The Egyptian International Journal of Engineering Sciences and Technology

Vol. 20 (July 2016) 1–9

http://www.eijest.zu.edu.eg

A Run-Time Program Phase Detection Technique for Optimizing Per-Phase L2 Cache Demand

Ibrahim E. Ziedan, Hazem I. Shehata and Shaymaa M. Seraga*

Computers and Systems Department, Zagazig University, Zagazig 44519, Egypt.

ARTICLE INFO

Article history:
Received: 31 March 2016
Received in revised form: 25 May 2016
Accepted: 1 June 2016
Available online: 13 July 2016

Keywords:
Program Phase Detection
Basic Block Vectors
Working Set Signatures
Phase-based Optimization
Cache Access Pattern
Performance Monitoring Unit

ABSTRACT

Understanding program behavior is at the foundation of computer architecture and program optimization. Programs pass through different behaviors where their performance characteristics and hardware resource requirements vary. Program phase detection and classification research aiming to understand the program time-varying behavior, can unlock a lot of phase-based optimizations which are specially tailored to improve the performance of each individual program phase. In this paper, we introduce an efficient run-time phase detection and classification technique, based on tracking changes in the L2 cache access pattern of different portions in the program execution. The proposed technique monitors a running program and keeps track what phase the running program is currently executing, with no need to recompile the tracked program, and with execution time overhead of 4%, on average. *Performance Monitoring Unit* (PMU) is exploited to sample the memory addresses causing L1 data cache misses. This profiling data is used to construct the *Cache Access Signature Vectors* (CASVs) that accurately reflect the L2 cache access patterns for each interval of execution. By comparing CASVs, the proposed technique classifies the program into a set of stable phases with high degree of intra-phase homogeneity. Our evaluation shows that phase changes detected by our technique have strong correlation with the variation in *Instruction Per Cycle* (IPC). Furthermore, our technique can contribute in reducing L2 cache miss rates, and optimizing L2 cache utilization, through its direct capability of estimating per-phase L2 cache demand.

1. Introduction

Understanding the program behavior is at the foundation of computer architecture and program optimization. Many programs have different behavior over the program's complete execution time. The way a program's execution changes over time falls into repetitive portions called phases.

Understanding the time-varying behavior of programs, can unlock a lot of optimization opportunities; and tend to improve not only the programs' average performance but also the performance of individual phases of the program execution.

* Corresponding author. Tel.: +2-050-695-1082.
E-mail address: shserag@zu.edu.eg

Examining the run-time behavior of programs was an interest of a lot of researchers [1],[2],[3]. They divided the program's execution into non-overlapping intervals. An interval is a contiguous part of execution (fixed number of instructions) of a program. The program phase could be defined as a set of intervals within a program's execution that have homogeneous behavior, including the cache miss, Instruction–Per-Cycle (IPC) and power consumption, and similar resource requirements, regardless of temporal adjacency. A phase can reoccur multiple times through the program's execution. The program phase classification could be defined as partitioning the program into groups of intervals with similar behavior, using a certain similarity metric and similarity threshold.

Recent researches classify and predict phases in program execution using varieties of techniques. The approach of Sherwood [4] focuses on identifying phase behavior by tracking executed code. In another approach of Dhodapkar and Smith [5], phase behavior can be detected by examining a program's working set. And other techniques like those make use of conditional branch counts [6], data reuse distance [7].

Program Phase identification and detection can be exploited to save energy by dynamically reconfiguring caches [5],[6],[8], to guide compiler optimization [9],[10], to assign processes to cores in a heterogeneous multi-core architecture [11], and to reduce the power consumption in mobile processors [12]. All of these techniques take advantage of program phase behavior.

In this paper, we proposed a run-time technique to detect and identify the stable program phases, and use that to calculate the L2 cache demand for each program phase, in order to optimize L2 cache utilization. The proposed technique is called "Cache Set Signature". In which, the main idea behind the Basic Block Vector (BBV) [4] and the Working Set Signature [5] methods are combined to increase the program phase detection accuracy and exploit the advantages of them both.

The rest of the paper is laid out as follows. In Section 2, prior work related to phase-based program behavior is discussed. Section 3 describes our program phase detection technique. Experimental setup and experimental results can be found in Section 4. Finally, section 5 concludes the paper.

2. Related work

This section presents an overview of the previously proposed techniques for detecting program phase changes.

Peter Denning [13] defined "the working set of information $\boldsymbol{W(t,\tau)}$ of a process at time $\boldsymbol{t}$ to be the collection of information referenced by the process during the process time interval $(\boldsymbol{t-\tau, t})$". Typically the units of information are considered to be memory regions of fixed size, such as pages.

Dhodapkar and Smith [5] defined a phase as the maximum interval over which the working set remains more or less constant. A working set signature is designed to work as the compact representation for a complete working set of the program intervals.

Unlike early working set researchers who were interested in program paging behavior, Dhodapkar and Smith [5] chose the working set elements to be of cache line granularity, because it is suitable for dealing with multi-configuration units (e.g. caches and predictors) that work at the same granularity. The non-overlapping windows (intervals) are used instead of sliding window which was used in paging studies.

The method to form a working set signature is shown in Figure 1. The working set signature consists of an $\boldsymbol{n}$-bit vector, where each entry of that vector corresponds to a certain memory block. The vector is cleaned at the beginning of each interval**.** $\boldsymbol{b}$: is the least significant bits of the program counter , where $\boldsymbol{2^b}$ is the number of instructions contained in the cache line. $\boldsymbol{m}$ bits are selected from the program counter, and are used to address 1 bit in the $\boldsymbol{n}$-bit signature via a pseudo-random hash function. A bit in the working set signature vector is set when the corresponding instruction block is touched.

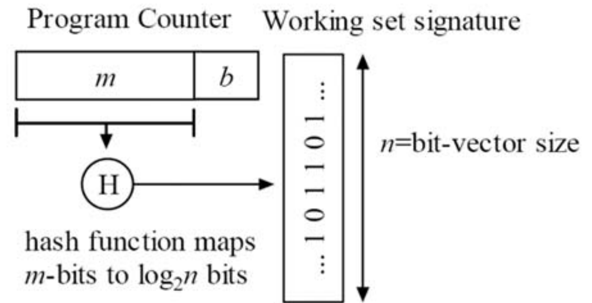

Fig. 1: Mechanism for collecting the working set signature vector [5]

A metric called relative signature distance is used to detect the working set change. Given two working set signatures to compare S_1, and S_2, the total number of ones in the exclusive OR (XOR) of the signatures is divided by the total number of ones in the inclusive OR (OR) of the signatures to obtain a ratio called a relative signature distance. Where num_of_"1"bit_in() represents the function that counts the number of "1" bits in the bit vector [5].

$$\delta = \frac{num_of_"1"bit_in(S_1 \oplus S_2)}{num_of_"1"bit_in(S_1 + S_2)} \quad (1)$$

If the working set signatures are very similar, the relative signature distance δ is close to zero. Consequently, the two intervals are classified into the same program phase. If the working set signatures are very different, the relative signature distance δ is close to one. Hence, the two intervals are classified into different program phases. The threshold value is obtained experimentally by comparison with several benchmarks.

Dhodapkar and Smith [5] estimated the working set size by counting the number of non-zero bits in the signature. Their technique directly configures, i.e., without a trial and error process, certain hardware whose performance depends on the working set size, including caches and branch predictors . The optimal configurations along with their corressponding working sets signatures could be stored. When a working set repeats itself during program execution, the optimal configuration could be directly set.

Sherwood [2],[4],[14] developed the Basic Block Vectors (BBVs) technique in order to capture information about the way the program changes its behavior over time. A basic block is defined as a section of code that is executed from start to finish with one entry and one exit.

A Basic Block Vector (BBV) is a single dimensional array with one element in the array for each basic block in the program. Each element in the array is the number of times the corresponding basic block has been touched during an interval of execution, multiplied by the number of instructions in that basic block [15].

In the BBV technique proposed by Sherwood, determination of program behavior is only a function of what the code is doing at a particular time and how often, and is independent on any hardware statistics or any architecture parameters.

The method to form the Basic Block Vector is shown in Figure 2. Accumulator buckets are the elements of the Basic Block Vector . Sherwood found that 32 bucket (32 basic blocks) is sufficient to distinguish between different phases, even for more complex programs. So every branch PC is hashed to one of 32 accumulator buckets (entry). Each accumulator bucket is a large saturating counter that gets increased by the number of instructions from the last branch to the current branch being processed. Updating the accumulator is performed once for every branch executed.

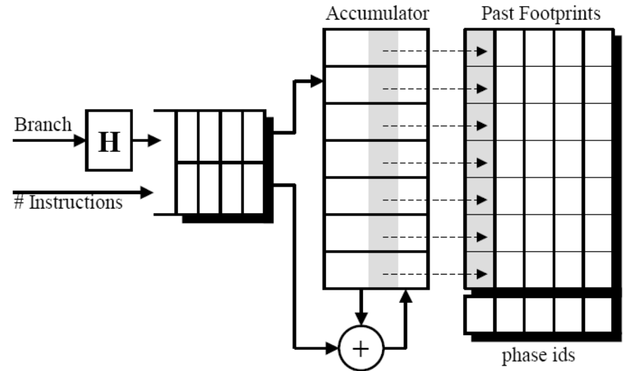

Fig. 2: Mechanism for collecting the Basic Block Vector [14]

At the end of each interval composed of 10 million instructions [14], a BBV corresponds to that interval is produced, and the classification stage begins. The Past Footprints table keeps only single BBV for each unique phase ID, as a representative of that phase. The tabled is looked up for a match with each currently produced BBV. If there is a match, the current interval's vector is classified into the same phase as the past footprint vector, and is not inserted into the past footprint table. If there is no match, then a new phase is detected, and hence a new unique phase ID will be created.

Manhattan distance is used in BBVs comparison [4]. The Manhattan distance is the distance between two points if the paths can only be taken in parallel to the axes. Such distance is computed by summing the absolute value of the element-wise subtraction of two vectors.

For any two $\boldsymbol{D}$-dimensional vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, the distance can be computed as:

$$\text{ManhattanDist}(a, b) = \sum_{i=1}^{D} |a_i - b_i| \quad (2)$$

A phase change is detected and a new phase ID is generated, when the Manhattan distance between consecutive BBVs exceeds a preset threshold.

Dhodapkar and Smith [5] used a bit vector to track the working set of the code (the code blocks which are *touched),* during a particular interval. Whereas in Sherwood [15] technique the *proportion* of time spent executing in each code block is tracked. This is an important distinction. Because in complex programs, there are many instruction blocks that execute only intermittently. When tracking the pure working set, these infrequently executed blocks can disguise the frequently executed blocks that dominate the behavior of the application. In other words, by tracking the frequency of code execution it is possible to distinguish important instructions (basic blocks) from a sea of infrequently executed ones [14].

Dhodapkar and Smith [16] conducted a comparative study among techniques used in detecting program phase changes. They concluded that BBV techniques performs better than the other techniques, providing higher sensitivity and more stable phases. However, the instruction working set technique yields 30% longer phases than the BBV method, although there is less stability within phases.

3. The Cache Access Signature technique

In this paper, we propose a technique to: (1) detect and identify the stable program phases, (2) estimate and tabulate the L2 cache demand for each program phase. It's a run-time technique with acceptable and minimal execution time overhead and memory cost.

Program phase detection based on working set signatures method has a direct capability to estimate the program phase cache demand (using working set size). On the other side, program phase detection based on BBVs method contains more information compared to working set signatures one, and produces more sensitive and efficient phase classification.

The proposed program phase detection technique combines the two methods, to exploit the advantages of them both, to increase the program phase detection accuracy, and to estimate the L2 cache demand for each phase.

3.1. Investigating the relation between program phase changes and L2 cache demand

The program cache demand varies according to different phases during the program execution time. David Tam [17] used the changes in the L2 cache miss rate as an indicator of phase transitions. Because it directly reflects the changes in the application's cache usage.

Figure 3 shows the impact of phase changes on the miss rates of mcf (one of SPEC 2000 Benchmarks) as an example. The measurements are taken by running the application 16 times, each time with a different L2 cache size, and using the Performance Monitoring Unit (PMU) to measure the cache miss rate. The x-axis shows the execution progress of mcf in terms of the number of completed instructions. The y-axis indicates the L2 cache miss rate in terms of the number of misses per thousand completed instructions (MPKI). Thus, the graph shows how the L2 cache miss rate varies during program execution. mcf oscillates between two phases repeatedly, a phase with relatively high L2 cache miss rates and a phase with relatively low L2 cache miss rates [17].

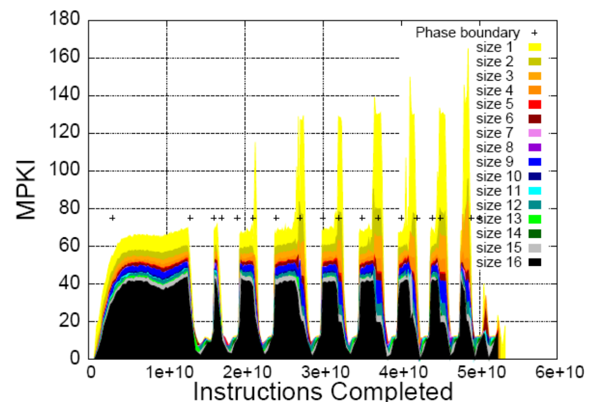

Fig. 3: Miss rate curves at different L2 cache sizes reflect the program behavior [17]

This graph also indicates how the L2 cache miss rate diminishes as the size of the L2 cache partitions is increased. For example, the time-varying miss rate of the size 16 configuration is always lower than the time-varying miss rate of the size 15 configuration, which is always lower than the time-varying miss rate of the size 14 configuration, etc. So if we have the ability to estimate the L2 cache size required by each program phase and allocate the required size to the corresponding phase, there will be a great potential to minimize L2 cache misses.

From David Tam studies [17], we can conclude that tracking L2 miss rates is sufficient to partition the program into phases. Consequently, we suppose that tracking L2 cache accesses is also sufficient to detect and classify the program into different phases.

In the proposed technique, the pattern or the way by which the program uses the L2 cache is tracked. In other words, the addresses causing L1 data cache misses, which consequently lead to L2 cache accesses, are captured. Then those captured addresses are used to construct a vector called Cache Access Signature Vector (CASV). Costructed CASVs reflect the pattern of L2 cache usage of each phase.

3.2. Using Performance Monitoring Unit for tracing and sampling memory addresses

Most modern microprocessors have Performance Monitoring Units (PMUs) with integrated Hardware Performance Counters (HPCs) that can be used to monitor and analyze performance in real time.

HPCs allow for counting microarchitectural events, such as branch mispredictions and cache misses. They can be programmed to interrupt the processor when a specified number of a certain event occurs. Moreover, PMUs make various registers available for inspection, such as the memory addresses causing cache misses [18].

On Intel x86 processors, it may be possible to use the Intel Precise Event-Based Sampling (PEBS) feature of the PMU to capture the required information [18]. Data Linear Address (DLA) is one of PMU new features and PEBS improvements in Haswell architecture. DLA enables capturing of the data linear address and data source of load/store memory instructions, at which certain precise memory access events, such as a data cache miss or a TLB miss, occur.

The proposed technique tracks the L2 cache access patterns to detect and classify different program phases. To monitor these patterns, we choose "MEM_LOAD_UOPS_RETIRED.L1_MISS", one of the precise memory access events, which is supported by both PEBS and DLA facilities [18], [19]. Each time the number of L1 data cache misses reaches a specific threshold (sampling period), the respective HPC overflows. Then, PEBS mechanism causes a sample to be captured including the linear data address at which L1 data cache miss occurred.

Perf_events tool, a user-space tool under Linux kernel, is used to manage HPCs, to capture the memory addresses causing L1 data cache misses, and to extract other information from the PMU. Perf_events tool is extremely useful as it barely adds any overhead [20].

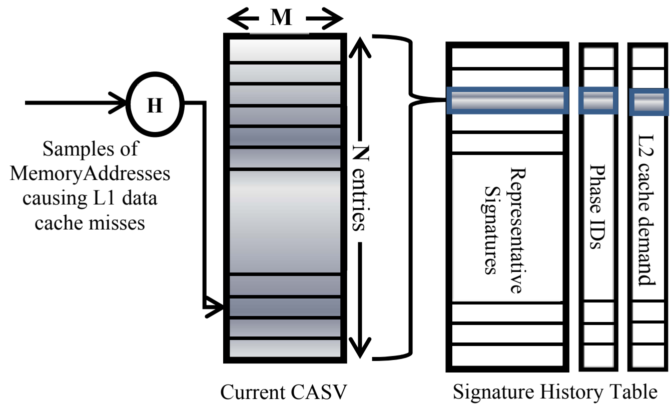

Fig. 4: The run-time Cache Access Signature technique for phase detection and classification

3.3. Constructing Cache Access Signature Vector

Figure 4 describes the Cache Access Signature Vector. CASV is a single dimensional array consisting of N elements. There is a single element corresponds to each set in the L2 set-associative cache. Therefore, N equals the number of L2 cache sets.

The program execution is divided into fixed-size intervals, with fixed number of instructions I in each. A single CASV is constructed for each interval.

Memory addresses, at which L1 data cache misses occurred, are tracked by using PEBS and DLA features of the PMU. Then a hash function determines the L2 cache set referenced by each address, and then increments the vector element corresponding to the referenced L2 cache set.

Each vector element is an M bits saturating counter that represents number of times the corresponding L2 cache set is referenced during an interval of execution.

At the end of each interval, the CASV is constructed. Such vector reflects the pattern of L2 cache usage, including not only which L2 cache sets are referenced during the interval, but also the access intensity of each set. The intervals that share patterns or have similar signatures are classified into the same program phase.

3.4. Signature History Table and phase classification

For phase classification, a table is maintained to hold the information related to the unique phases, previously encountered during the program execution. This table is called the Signature History Table.

In which, a single CASV is kept for each phase, along with a unique phase ID for that signature. Each CASV stored in the signature history table serves as a representative of its corresponding phase. The L2 cache demand for each phase is also tabulated.

3.4.1. The Similarity Metric

We define the similarity of two CASVs $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$, corresponding to two intervals, as their dot product:

$$\text{Similarity}(S_1, S_2) = \sum_{i=0}^{N-1} S_1[i] \times S_2[i] \quad (3)$$

Where $\boldsymbol{S_x}$ is the CASV of the interval $\boldsymbol{x}$, and $\boldsymbol{i}$ is the $\boldsymbol{i^{th}}$ entry of the vector consisting of $\boldsymbol{N}$ entries. There are two reasons behind choosing this metric for similarity. First, it automatically takes into account only those entries where both vectors have non-zero values. Note that $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$ have non-zero values in the same entry, only if the same L2 cache set is referenced during their corresponding intervals. Second, the metric takes into account the intensity by multiplying together the number of times the target L2 cache set is referenced during both of the compared intervals.

Larger dot product indicates more similarity between the compared intervals. On the other hand, totally dissimilar intervals should result in zero values for their dot product. The value of the similarity threshold is experimentally set, so as to distinguish accurately among phases and produce more homogenous ones.

3.4.2. Classifying the Cache Line Signatures to a Phase ID

In the classification process, we compare the current interval signature to a set of representative signature vectors stored in the Signature History Table illustrated in figure 4. If there is a match, we classify the current interval of execution into the same phase as the phase of the matched representative vector, and the current vector replaces the matched table entry. If there is no match, then a new phase is detected, and hence a new unique phase ID is created and inserted to the Signature History Table along with its representative signature vector.

3.4.3. First similar or best similar

In order to improve the homogeneity of our phase classifications, when multiple signatures satisfy the similarity threshold, we choose the most similar representative signature, not the first similar one. In other words, we choose the phase ID whose representative signature is most similar to the CASV of the current interval.

3.4.4. Stable and transition phases

An important aspect of phase classification is how to handle phase transitions. During the program execution, the program behavior passes through some stable long phases and other intervals of transition or unstable phases in between the stable ones. These transitional intervals do not last for very long, and infrequently occur, so it is not worthwhile to optimize for their behavior.

We group all transitional intervals into a single phase called *Phase Zero*. In order to reduce the number of unique phase IDs generated.

Each entry in the Signature History Table is augmented with a small counter called Interval Counter that counts the number of intervals which are classified into each phase. If the current signature has no match, a new signature is added to the Signature History Table, its Interval Counter is set to zero, and phase ID zero is assigned to that signature. The Interval Counter is increased every time another interval is classified into the corresponding phase. Phase Zero has only a real phase ID, when the Interval Counter value exceeds a certain threshold. To achieve more accurate phase classification, number of intervals which change the phase from transition phase to a stable one (Transition Threshold), is a subject of experimental results.

3.5. Estimation for per phase L2 cache demand

In the Signature History Table, representative signatures reflect the pattern of L2 cache usage of each phase, so the technique introduces a direct method to estimate the L2 cache demand for a certain program phase including all of its intervals that have high degree of L2 cache usage pattern similarity.

L2 cache demand or number of cache sets required by any program phase is equal to the number of non-zero elements in the corresponding representative CASV. Allocating L2 cache size per phase basis is beyond the scope of this paper, and it is a point of future research.

4. Experimental Results

4.1. Experimental Setup

Our experiments and analysis are performed on Intel Core i7 machine running Linux kernel (version 4.2.0-21-generic). The hardware specifications are described in Table 1.

Table 1. Intel Core i7 specifications

Item	Specification
# of Chips	1
# of Cores	2 per chip
CPU Core	Inter Core i7 – 4510U, 2.6GHz, 2-way SMT
L1 ICache	2 × 32 KB, 8-way set associative, 64 bytes line size
L1 DCache	2 × 32 KB, 8-way set associative, 64 bytes line size
L2 Cache	2 × 256 KB, 8-way set associative, 64 bytes line size
L3 Cache	4 MB, 16-way set associative, 64 bytes line size
RAM	8GB

A simple benchmark is developed to test the proposed technique on. The testing program exhibits different behaviors, i.e. some small portions are CPU-bound code, and other portions, which dominate most of the program execution, walk through memory using different fashions. The variance introduced through the testing program aims to: first, access different memory locations. Second, generate different miss rates on the level of L1 data cache.

4.2. Demonistrating the Correlation between IPC variation and the changes in L2 cache access pattern

As Figure 5 shows, the IPC variation in the program (the upper graph) has a strong correlation with the phase changes detected by the proposed technique (the lower graph). Consequently, our technique is able to not only detect phases that reflect the program behavior but also track the boundaries of behavior changes.

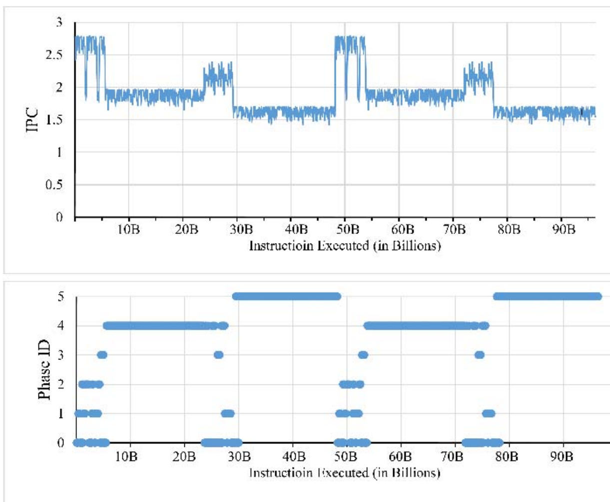

Fig. 5: Correlation between changes in the IPC and phase changes detected by the proposed technique

4.3. The intra-phase homogeneity evaluation

Homogeneous phase means that architectural metrics such as IPC, should have quite similar values at all the intervals that the phase occurs in. To quantify the extent to which the proposed technique achieves this goal, the homogeneity of IPC on a per-phase basis is measured for the phases detected at run-time.

IPC is calculated for each interval of execution. Then, average IPC is calculated for each phase, i.e. for all intervals which are classified into that phase. The standard deviation in IPC values, in addition to their average, are shown in Table 2.

When we compare the standard deviation of Phase 4 or Phase 5, the longest phases detected by our technique, with that of the entire program (denoted by "Full"), we can see that after dividing the program into phases, each phase has a little variation within itself.

The entire testing program has IPC CoV (standard deviation / average) of 17.2%. By dividing it up into different phases, we achieved overall IPC CoV of 5.6%. Therefore, our technique demonstrates its ability to detect phases with high intra-phase homogeneity.

Table 2. Examination of the per-phase homogeneity

Phase ID	Full	Phase 4	Phase 5
Percentage of Execution	100%	40.25%	38.69%
IPC (Average)	1.978	1.915	1.734
IPC (Standard Deviation)	**0.34**	**0.105**	**0.153**
CoV of IPC (%) (Standard Deviation / Average)	17.189%	5.48%	8.824%

4.4. Impact of the similarity threshold

The similarity threshold determines how much two intervals can deviate from each other without being classified into separate phases. Changes in the similarity threshold will affect the number of generated phase IDs, and the intra-phase homogeneity.

Figures 6 and 7 respectively show the number of detected phases and the IPC CoV values at different similarity thresholds. The number of detected phases increases as we increase the similarity threshold. At similarity threshold of zero, 17.2% is the highest value for the overall CoV calculated when treating the whole program as a single big phase. Then, the overall CoV decreases indicating that we can produce more homogeneous phases by using higher similarity thresholds. By fine tuning, the suitable similarity

threshold value is picked up, which produces few phases with low overall CoV.

4.5. Impact of the transition threshold

As previously explained in section 3.4.4, it is important to identify the infrequently occurring behaviors and isolate their transitional intervals into a single transition phase which we called *Phase Zero*. A new phase ID is only generated, if the number of times in which a representative signature appears, exceeds a preset Transition Threshold. Otherwise *Phase Zero* is generated.

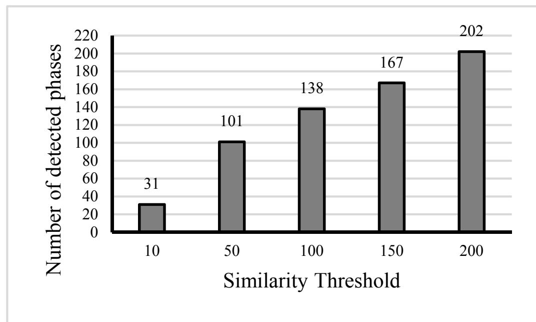

Fig. 6: Number of detected phases at different similarity thresholds

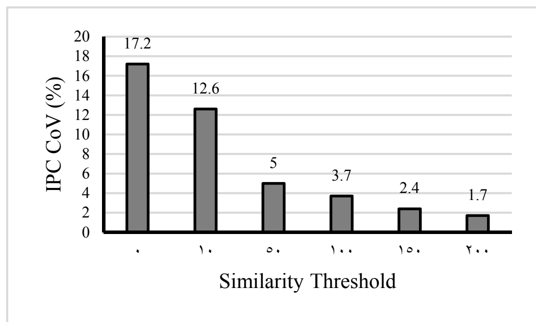

Fig. 7: The overall CoV in IPC changes according to different similarity thresholds

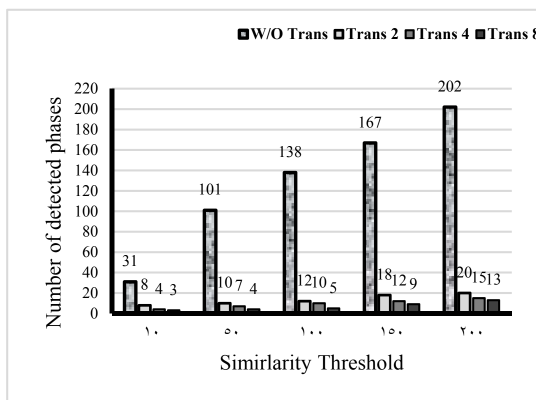

Fig. 8: The impact of applying *Transition Phase* utility on the number of detected phases

As illustrated in Figure 8, the number of detected phases is inversely proportional to the value of the transition threshold. “W/O Trans” means that the phases are detected without applying the utility of the transition phase, i.e. once a new signature appears, a new phase ID is generated. “Trans 2” means that the representative signature must appear twice before it is considered stable, and a new phase ID is assigned to it. After applying the transition phase utility, the number of detected phases is significantly reduced. We can see that tens of phase IDs are generated instead of hundreds.

5. Conclusion

This thesis introduced an efficient run-time phase detection and classification technique, that is based on tracking changes in the L2 cache access pattern of different portions in the program execution. Our evaluation shows that the average execution time overhead was about 4%.

We examined how the Performance Monitoring Unit (PMU) and its related tools could be used to sample the memory addresses causing L1 data cache misses. This data collected through lightweight hardware monitoring, is used to construct the Cache Access Signature Vectors (CASVs) which accurately reflect the L2 cache access patterns for each interval of execution. By comparing CASVs on-the-fly, our technique accurately classifies the program execution intervals into phases. We show that the variation in, an important architectural metric, IPC has a strong correlation with the phase changes detected by our technique.

The proposed technique demonstrates its ability to detect phases with high intra-phase homogeneity. Hence, any optimization adapted and applied to a single segment of execution from one phase, will potentially apply well to the other parts of that phase. Furthermore, our technique is able to capture long stable phases, that is the major beneficiary of run-time optimizations. Grouping transitional intervals into “Phase Zero”, improves the detection accuracy by significantly reducing the number of detected phases

We show that picking the suitable similarity threshold should balance the trade-off between homogeneity (IPC CoV), and number of detected phases.

The direct capability of the proposed technique to estimate and tabulate L2 cache demand for each phase, is considered as an advantage, and makes the

technique to be distinctive. This is mainly because per-phase tabulated L2 cache demand estimation, could be exploited for reducing L2 cache miss rates, and optimizing L2 cache utilization.

For future research, the technique will be tested on SPEC CPU Benchmarks, and comparing its findings with other techniques tested on the same benchmarks. We intend to extend our technique to be able to predict the phase ID of the next interval of execution, and to use the per-phase tabulated L2 cache demand to optimize for the reoccurring phases.

References

[1] T. Sherwood and B. Calder, "Time varying behavior of programs", Technical Report UCSD-CS99-630, UC San Diego, 1999.
[2] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications", In International Conference on Parallel Architectures and Compilation Techniques, 2001.
[3] Jeremy Lau, Stefan Schoenmackers, and Brad Calder, "Transition Phase Classification and Prediction", In the 11th International Symposium on High Performance Computer Architecture, 2005.
[4] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, " Automatically characterizing large scale program behavior", In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.
[5] Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis", In 29th Annual International Symposium on Computer Architecture, 2002.
[6] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general purpose architectures", In MICRO 33rd, pp 245–257, 2000.
[7] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction", SIGPLAN Not., 39(11):165–176, 2004.
[8] Dhodapkar and J. E. Smith, "Dynamic microarchitecture adaptation via co-designed virtual machines", In International Solid State Circuits Conference, 2002.
[9] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu., " Vacuum packing: Extracting hardware-detected program phases for post-link optimization", In 35th International Symposium on Microarchitecture, 2002.
[10] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and Wen mei W. Hwu., " An architectural framework for run-time optimization", IEEE Transactions on Computers, 50(6):567–589, 2001.
[11] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, " Processor power reduction via single-ISA heterogeneous multi-core architectures", Computer Architecture Letters, 2, 2003.
[12] Jun Yao, Hajime Shimada, Yasuhiko Nakashima, Shin-ichiro Mori, and Shinji Tomita, "Program Phase Detection Based Dynamic Control Mechanisms for Pipeline Stage Unification Adoption", ISHPC 2005 AND ALPS 2006, LNCS 4759, pp. 494-507, 2005.
[13] Denning, P.J., "The working set model for program behavior", Communications of the ACM, 5/1968, Volume 11, pp. 323-333, 1968.
[14] T. Sherwood, Suleyman Sair, and Brad Calder, " Phase Tracking and Prediction", In Proceedings of the 30th International Symposium on Computer Architecture (ISCA), pp. 336-347, 2003.
[15] T. Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder, "Discover and exploiting program phases", IEEE Computer Society, 0272-1732/03, 2003.
[16] Dhodapkar and James E. Smith, "Comparing Program Phase Detection Techniques", In the 36th International Symposium on Microarchitecture, pp 217–227, 2003.
[17] David Tam, "Operating System Management of Shared Caches on Multicore Processors", Ph.D. Thesis, Department of Electrical and Computer Engineering, University of Toronto, 2010.
[18] Intel 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation, volume 3B: System Programming Guide Part 2, 2015.
[19] Stephane Eranian, "Update on perf_events", CERN workshop, 2013.
[20] Georgios Bitzes, and Andrzej Nowak, "The overhead of profiling using PMU hardware counters", Technical Report, CERN Openlab, 2014.