é armazenado. Os projetos de computadores de propósitos gerais em geral apenas coletam informações no nível de instruções. Embora vários artigos científicos usem análise no nível de bloco, a maioria o faz através do uso de software, mesmo para adaptações de hardware (PANAIT; SASTURKAR; WONG, 2004; RATANAWORABHAN; BURTSCHER, 2008). Uma das raras técnicas que faz uso de blocos básicos no nível de hardware foi o projeto rePlay (PATEL; LUMETTA, 2001). Neste trabalho, o código é analisado para realização de otimizações durante execução e armazenamento em uma cache de traços para execuções futuras, embora nenhum perfilamento de gargalos de execução seja feito.

Nesta dissertação de mestrado, propomos um criador de perfis de blocos básicos em hardware, em inglês *Block-level Architectural Profiler*(BLAP). Tal mecanismo caracteriza blocos básicos de acordo com os atrasos mais relevantes que ocorreram por bloco, permitindo melhoria de desempenho do bloco em execuções futuras. O BLAP tem várias vantagens em relação a outros mecanismos. Devido ao uso de blocos básicos, ele se adapta automaticamente à mudanças de fase do programa, por estar fortemente ligado aos blocos de instruções sendo executados. Ele também precisa de menor espaço de armazenamento que mecanismos que usam granularidade de instruções, pois ele agrega as informações dos atrasos apenas com um valor relevante. É possível usar o Branch Target Buffer (BTB) para armazenar eficientemente essa informação adicional, já que o BTB

possui os endereços iniciais de cada bloco. O BLAP também é capaz de detectar diferentes tipos de problemas que influenciam no desempenho de um bloco, portanto podendo providenciar informações para vários mecanismos diferentes.

Para demonstrar o potencial do BLAP, é explorado o uso de sua informação para projetar um novo controlador de memória, capaz de usar esta informação para diferenciar leituras de diferentes prioridades e descartar *prefetches*. Comparado com a informação de granularidade de instrução usada por Ghose et al. (2013) e Lee et al. (2008), o perfil de nosso mecanismo pode providenciar melhor desempenho com mínima área adicional em hardware. Adicionalmente, a implementação básica do BLAP pode ser estendida para fornecer informações e mais detalhes para outros mecanismos, mantendo praticamente o mesmo custo em hardware. Ao nosso conhecimento, nenhuma pesquisa prévia criou perfis de blocos básicos em hardware. As principais contribuições desta tese são as seguintes:

Mecanismo de Caracterização: É proposto o BLAP, um mecanismo de detecção eficiente capaz de caracterizar aplicações no nível de blocos básicos durante a sua execução.

Perfil Eficiente: o BLAP requer espaço de armazenamento negligível para manter informações sobre as características relevantes de cada bloco básico. Tal mecanismo pode ser implementado estendendo o BTB com alguns bits por entrada.

Melhoria de Desempenho: o BLAP foi integrado com mecanismos que melhoram o desempenho da memória ao adaptá-los para usarem a informação do perfil ou ao criar um novo mecanismo que reproduzisse o conceito usando o BLAP.

O objetivo final deste trabalho é propor e estudar um mecanismo em hardware capaz de detectar os blocos que compõe um programa e caracterizá-los com as instruções responsáveis pelas maiores latências no bloco. Tal caracterização deve tornar possível a melhoria de desempenho através do uso da informação por parte de outros mecanismos, tal como prefetchers ou políticas de prioridade.

9.2 Detecção de Blocos Básicos

Um bloco básico é uma porção de código com um único ponto de entrada e um único ponto de saída. Portanto, todo bloco acaba com uma instrução de desvio ou com a instrução anterior ao alvo de uma instrução de desvio. Isto permite com que mecanismos baseados em blocos acompanhem a fase de um programa automaticamente, já que uma fase é caracterizada pelos blocos básicos usados (RATANAWORABHAN; BURTSCHER, 2008). No entanto, uma implementação eficiente em hardware deve ser relaxada permitindo múltiplos pontos de entrada, já que é impossível detectar que instruções são alvos de desvios sem grandes mudanças na funcionalidade do conjunto de instruções da arquitetura.

Um fenômeno que deve ser considerado ao estender o BTB é que tal estrutura apenas armazena informação para blocos que começam após um desvio tomado. Dado que o comportamento a ser explorado é normalmente repetitivo, isto normalmente não é um problema, já que o formato do código de *loops* fará com que blocos básicos comecem após desvios tomados. Já que não podemos reconhecer quais instruções são alvos de desvios, estamos quebrando a definição de bloco básico, pois iremos gerar sobreposição de código analisado entre blocos. Tais blocos irão agregar comportamento de todas instruções dos poucos blocos básicos reais e menores contidos dentro deles, e portanto não serão caracterizados separadamente. Porém, tais blocos básicos reais serão caracterizados corretamente assim que um desvio tomado mudar o fluxo de execução para eles, obtendo

assim seu endereço de começo real. Como blocos menores representam condições dentro de *loops* na maior parte dos casos, eles serão executados vezes o suficiente para serem caracterizados. Se não forem, então eles provavelmente não são relevantes.

Para detecção de características, podemos utilizar vários métodos. Atualmente, processadores da Intel vem com contadores de hardware disponíveis para perfilamento e análise de desempenho de código, usados por ferramentas tais como Vtune (REINDERS, 2005). Tais contadores mantém informações sobre vários eventos durante a execução do processador, tais como faltas de dados nas caches, número de operações por tipo, ao ponto de detalhar transações nos barramentos. Tais estatísticas intuitivamente correlacionam-se com o desempenho do sistema, e podem ser usadas para providenciar idéias em relação a motivos para desempenho sub-ótimo de um código. Porém, estatísticas não podem ser diretamente comparadas à latências, pois "um acesso à cache L2" pode significar 7 ou 15 ciclos, dependendo do estado de requisições pendentes nas caches. Adicionalmente, estatísticas não ocorrem em ciclos específicos, gerando dificuldade em relação à estabelecer quais estatísticas pertencem a quais blocos em um processador fora de ordem. É possível que um acesso a memória seja feito em curto tempo e assim se encaixe em um bloco vários blocos à frente do seu bloco original.

Outro método de gerar características é observar o número de ciclos que instruções atrasam outras devido a uma dependência. Este é um método direto de observar quais instruções são mais relevantes em cada bloco básico, mas não necessariamente confiável devido à irrelevância de instruções que não estão no caminho crítico. Adicionalmente, a complexidade de hardware é enorme para obter latências para todos registradores.

Em geral, obter informações detalhadas de execução eficientemente é um problema complexo. Como almejamos eficiência, temos três necessidades em relação à detecção de desempenho de instruções. Primeiro, uma estatística deve demonstrar relevância, ou seja, um impacto direto no desempenho do código. Embora faltas de dados nas caches sejam um bom indicador de problemas com a memória, arquiteturas modernas são normalmente tolerantes a falhas nos níveis mais altos devido à alta capacidade de ILP, o qual provê computação o suficiente para mascarar tal latência. Isto é, para a maioria dos casos, faltas de dado na cache de dados de nível mais alto não atrasam o processador. E latências de instruções específicas nem sempre estão no caminho crítico de execução do processador, tornando-as muitas vezes inúteis.

Segundo, eventos diferentes em hardware necessitam comparação direta. Quando uma falta de dados ocorre em uma cache, sabemos o nível que responderá, mas não podemos medir com precisão a latência devido à variações de carga nas estruturas de MSHR das caches. E mesmo uma falta de dados no último nível de cache poderia ter sua latência escondida por um desvio previsto erroneamente. Se for desejado achar qual a latência mais relevante de um bloco ou comparar ela a um valor fixo, não é possível fazê-lo.

Terceiro, são necessárias características com local determinado e em ordem para obtenção de dados. Caso as estatísticas não possam ser alocadas a um determinado trecho de código com simplicidade, torna-se muito custoso gerar mecanismos que reordenem tais dados para encaixá-los nos blocos básicos apropriados. Se isto não for feito, ocorre a descaracterização de blocos e poluição de dados, gerando estatísticas que não tem credibilidade para serem usadas.

Para sobrepor tais desafios, optou-se por explorar o estágio de graduação do processador. Instruções apenas causam gargalos visíveis ou atrasam o processador caso não possam ser graduadas. Como o estágio de graduação é feito em ordem, isto acarreta em parar todo o *pipeline* do processador. Assim, resolve-se o primeiro problema. Como

observa-se o número de ciclos que cada instrução parou o estágio de graduação, temos informação que pode ser comparada diretamente entre as instruções de diferentes tipos, resolvendo o segundo problema. Finalmente, devido às instruções serem graduadas em ordem, podemos obter estes dados em ordem facilitando a delimitação de blocos básicos, resolvendo o terceiro problema.

9.3 BLAP: Proposta de Detecção de Blocos Básicos

Nesta Seção, é apresentada a implementação do BLAP. O mecanismo pode ser dividido em 3 partes: detecção, armazenamento e rotulamento.

9.3.1 Detecção

Com um estágio de graduação em ordem (CRISTAL et al., 2004), pode-se observar quais instruções são desvios para terminar blocos e delimitar a análise de blocos. Para observar as latências das instruções e caracterizar um bloco utilizam-se quatro registradores: MaiorLat (armazena qual maior latência achada até agora), Gargalo (armazena qual tipo de gargalo encontrado para a maior latência), Contador (conta ciclos em que nenhuma instrução foi graduada, ou seja, que a instrução no topo da fila de instruções bloqueou o estágio) e EndInicial (marca o endereço do desvio que levou a este bloco).

Em todo ciclo, observa-se se a instrução no topo da fila está pronta para ser graduada. Sempre que nenhuma instrução é graduada em um ciclo, incrementa-se o registrador Contador.

Caso uma instrução estiver pronta para ser graduada, deve-se observar se ela é um desvio, o que sinaliza o fim de um bloco. Se ela não for 1 desvio, também devemos observar apenas a primeira instrução graduada em cada ciclo, pois apenas ela pode ter bloqueado a graduação em ciclos anteriores. Outras instruções que não são desvios no mesmo ciclo podem ser ignoradas.

Caso seja a primeira instrução, o registrador Contador é comparado com o registrador MaiorLat, para observar se a instrução teve a maior latência detectada no bloco até agora. Se, e somente se, Contador for maior, atualiza-se o registrador Gargalo com o tipo da instrução no topo da fila e MaiorLat com o valor atual de Contador. Em qualquer caso, reseta-se o valor de Contador para iniciar a contagem de ciclos da próxima instrução a ser graduada no próximo ciclo.

Caso uma instrução seja um desvio, deve-se armazenar a informação do bloco. Primeiramente, é observado se o desvio foi corretamente previsto, pois desvios não atrasam diretamente o estágio de graduação, sendo necessária uma forma indireta de comparar o atraso destas instruções com as outras. Se o desvio não foi corretamente previsto, muda-se o valor do registrador Gargalo do bloco para *Brch* (tipo usado para desvios).

Após esta possível atualização do registrador, armazena-se o valor de Gargalo em um buffer, que escreverá o valor no BTB, usando o valor do registrador EndInicial como índice (o qual também necessita de um buffer para realizar-se um pipeline da escrita). Este registrador EndInicial então recebe o valor da instrução de desvio graduada para ser futuramente usado como índice para o próximo bloco. Como este é o fim de um bloco, todos os outros registradores, MaiorLat, Contador e Gargalo são resetados, para capturar o comportamento de um novo bloco.

9.3.2 Armazenamento

Para uso posterior dos bits de caracterização, é necessário armazená-los no BTB. De acordo com os problemas recorrentes em benchmarks, foram usados 2 bits para caracterizar cada bloco, expressando quatro características (*None*, *Brch*, *Mem*, *FP*).

9.3.3 Rotulamento

Para usar a informação efetivamente, foi criado um método genérico que permite a implementação simples de múltiplos mecanismos. Quando um desvio acessa o BTB no estágio de busca de instruções, o BTB é acessado para obter um possível endereço alvo, o qual possui uma informação de bloco relacionada. Então, ao acessar o BTB, carrega-se também esta informação para um novo registrador no estágio, chamado de *Característica do Bloco*.

A informação deste registrador é copiada em um novo campo de 2 bits em todas entradas de instruções em todos buffers do processador. Assim, qualquer mecanismo pode facilmente obter a característica de um bloco para observar o que pode ser melhorado nele.

9.3.4 Implicações no caminho Crítico

Todo o mecanismo exige hardware adicional, mas a implementação garante nenhuma implicação no caminho crítico.

Existe um caso específico que deve ser tratado pois é possível dois desvios graduarem em um mesmo ciclo de execução. Isto significa que o bloco iniciado pelo primeiro desvio do ciclo acaba no mesmo ciclo, ou seja, não tem atraso algum em suas instruções. Portanto, agregam0se as informações com o segundo bloco, ignorando os desvios subsequentes e tratando apenas o primeiro desvio achado no ciclo para delimitar blocos. Através de experimentos, é possível confirmar que a probabilidade de ocorrência de 2 desvios no mesmo ciclo é insignificante, acontecendo em menos de 1% dos desvios graduados.

Adicionalmente, armazenar a informação no BTB no mesmo ciclo poderia requerer um ciclo mais longo. Este é o motivo para o uso de um buffer, pois assim cria-se um pipeline para escrita da informação, dividindo o tempo entre os cálculos do mecanismo e o acesso real ao BTB. Também evita-se assim a necessidade de criação de uma nova porta para o BTB, pois o buffer pode esperar a porta estar desocupada por escritas oriundas de desvios no estágio de execução para assim escrever a informação referente ao mecanismo BLAP. Tal estágio extra não afeta o desempenho do processador, pois não aumenta o tempo de ciclo e não entra no caminho crítico de instruções (toda esta informação já foi graduada).

De todo resto, apenas são adicionados bits junto às instruções pelo caminho do processador, nenhuma lógica a mais é necessária para o funcionamento básico do mecanismo.

9.3.5 Custos de Hardware

Para implementação do mecanismo, são necessários 2142 bytes de armazenamento, três multiplexadores de 2 entradas de 2 bits, um multiplexador de 2 entradas de 8 bits, um somador de 8 bits e um comparador de 8 bits. Portanto, a área adicional de hardware por unidade de processamento é estimada em 206164 transistores. Na arquitetura Sandy Bridge, a qual possui 8 cores, isto significa 1649216 transistores. Tal arquitetura possui mais de 2.27 bilhões de transistores, portanto o mecanismo possui área adicional menor que 0.08% da área total.

9.4 Resultados

Para avaliar a caracterização do mecanismo, foram implementados dois trabalhos correlatos capazes de serem aproximados para usarem a informação. Como ambos mecanismos melhoram o desempenho da memória, foram usados benchmarks paralelos para exercício de pressão sobre a mesma.

O primeiro mecanismo é o preditor de criticalidade binário (CBP), criado e demonstrado por Ghose et al. (2013). O segundo mecanismo é o controlador de memória consciente de prefetches (PADC), criado e demonstrado por Lee et al. (2008).

O mecanismo CBP dá prioridade a instruções de *load* que atrasam o estágio de graduação. Como o mecanismo só mantém informação de *loads*, ele usa apenas uma tabela em SRAM sem *tags* de 64 bits por cores, a qual é resetada a cada cem mil ciclos para se adaptar à diferentes fases do programa. Tais *loads* armazenados na tabela são priorizados no controlador de memória.

O mecanismo PADC estende todas linhas de cache em 2 bits para calcular informações de poluição gerada por prefetches. Ao medir a precisão de prefetches a cada cem mil ciclos, o mecanismo decide se deve dar prioridades iguais para prefetches e requisições por demanda, ou se deve priorizar a demanda e apagar requisições de prefetch que demoram demais dada a sua poluição.

Criamos adaptações para ambos mecanismos. *BLAP-CBP* utiliza o mesmo controlador de memória do CBP, mas utiliza a informação de pacotes com valor de gargalo *Mem* do BLAP. Estas instruções são consideradas *críticas* no controlador de memória. Assim, a prioridade dos acessos se torna:

1. Prioridade a requisições críticas que são acertos na linha aberta de memória;
2. Prioridade a requisições normais que são acertos na linha aberta de memória;
3. Prioridade a requisições críticas que não são acertos na linha aberta de memória;;
4. Prioridade a requisições normais que não são acertos na linha aberta de memória;.

O *BLAP-PADC-8L* repassa a informação dos blocos para prefetches gerados pelos acessos à memória pertencentes a cada bloco. Para emular o conceito de remoção de requisições de prefetches, o *BLAP-PADC-8L* remove requisições de prefetches que demoram mais que a espera média de requisições feitas por demanda. Como agora existem as informações de quais requisições por demanda são críticas, quais prefetches são críticos, e se o endereço requisitado pertence à linha aberta da memória, faz-se uso de 2^3 níveis de prioridade. Os 8 níveis são:

1. Prioridade para requisições por demanda críticas na linha de memória aberta;
2. Prioridade para requisições de prefetch críticas na linha de memória aberta;
3. Prioridade para requisições por demanda normais na linha de memória aberta;
4. Prioridade para requisições de prefetch normais na linha de memória aberta;
5. Prioridade para requisições por demanda críticas em outra linha;
6. Prioridade para requisições de prefetch críticas em outra linha;
7. Prioridade para requisições por demanda normais em outra linha;

8. Prioridade para requisições de prefetch normais em outra linha;

Os resultados com os mecanismos apontaram médias de ganho de desempenho em relação à arquitetura base de 1.89% para o CBP, 0.80% para o BLAP-CBP, 3.10% para o PADC e 3.9% para o *BLAP-PADC-8L*, demonstrando que a granularidade de bloco básico é de fato tão útil quanto a granularidade de instruções para caracterizar acessos à memória.

Também foi observado no Capítulo 7 que o mecanismo escala para latências e pressões maiores na memória principal.

9.5 Conclusões

Nesta dissertação de mestrado foi apresentado o BLAP, um mecanismo capaz de caracterizar blocos para informar outros mecanismos sobre o comportamento de blocos. O mecanismo se baseia na ideia de que blocos podem ser delimitados no estágio de graduação do processador e caracterizados através da latência de suas instruções neste estado. Para demonstração de utilidade do conceito, foram implementadas duas técnicas do estado da arte, com as quais comparamos o desempenho quando usando suas próprias informações e implementações contra o uso de informações do mecanismo BLAP.

Em geral, o mecanismo desenvolvido junto com o BLAP demonstrou o melhor ganho de desempenho usando uma área de hardware reduzida quando comparado com a técnica PADC (LEE et al., 2008). A média de desempenho foi de 3.9%, ultrapassando 10% nos benchmarks que realmente dependiam de memória.

A informação providenciada pelo mecanismo demonstrou várias vantagens, como a progressão automática junto às fases de programas, menor área de armazenamento e variedade de caracterização. O uso do BTB demonstrou-se como forma eficiente de armazenar características referentes a blocos, devido à sua função original. A agregação de características em uma única característica permitiu com que a informação adicionada fosse mínima, embora no futuro tal informação possa ser expandida para abranger várias características de um único bloco.

Esta tese gerou duas publicações distintas: "Influência das Características de Processadores e Aplicações no Nível de Blocos Básicos" no WSCAD 2013 (Workshop de Sistemas Computacionais de Alto Desempenho), e "Profiling and Reducing Micro-Architecture Bottlenecks at the Hardware Level" no SBAC 2014.

Para o futuro, espera-se utilizar as informações de outras características para obtenção de melhorias de desempenho. Objetiva-se também melhorar o próprio mecanismo ao usar latências mínimas para filtrar quais blocos são relevantes e assim filtrar blocos que sofrem degradação de desempenho devido ao mecanismo.