Parallel Computing 54 (2016) 15–28

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A dynamic block-level execution profiler

Francis B. Moreira[a,*], Marco A.Z. Alves[a], Matthias Diener[a], Philippe O.A. Navaux[a], Israel Koren[b]

[a] Informatics Institute, Federal University of Rio Grande do Sul Porto Alegre, Brazil
[b] Department of Electrical and Computer Engineering, College of Engineering, University of Massachusetts at Amherst, USA

ARTICLE INFO

Article history:
Available online 12 February 2016

Keywords:
Basic Block Profiling
Processor Design
Memory Hierarchy Performance
HPC
Computer Architecture

ABSTRACT

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

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

1. Introduction

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

Current general-purpose processor designs only collect information at the instruction level. Although several research papers used basic block analysis, most performed it in software, even for hardware adaptations [5,6]. One of the few techniques that actually performed basic block analysis at the hardware level is the rePlay framework [7]. It analyzes the

* Corresponding author.
E-mail addresses: fbmoreira@inf.ufrgs.br (F.B. Moreira), mazalves@inf.ufrgs.br (M.A.Z. Alves), mdiener@inf.ufrgs.br (M. Diener), navaux@inf.ufrgs.br (P.O.A. Navaux), koren@ecs.umass.edu (I. Koren).

http://dx.doi.org/10.1016/j.parco.2016.01.010
0167-8191/

executing code to perform online code optimization and stores the collected information in a trace cache for future use. However, no bottleneck profiling was performed. Block profiling is usually done in software due to the high complexity of detailed profiling and the required analysis. Still, profiling in hardware is worth investigating, as it can efficiently generate relevant information regarding the program's execution without pre-analysis or source code modification.

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

This paper is an extended version of our previous work which described our Block-Level Architecture Profiler (BLAP) [8]. BLAP showed the potential of basic block-level online profiling by using the profile to modify different memory controller designs. In this paper, we introduce an improved profiler DBLEP, and analyze the dynamic behavior during the execution of the application and the mechanism sensitivity to its detection methodology and accuracy. Furthermore, we study the the reasons behind DBLEP's performance improvements.

The main contributions of this paper are as follows:

Characterization Mechanism: We introduce DBLEP, an efficient mechanism to characterize applications at the basic block level during their execution.

Accuracy Enhancement: The accuracy of the proposed mechanism is further increased by using finite-state machines for the information collected, allowing unstable blocks to be re-characterized. This in turn increases the performance gains of the memory controller that relies on DBLEP's profiling.

Performance Analysis: DBLEP's performance improvements are analyzed, showing that it achieves reduced demand-based service time in the main memory, leading to overall fewer stall cycles while the processor waits for memory requests.

2. Correlating microarchitectural bottlenecks to performance

In this section we explore the relationship between basic blocks and processor performance. We use here a relaxed definition of a basic block [2,3]. A basic block is a code segment with a single point of entry and a single point of exit. Thus, every basic block ends either with a branch instruction, or an instruction that is the target of another branch instruction and therefore indicates the beginning of a new basic block. This enables mechanisms based on basic blocks to identify program phases automatically, as a program's phase is characterized by the blocks being executed [6]. Our definition allows for multiple entry points, as it is not possible to efficiently detect the beginning of a block which was not targeted by a branch. Moreover, since we use the Branch Target Buffer (BTB) to store the block's behavior (as explained in Section 4), we collect information only for blocks that are executed following a taken branch. The implications of this implementation are further discussed in Section 4.

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

2.1. Complete execution correlation

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

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