Journal of Systems Architecture 90 (2018) 85–93

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Detecting the phase behavior on cache performance using the reuse distance vectors

Shan Shen, Ming Ling*, Yongtao Zhang, Longxing Shi

National ASIC System Engineering Technology Research Center, Southeast University, Nanjing, Jiangsu 210096, China

ARTICLE INFO

Keywords:
Phase behavior
Reuse distance
Cache performance
Simulation points

ABSTRACT

Previous studies proposed several code signatures, with large vector dimensions and time-consuming profiling processes, to detect phase transitions of the overall processor performance. However, there still lacks an effective and efficient method to detect and leverage the phase characteristics of memory access. In this paper, we propose the reuse distance vector (RDV), a new metric that tightly coupled with the cache performance, to summarize the phase behavior in the memory hierarchy. Different from the commonly seen huge dimensionality of other code signatures, RDVs are measured at very low dimensions. Meanwhile, the profiling overhead can be further reduced by our sampling technique. Based on RDVs, we can pick simulation points from the whole program via a clustering method to act as the representative subset to reduce the time consumption of cycle accurate simulations.

Using the simulation points found by RDVs, the average relative error of cache miss rate estimation is as low as 1.08%, which outperforms the accuracies of BBVs and EIPVs by 79% and 22% (all compared methods use the same number of simulation points that takes merely 0.4% of the whole program). Meanwhile, the average errors of MLP and the cache miss service time are only 0.9% and 1.8%, respectively.

1. Introduction

Programs have different behaviors, which can be measured by different performance metrics, during different code region execution. A same code region can be executed several times when existing within a loop or a frequent called subroutine. Thus, the periodic behavior, or phase behavior, was observed and studied by many previous studies [1,2,6,8]. A phase is a set of intervals with similar behavior in a program's execution, regardless of their temporal adjacency [8]. Accurately capturing phase behavior by ISA-level information, which is independent of the underlying architectural details and performance, allows us to partition an entire execution into phases. Phase information for the same application can be reused when performing a design space exploration (DSE) or guiding optimizations across different architecture configurations. Additionally, this phase information can be also applied to determine when to do re-configuration for advanced self-adaptive micro-architectures, such as caches [23–25,34] and pipelines [35].

The existing phase detection methods first divide the entire program execution into continuous time intervals (also called profiling intervals). Then they collect specified instruction information as code signatures of each interval. For example, a basic block [2] is defined as a single-entry, single-exit section of code, i.e. the code section between two branches, without any internal control flow. The execution stream of an interval is structured into a basic block vector (BBV), where each element in the vector is the executed frequency of the corresponding basic block in the program. Extended instruction pointer vectors (EIPVs) [6] are similar to BBVs, but instead of capturing execution frequencies of basic blocks, they capture execution frequencies of individual instructions. Similarly, sparse conditional branch vectors (Sparse CBRVs) [10] count the number of conditional branches in each execution interval.

Although the previous methods successfully detect and leverage the phase behavior, they focus on the overall processor performance (e.g., IPC) rather than the memory system performance. Memory system designers are eager to find the simulation points [2], equipped with more representativeness of memory access, to accurately evaluate the performance of memory subsystem while keeping the simulation overhead under the budget. On the other hand, for the self-adaptive memory architectures, an accurate and efficient memory performance metric is also needed to detect the phase changing of the memory performance [24], instead of the overall processor performance which actually may mislead the reconfiguring. Last but not least, even the same instructions can have very different memory system behavior given different types of program inputs [11], previous approaches which simply consider instruction behavior cannot capture memory performance phase transitions and will over-homogenize the program execution time.

* Corresponding author.
E-mail addresses: shanshen@seu.edu.cn (S. Shen), trio@seu.edu.cn (M. Ling), ytz@seu.edu.cn (Y. Zhang), lxshi@seu.edu.cn (L. Shi).

https://doi.org/10.1016/j.sysarc.2018.09.001
Received 1 June 2018; Received in revised form 16 July 2018; Accepted 3 September 2018
Available online 13 September 2018

Furthermore, due to the large dimensionality of BBVs and EIPVs, an additional transformation must be applied. Direct hashing the original data into a small-sized vector hurts the distinctiveness of signature vectors [3]. Thus, in [2,6], each signature vector with D elements needs to be multiplied by a $D \times M$ random projection matrix to reduce its size to M. For the total N intervals, it needs up to $D \times M \times N$ times of floating-point multiplications. Moreover, storing such large vectors and the projection matrix wastes the memory resource even for an offline analysis.

To solve these problems, we propose the reuse distance vectors (RDVs) to analyze phase behavior exhibiting in the memory hierarchy. This paper further demonstrates the effectiveness and the efficiency of RDVs by applying them to the simulation point selection. The contributions of this paper are:

- We propose the reuse distance vectors (RDVs), a low-dimensional (16 dimensions in this paper) hardware-independent metric, to summarize the memory behavior of an arbitrary section of program execution. RDVs are collected at a low overhead by using straightforward operations and a random sampling technique.
- We study the memory phase behavior from three perspectives of cache performance: the cache miss rate, the memory level parallelism (MLP) and the cache miss service time (service time for short). Through detailed experiments, we have proven that RDVs are not only strongly correlated with cache misses, but also with MLP and the service time as well.
- By using the RDVs, we demonstrate a unified method to select the best simulation points for DSE of memory systems. The overall accuracies of the performance predictions from the simulation points found by our approach and those chosen by the prior methods are evaluated. Meanwhile, the overheads of different signature extractions are also compared.

The rest of the paper is organized as follows. Prior studies are summarized in Section 2. Section 3 introduces the definition and collection method of RDVs/sampled RDVs (SRDVs). We also show the strong correlations between RDVs and the memory performance metrics. Section 4 describes our experimental setup. Section 5 compares the performance of phase clustering based on the existing phase detection methods, as well as the accuracies of simulation points found by them. This section also analyzes the influences of vector dimensionality and the information loss caused by our sampling approach. At last, we conclude this paper and introduce our future work in Section 6.

2. Related works

Sherwood et al. first proposed BBVs [1] that measured the periodic behavior in programs in terms of several different architectural metrics. In the following work [2], they used offline k-means clustering to classify the similar behavior execution intervals of the program into phases (clusters), and selected the representative simulation points based on the phase analysis to reduce the simulation workload. However, before the clustering, BBVs must be multiplied by a random projection matrix to reduce their dimensions to an acceptable range (as discussed in Section 1). Sherwood et al. [3] and Lau et al. [4] also extended their work to a hardware scheme to do the online phase tracking and prediction with a sacrifice of precision. Perelman et al. [16] extended those prior works to the scenarios of multi-threaded workloads.

Dhodapkar et al. [5] used instruction working set of the program (i.e. the set of instructions executed in a fixed interval of time) to detect phases. In [5], a summary on phase detection methods was given. However, it needs an extremely large bit vector [5] to contain all data accessing information in the granularity used in this paper.

Davies et al. [6] proposed the EIPVs collected by Intel VTune [18] Performance Analyzer to faithfully represent the program execution within a given interval. However, the drawbacks of EIPVs are the same as those of [2]. Annavaram et al. [7] used a regression tree to classify the benchmarks into four quadrants based on their CPI variances. Lau et al. [8] found that the information loss of sampled EIPVs weakened the correlation between code signatures and hardware metrics in [7]. They improved the sampled EIPV-based method by mapping each EIP to its corresponding loop or procedure. Yet, the mapped EIPVs still performed worse than the full BBVs in their experiments.

Sembrantet al. [10] proposed a new method called Conditional BRanch Vectors (CBRV). Using Linux-perf_events [17], they extracted the sparse CBRVs at extremely low overhead. They developed ScarPhase, a new online phase detection library, to further reduce the running time of the profiler. Not surprisingly, the predicted CPI had a large error (the maximum error is larger than 20%) due to the sparse sampling compared to our RDV-based method.

Regarding the memory system phase detection, Balasubramonian et al. [9] detected phase changes by monitoring performance counters that collect information of cache misses, CPI and branch frequencies. Ding et al. [11] first proposed the LRU stack distance histogram (SDH) as a data locality signature to predict the program behavior with different data inputs. Shen et al. [12] leveraged Wavelets and Sequitur to build a hierarchy of phase information to represent program's behavior patterns based on the LRU SDH. However, compared to the reuse distance extraction, collecting stack distance has a significantly larger overhead due to the requirement of recording all unique addresses. Ipek et al. [13] discussed an online solution of phase detection for distributed shared memory systems by introducing data distribution vectors (DDVs). Unfortunately, DDVs cannot work without the help of BBVs, which brings a larger computation complexity and storage usage.

An orthogonal method to help memory system evaluation is trace synthesis, which reconstructs the pattern of memory access after profiling. Maeda et al. [36] introduced hierarchical reuse distance (HRD) to represent the data locality and traffic at different cache block granularities with good accuracies. But HRD model couldn't provide the estimation of IPC or other performance metrics.

In this paper, we propose a new code signature, reuse distance vectors (RDV for short), based on the reuse distance (RD), which is widely utilized in benchmark analyses [14,15] and cache analytical modeling [20,21,38] for its low-cost extraction and effectiveness. Compared to prior metrics, RDV is an efficient and more accurate code signature dedicated to the memory performance phase detection.

3. RDVs and phase behavior in the memory hierarchy

We introduce the concept of RDVs and describe how to collect them in this section. To further lower the extracting overhead, a sampling approach for RDVs is also explained. In addition, by visualizing the phase behavior of cache performance metrics, we will show the strong correlations between RDVs and the memory phases.

3.1. Definition and collection of RDVs

Reuse distance. In this paper, a reuse distance (RD), defined same as [21], is computed as the number of memory accesses in a reuse interval, where the reuse interval is the time duration between two successive memory accesses to the same cacheline.[1] For example, a sequence of memory references from left to right, *ABCBCBA*,[2] forms a reuse interval of two '*A*'s, in which the RD of the second *A* is 5. A reuse distance histogram (RDH) is distribution of RDs observed in a program within a given time duration. The height of a bin RDH(k) indicates the number of references that have a reuse distance of k. For the above example, the

[1] Different from the work [34,36,37], the term "reuse distance" in this paper is not equivalent to the LRU stack distance. The LRU stack distance is actually the number of accesses to unique addresses made since the last reference to the requested data.

[2] All the addresses are cacheline-aligned in this paper. The sub-sequence *BCB* and *CBC* also construct two and one smaller reuse intervals for the references *B* and *C* respectively.