Unsupervised Variable-Grained Online Phase Clustering for Heterogeneous/Morphable Processors

Miguel Tairum Cruz
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa, Portugal
Email: mtairum@sips.inesc-id.pt

Pedro Tomás
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa, Portugal
Email: pedro.tomas@inesc-id.pt

Nuno Roma
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa, Portugal
Email: nuno.roma@inesc-id.pt

***Abstract*—Phase clustering and identification schemes provide useful insights to optimize the application execution, providing the means to accurately control scheduling and migration mechanisms in heterogeneous and reconfigurable architectures. However, while conventional coarse-grained classification mechanisms can be used for most straightforward optimizations, fine-grained resolutions are often required to identify their irregular behaviours. Nevertheless, typical approaches often result in an over-segmentation and in an excessive number of phases. The proposed scheme aims to solve these issues by providing a lightweight runtime variable-grain phase classification system, which can be easily embedded in general processor architectures. Moreover, by relying on fine-grain basic block identification, appropriate control mechanisms can be devised to follow the application execution and identify the different phases, resulting in a convenient segmentation that is properly matched with the application behaviour. According to the devised experimental evaluation, the proposed system achieves a better characterization than state-of-the-art coarse-grained approaches and provides the same level of detail as fine-grained schemes, resulting in a sparser and more intelligible segmentation.**

***Index Terms*—Phase Clustering and Identification, Fine-Grain Parallelism, Heterogeneous Architectures, HPC Monitoring.**

I. Introduction

When analysed at a coarse granularity, most general purpose applications tend to exhibit a regular and repetitive behaviour throughout their execution. Such observed set of recurrent and similar execution patterns are defined as phases, which are classified by their specific characteristics and performance metrics at those intervals. Execution phases can be particularly useful for application characterization and performance optimization [1], [2], [3].

In accordance, the granularity of the inferred program phases dictates the resulting application evaluation and characterization, with coarse-grain phases (in the order of millions of instructions) detecting the major functions and loops present in the code [1], [3], [4]; and fine-grain phases (in the order of hundreds or thousands of instructions) detecting more subtle and irregular variations in the code [5], [6]. Such fine-grained phase identification is particularly useful for dynamic scheduling and migration mechanisms, required for an efficient exploitation of heterogeneous and reconfigurable architectures [5], [6], [7]. These architectures use fine-grain phase classification to guide migrations mechanisms, aiming the optimization of the resulting performance and energy efficiency.

Different execution metrics can be used to identify and classify the program phases, ranging from code execution frequencies [3], to specific performance metrics that depend on the target architecture [1], [5]. In particular, the identification of basic blocks (i.e., sections of code with only one entry and exit point) is frequently used to devise offline phase identification and classification schemes that are independent of any particular hardware-based counters [3], [8]. These basic blocks are usually identified by capturing the memory address of branch instructions, along with the number of instructions that occurred in between. The collected information is used to examine the code execution frequencies (the ratios in which different regions of code are being executed over time), and subsequently used to find and classify a phase behaviour into a program phase.

However, despite the different existing techniques for phase classification, most current implementations are mainly devised for coarse-grain phase identification, bypassing subtle fine-grain phase changes that are present in general purpose applications [2]. In particular, commonly used techniques based on fixed-time intervals or sampling-based implementations are not suitable to identify such short phases, since the application phase behaviour at such a fine grained scale is much more disparate, often resulting in wrong clusterings. Moreover, when targeting thread scheduling and migration mechanisms in highly heterogeneous and reconfigurable architectures [5], [6], [7], with the purpose of improving performance and/or energy-efficiency, such approaches usually result in large and unsustainable overheads [9].

In contrast, the new phase identification structure that is herein proposed aims to provide an accurate but variable-grained phase identification scheme that offers a similar level of detail as current state-of-the-art fine-grained implementations. To attain such objective, it is complemented with a post-processing clustering step to filter spurious or misidentified phases. Such an approach is specially devised to be integrated with efficient and low-overhead thread migration schemes [10] and with accurate performance and power models [11], [12], in order to allow the development of insightful thread scheduling techniques.

To achieve such a variable-grained phase identification, the new identification structure that is herein proposed captures specific branches information at a fine-grain level and ap-

plies convenient clustering techniques to identify the program coarser loop structures for accurate classification. In particular, every time a new branch instruction occurs [3], [4], [13], the previously identified branch addresses are efficiently iterated, until a closed branch loop is found. Such an identified loop is comparable to a basic block, containing information regarding the instructions in between. Depending on the desired granularity, these loops are categorized and classified into individual program phases, guided by several performance metrics and various control mechanisms. Depending on the target resolution, the control can be appropriately tuned to filter the noise that is introduced by fine-grain irregularities, thus resulting in variable and adjustable granularity.

The supporting hardware architecture that is herein proposed integrates two fixed-sized table structures to index the branch loops and program phases. Additionally, an existing Branch Table Buffer (BTB) structure is slightly modified, in order to iterate over the branch instructions, such as to enable a conveniently devised control system to monitor the incoming loops and program phases and compare them with the ones already stored in the tables, allocating and classifying the information accordingly.

In summary, this paper presents the following contributions:

- Proposal of a new online variable-grained phase clustering architecture for heterogeneous processors. By conveniently capturing the branch instructions information, it manages to split the application into fine-grained and coarse-grained phases. Such phase information can then be usefully applied for dynamic scheduling and efficient migration mechanisms (not included in this manuscript).
- Proposal of a supporting architecture characterized by a low hardware overhead. It only requires a few new structures and minor modifications of the existing processing structure. In order to store the branch loop information and classified phases, two fixed-sized table structures are used. Additionally, elementary control mechanisms are responsible for phase identification and classification.
- Prototyping and evaluation of the proposed structure in a cycle-accurate simulator. The phase clustering results are compared to a state-of-the-art phase classification tool [4] and extrapolated against a fine-grain classification system present in a state-of-the-art heterogeneous architecture [6], [7].

The rest of this manuscript is organized as follows. Section 2 presents the state-of-the-art and related work. Section 3 describes the proposed phase clustering architecture, together with the set of supporting structures. Section 4 presents an experimental evaluation of the proposed architecture, comparing it to a state-of-the-art phase classification implementation. Section 5 addresses the future work and presents the main conclusions.

II. Related Work

The identification and classification of program phases has been the focus of several studies, with coarse-grain solutions being often used for the purpose of promoting applications optimization (e.g. to find memory or compute bound sections) [1], [3] and fine-grain systems serving as input to enhanced dynamic scheduling and switching mechanisms. In particular, such mechanisms are especially useful for heterogeneous architectures and reconfigurable/morphable structures [5], [6], [14], where the identification of program phases is often regarded as utmost importance to allow attaining an appropriate balance between performance and energy efficiency [2].

A. Coarse-Grain Clustering

Coarse-grained phase clustering systems are usually based on monitoring the program execution at fixed-size intervals to classify a given phase. Such approach was proposed by Dhodapkar et al. [8] and Sherwood et al. [3], [15], [16], who considered offline phase clustering at granularities of 100k and 10M instruction intervals, respectively, by profiling each interval with basic block information. Basic blocks are sections of code with only one entry and exit point, and are usually identified by capturing the memory address of the branch instruction, along with the number of instructions that occurred in between. In particular, Dhodapkar et al. adapted a bit vector to keep track of which basic blocks have been executing, accounting only for the number of touched blocks in each interval, and using the result for phase clustering. In turn, Sherwood et al. tracked the proportion of time that is spent executing each code block, and used such information to cluster the interval into a phase. This subtle distinction can reduce the classification noise in applications that present a number of blocks that execute only intermittently.

Lau et al. [1] focused on the same intervals of 10M instructions as [3] but, instead of relying on code execution frequencies for phase clustering (i.e., basic block frequency), they used performance metrics in their clustering (e.g., Cycles per Instruction (CPI)), resulting in an architectural specialization (and dependence). They also proposed some improvements on the phase clustering, by introducing a transition phase, i.e., a short interval between stable phases with irregular behaviour. By grouping all transition phases into a single one, it results in the identification of fewer phases and in an increase of the phase prediction accuracy. Isci et al. [17] also relied on performance monitoring counters for their phase clustering, by using the ratio between memory transactions and micro operations not only for the clustering but also for reducing the power consumption through Dynamic Voltage Frequency Scaling (DVFS).

Alternatively, Huang et al. [18] also aimed for a reduced power consumption, by replacing the previously proposed fixed-intervals sampling scheme with subroutines identified by a call stack, resulting in a phase clustering closer to the actual application's code.

Hsu et al. [13] exploited the usage of the hardware branch trace buffer, featured in some Intel processors, to sample the branch instructions through available performance counters. After capturing a branch instruction, a trace is formed by following the last four *taken* branches in the buffer, resulting

in the identification of branch loop basic blocks that will then be classified into phases.

More recently, Sembrant et al. [4] iterated over the basic block implementations, by developing the *Scarphase* software, that identifies basic blocks using only conditional branch instructions information. Given a fixed sampling interval, the conditional branch addresses (identified by performance counters available in the architecture under evaluation) are sparsely sampled and classified into a phase, similar to the basic block vectors in [3]. By only using a small test sample of the intervals, the implementation's overhead is reduced, while keeping a low relative error of the vector's execution frequency versus more dense approaches [8]. Furthermore, they also implemented a dynamic sample rate adaptation, by reducing the number of samples whenever sequential intervals are classified into the same phase, thus reducing the runtime overhead, at the cost of a small precision loss.

B. Fine-Grain Clustering

The main problem that is observed in the referred coarse-grained phase clustering implementations results from the fact that they bypass more subtle fine-grained phase changes (in the order of hundreds or few thousands of instructions) that are present in general purpose applications [2]. Although fine-grained phase identification is particularly useful for dynamic scheduling and migration mechanisms, Shelepov et al. [14] implemented a static control-flow profiler to be embedded in a scheduler, which acquires the required information for phase clustering. In particular, it trades accuracy for simplicity and efficiency, when compared to online-based implementations.

Cochran et al. [5] used both offline profiling and online classification in their implementation. Offline data profiling is used to train the performance models based on hardware counters. The acquired data was used to model the optimal settings of a modified K-means clustering algorithm. During the application runtime, a DVFS thermal model guides the control decisions according to the clustered information, resulting in a varying small phase granularity, aiming the energy and thermal optimization.

Alternatively, the heterogeneous architecture presented by Padmanabha and Lukefahr et al. [6], [7] uses only dynamic phase clustering, identifying fine-grain phases by using the concept of traces. Similar to basic blocks, traces are defined as sequences of instructions with an entry and exit point. However, they are identified by the backedge boundaries created by control instructions (branches, function calls and returns) that branch to a previous target address. Then, the resulting re-convergence points create small control-independent code blocks (traces) that can be easily used for future phase prediction. Smaller traces are also grouped into super-traces (in the order of hundreds or few thousands of instructions), increasing their size until the decision to conduct a core migration achieves a minimum energy efficiency.

While the referred coarse-grained implementations are not adequate for energy efficient heterogeneous and morphable architectures, the fine-grained implementations that have been proposed are often too detailed in their classification, leading to redundant phases or erroneous migrations, which limit the potential energy efficiency gains. A variable-grained implementation that detects phases at a fine and accurate grain and clusters them into coarser-grained phases that depend on the application's behaviour, can thus provide the same energy efficiency gains without the handicap caused by redundant phases, leading to an overall better efficiency. The architecture herein proposed aims to solve this exact problem. Furthermore, the architectures relies solely on a hardware implementation, eliminating overheads introduced by software implementations [4], [13].

III. Proposed Phase Clustering Scheme

As it was previously referred, general purpose applications often exhibit irregular and unpredictable execution patterns, interleaved with regular and repetitive code structures. To identify the several phases of a given application, sequences of instructions must be analysed and clustered, by using either conventional performance metrics or hardware independent metrics, such as committed branch instructions. This section presents the proposed phase clustering and identification scheme and its respective implementation.

A. Phase Clustering and Identification Algorithm

The proposed phase clustering and identification scheme gathers repetitive patterns of instructions by identifying loops in the executed code. Hence, whenever a conditional branch instruction occurs, the algorithm follows the preceding branches until a closed loop is found (or until a maximum number of iterations is reached), resulting in a *branch-loop* block with the respective code section information. Upon the identification of a new *branch-loop*, it is then stored or merged in a history table for future lookups (see Algorithm 1).

However, the resulting granularity of the *branch-loop* blocks tends to be too fine-grained (e.g. below a hundred instruction) to be positively exploited by reconfigurable or heterogeneous architectures, when considering the usual reconfiguration or thread migration overheads. As a result, multiple instances of the same block must be identified in order to characterize a given program phase. A program phase will thus aggregate several *branch-loop* blocks and, as such, must be updated every time a new *branch-loop* is identified (see Algorithm 1). Additionally, control mechanisms responsible for the identification of new (or already existing) *branch-loop* blocks must also be updated every time a new block is found. Specifically, a global threshold (GT) that controls the total amount of identified *branch-loop* blocks, must be relaxed every time a new block is identified (see Algorithm 1). This threshold serves the purpose of reducing the probability of an irregular or uncommon *branch-loop* block triggering a new phase, by requiring that newly identified blocks present an Identification Rate (IR) (the ratio between the number of times the block was identified and the total number of blocks) greater than the threshold.

Finally, if the same *branch-loop* block is successfully identified during the latest traces, it will trigger a new phase change. This scheme is controlled by another mechanism, where a History Threshold (HT) defines the number of past *branch-loop* blocks that have to be identified for the current block to trigger a new phase change (see Algorithm 1). If the current *branch-loop* block was identified more than a certain number of times, it passes the HT, and the current phase will then be updated and classified into a history table, creating a new phase entry or merging it with a previously identified one. Afterwards, a new phase starts its characterization, thus restarting the entire scheme.

On the other hand, to enable the development of intelligent scheduling systems and guarantee the integration with efficient performance and power models, the proposed scheme can also record a set of performance counters (e.g., number of load and store instructions, number of cache misses, number of floating-point and SIMD instructions, and total number of instructions executed). Accordingly, whenever a previously identified basic block re-appears further along the execution, the previously monitored information can be passed to the system scheduler and/or hypervisor modules. For example, these modules can then take this information into account in order to apply dynamic voltage and frequency scaling, to migrate the application/thread to a more energy-efficient or high performance core (in the context of big.LITTLE systems), or by reconfiguring/morphing the core in order to allow for an specialization of the underlying architecture.

Algorithm 1 Phase clustering and identification architecture: pseudo-algorithm.

```
1: function UPDATE_GLOBAL_THRESHOLD
2:     GT ← (fixed_value% × GT)
3: end function

4: function UPDATE_BLOCK_IDENTIFICATION_RATE()
5:     Identification Rate (IR) ← (Identified_Loop × 100)/Total_Loops
6: end function

7: if branch_instruction then
8:     #Branch_Loop Block generation
9:     repeat
10:        check_prev_branch()
11:    until loop = found ∨ iterations = max
12:    if loop ≠ found then
13:        exit()
14:    end if

15:    Lookup_History_Table()
16:    Add/Merge_Loop_Block_to_Table()

17:    Update_Block_Identification_Rate()
18:    Update_Global_Threshold()    ▷ Threshold relaxation

19:    #Threshold verification
20:    if IR > GT then
21:        if Identified Loop > Hist. Thres. (HT) then
22:            Update_Current_Phase()
23:            Add_Phase_to_History_Table()
24:        else
25:            Update_Current_Phase()
26:            Continue
27:        end if
28:    end if
29: end if
```

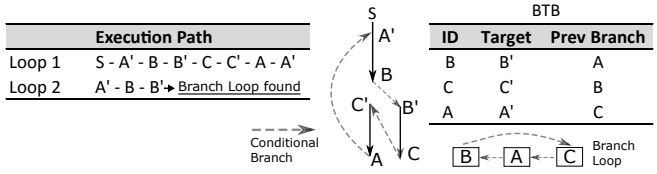

Fig. 1. Runtime definition of *branch loops*, by using a modified BTB.

B. Phase Clustering and Identification Implementation

The proposed scheme to identify *branch-loops* whenever a conditional branch occurs is depicted in Figure 1. For such purpose, the Branch Target Buffer (BTB), commonly used in processor architectures to support branch predictors, traces the branch instructions and their respective target address, to be looked up whenever a new branch appears in the code. In accordance, the proposed approach considers a simple modification of the processor's BTB, in order to include useful information regarding the branch that preceded the current one (see Figs. 1 and 2). Then, when a conditional branch occurs in the code, the BTB will be looked up, starting at the current branch entry (if existent) and following the preceding branch path, until a closed loop is found (or until a maximum number of iterations is reached). The resulting *branch-loop* will be a block containing the instructions inside the loop (usually, in the order of tens or hundreds of instructions) and their respective performance information is acquired and registered at runtime (e.g. average CPI, cache misses, etc.).

The identified blocks are subsequently stored in a fixed-sized Branch-Loop Block History Table (BLBHT) (see Fig. 2). This structure is mainly used to reduce the number of different blocks and to filter smaller behaviour variations, i.e., blocks similar to the ones already present in the history table that can be merged, by updating their performance information accordingly and increasing their number of occurrences. The merge operation is controlled by instruction and performance thresholds that compare the size, number of instructions and average CPI of the identified blocks against predetermined values.

Nevertheless, the size of the identified blocks in the BLBHT will tend to be too fine-grained, making them difficult to be efficiently used by reconfigurable or heterogeneous architectures [6]. To circumvent such an over-segmentation, the *branch-loop* blocks are subsequently used by a clustering step to classify

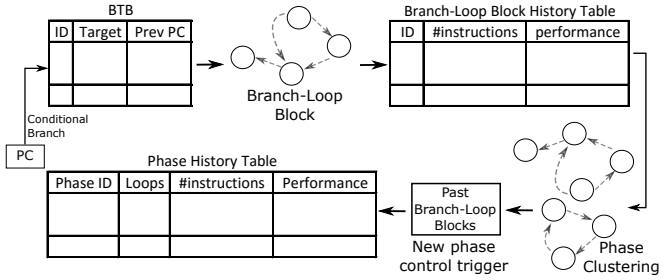

Fig. 2. Overview of the proposed phase clustering and identification architecture.

the current phase of the application (see Fig. 2). Hence, by using the information gathered for the *branch-loop* blocks, the current phase undergoes an update of its classification, as new blocks keep appearing in the code.

In accordance, when an already identified *branch-loop* block is found, a control mechanism analyses its occurrence rate against the total number of blocks that were found until that point. This control mechanism is based on a global threshold (GT) (see Algorithm 1) that is dynamically adapted throughout the execution, in order to account for the increasing number of total identified blocks. The initial value of this threshold can also be tuned, according to the target resolution. Whenever a *branch-loop* block is identified, the global threshold gets relaxed, thus accounting for the increasing total number of identified blocks. If the Identification Rate (IR) (the number of times the block was identified versus the total number of identified blocks) of such a newly identified block passes this threshold, a trigger for a new phase detection is set. The basis for this mechanism considers the fact that new *branch-loop* blocks are only recognized as new phases when they occur a minimum number of times. Furthermore, it also assumes that existing blocks have already been used to identify previous phases in the past and they present a higher chance to represent future phases.

To achieve this, a complementary control mechanism (see Algorithm 1) analyses the *branch-loop* block that triggered a new phase detection, by comparing it to a short history of the previously occurred blocks (*branch-loop* block history threshold (HT)), preventing the creation of redundant and too short phases (see Fig. 3), which often occur at this level of resolution. Such undesired sections are perceived by a repetitive behaviour consisting on several interleaved blocks with different performance metrics, suggesting the existence of multiple phases in that section. Hence, the proposed approach contrasts with the scheme that was used in [6], by grouping the instruction blocks until they satisfy a minimum threshold of instructions. In [6], sections as the one described above would be identified as multiple phases (see top of Fig. 3), while the approach that it is herein proposed identifies these sections as a single phase (see bottom of Fig. 3).

When a *branch-loop* block passes this control mechanism, the current phase is stored in a Phase History Table (PHT) (see Fig. 2) and a new phase will start. Similarly to the BLBHT,

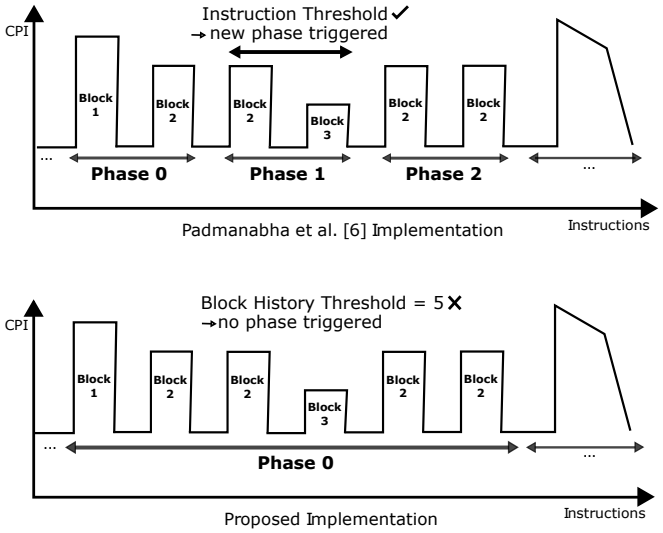

Fig. 3. Phase identification example of intermittent and interleaved small blocks. The top implementation classifies phases whenever a group of blocks surpasses the instructions threshold, resulting in the identification of several phases [6]. The proposed implementation (bottom) does not trigger a new phase since the same block has not appeared enough times during the last traces, resulting in a single coarse-grain phase.

if a phase is already present or is very similar to one in the history table, their respective information will be merged and updated.

Naturally, this fine-grained phase clustering results in varying phase lengths, closely fitted with the applications behaviour, and offering the possibility to further tune the granularity, by changing the initial values of the considered control thresholds. This contrasts with most coarse-grain classification schemes that have been proposed in the literature, based on fixed-sampled intervals.

IV. Experimental Results

To evaluate the proposed phase clustering architecture, the cycle accurate Gem5 simulator [19] was used to simulate a modified x86 processor. The modelled processor consists of an out-of-order core with an 8-issue width and a 5-stage pipeline, operating at a frequency of 2.4GHz. The memory system includes an unified 32KB L1 cache and a 256KB L2 cache.

As stated in the previous section, some additional fields were added to the original BTB of the processor, in order to include the information concerning to the previous branch. Likewise, the Branch-Loop Block History Table, the Phase History Tables and all the required control mechanisms were included in the processor architecture. In particular, for the experimental evaluation that is herein presented, the BLBH table and the PHT were configured with a dimension corresponding to 1024 *branch-loop* blocks and 128 phases, respectively. By considering such sizes, the amount of additional hardware resources introduced by the proposed architecture can be estimated. The corresponding control mechanism was not considered in this estimate, since it can mostly be realized with registers

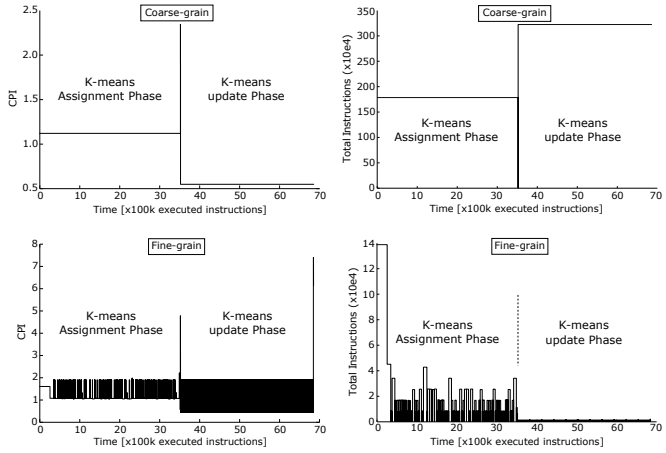

Fig. 4. Coarse- and fine-grained phase clustering and identification of the *K-means* benchmark using the proposed phase identification system. Both results were obtained by tuning the initial values of the control mechanism thresholds. The left plots represent the average CPI of the identified phases, while the right plots present the total number of instructions of the same phases during the benchmark execution.

and simple logic structures, resulting in a lesser hardware impact than the main architectural structures. The estimate only accounts for the memory necessary to accommodate the main additional hardware structures (i.e., BLBHT, PHT and the additional BTB modifications), which also include a set of additional performance counters, such as the total number of executed instructions, the average CPI and the number of floating-point instructions. In accordance, considering that the size of both the BLBHT and PHT structures influence the amount of detail that can be extracted, a more conservative value was estimated. As such, the modifications in the BTB introduce a memory cost of 128KB, while the costs of the BLBHT and PHT are estimated to be about 120KB and 8MB, respectively. These memory requisites are equivalent to those required by other tools referred in the literature, such as [4].

Furthermore, it was decided to integrate the phase clustering and identification system in the commit stage of the pipeline, in order to avoid additional branch misprediction logic. Such design option provides a comparable environment for both in-order and out-of-order cores which, when running the same application, should exhibit the same number of branch instructions and be independent of eventual miss-speculations.

A. Clustering Quality Assessment

To validate the obtained results, the application phases had to be manually correlated with their respective execution code. For such purpose, it was used a benchmark suite characterized by a phase behaviour that is easily identifiable (at least) at coarse-grained resolution, in order to understand if the main code loops are being correctly represented by the proposed phase clustering system. For such purpose, it was decided to use the *K-means* benchmark included in the Rodinia Suite [20], whose underlying algorithm partitions the several observations into k clusters, grouping them with the nearest mean cluster. It consists of two major program phases: an *assignment phase*, where each observation is assigned to its nearest cluster; and an *update phase*, where the new means are calculated for the new formed clusters.

Figure 4 represents a coarse- and fine-grained phase clustering and identification of the *K-means* benchmark algorithm, outputted by the proposed implementation. To obtain such different resolutions, the initial values of the thresholds that analyse the block occurrence frequency and the number of previously occurred blocks were conveniently tuned, by tightening those values for a coarse-grained resolution (requiring more identical blocks to trigger a new phase) and relaxing them for a fine-grained resolution (less number of identical blocks required to trigger a new phase). In the presented example, both resolutions can identify the two main program phases. However, the fine-grained clustering, with more relaxed initial threshold values, can further detect irregular behaviours present in the application, showcasing previously hidden details. Nonetheless, this clustering still presents coarse-grained phases in the order of hundreds of thousands of instructions (specially during the K-means *assignment phase*, as it is visible in the bottom right plot in Figure 4), showcasing the variable-grained resolutions offered by the proposed architecture.

B. Comparison With Scarphase

Then, the proposed phase clustering implementation was compared with the Scarphase framework [4], by considering a x86 host processor with the same frequency and similar memory sizes as those used in the Gem5 simulator. As it was previously referred, this Scarphase framework consists in a software implementation of a coarse-grained phase clustering system with dynamic branch instruction sampling. This framework was chosen for comparison due to its software implementation accessibility as well as its promising results when compared with other referred coarse-grained implementations.

Figure 5 presents the resulting comparison between the proposed implementation and Scarphase's fixed-intervals approach. This comparison aims to study the advantages of a varying-grained phase clustering scheme (proposed implementation) against a fixed-interval coarse-grained approach (Scarphase). To perform this comparison, it was used the Rodinia *LavaMD* benchmark, since it exhibits a larger number of program phases, specially at fine-grained resolutions. For both implementations, coarse- and fine-grained resolutions are presented, by changing the initial values of the previously mentioned thresholds (in the proposed implementation) and the considered interval sizes (in Scarphase implementation). As such, Figure 5 presents three results measured for the two different granularities: the top plot depicts the average CPI per identified application phase; the middle plot presents the phase chart, which allows identifying the starting position of each program phase; and the bottom plot presents the total number of instructions per identified application phase, providing an overview of the phase size.

By analysing the obtained results, it is possible to observe that Scarphase introduces a significant amount of noise (phase

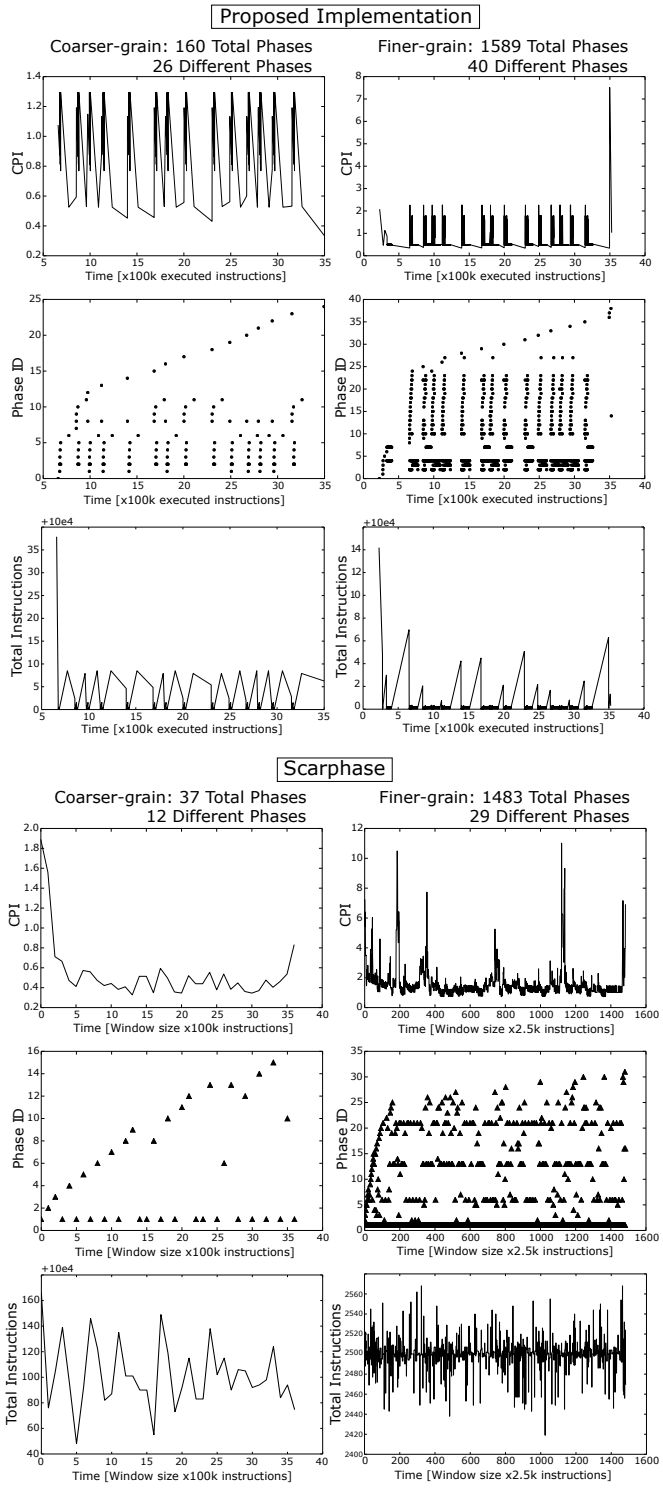

Fig. 5. Resulting phase clustering and identification provided by the proposed implementation and Scarphase's phase classification framework for the *lavaMD* benchmark. The top plot presents the average phase CPI, the middle plot presents the phases identified during the benchmark execution and the bottom plot presents the total number of instructions in each phase during the benchmark execution.

misclassification) in its fine-grained classification for smaller window sizes, resulting from the larger performance disparities

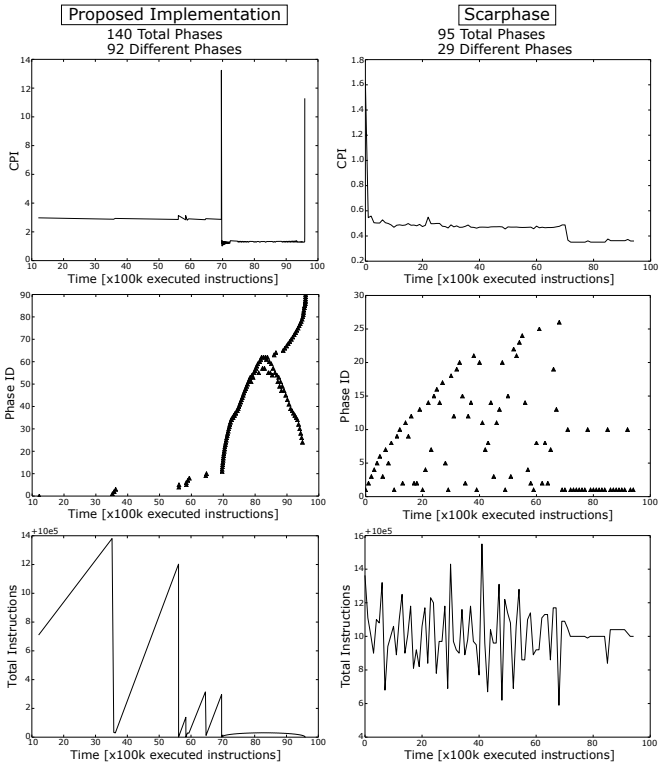

Fig. 6. Phase clustering and identification provided by the proposed implementation compared with Scarphase's phase classification for the *LUD* benchmark. The three groups of plots represent the average phase CPI, the identified phases during the benchmark execution and the total number of instructions in each phase.

in small phases (see the "CPI" plot in Figure 5). In contrast, for coarse-grained phases, Scarphase can extract relatively accurate monitoring information of the application at given points of its execution, but fails to identify the overall behaviour of the application (e.g. phase repetition along the execution time), as it can be seen by comparing the obtained results with the "CPI" and "Phase ID" plots of the proposed implementation (see Figure 5). Furthermore, the phase clustering provided by the proposed implementation not only presents a higher resolution, but it also identifies varying-grained phases, which can be best seen by the disparate number of total instructions present in each phase, ranging from a few thousands to tens of thousand instructions (see the "Total Instructions" plot in Figure 5). In turn, the Scarphase implementation presents a more stable (lower resolution) total number of instructions per phase, ranging between hundreds of thousand instructions (in the coarse-grained implementation) and between tens of thousand instructions (in the more fine-grained approach).

An additional comparison between the proposed implementation and Scarphase's proposal is presented in Figure 6, for the Rodinia's LU Decomposition algorithm. The threshold initial values of the proposed architecture were set in order to identify a similar number of phases with both implementations. In this example, the proposed phase clustering scheme identifies a total of 140 phases (with 92 different

phases) while the Scarphase implementation identifies a total of 95 phases (with 29 different phases). By observing the average CPI plots of both implementations (see Figure 6) it is possible to distinguish two different algorithm phases, with a CPI threshold after the $7M^{\text{th}}$ instruction in the algorithm execution. By analysing the identified phases throughout the algorithm's execution (see Figure 6), it is possible to observe that Scarphase identifies a higher number of phases during the first major phase of the LUD algorithm, with the proposed architecture barely identifying any phases. In fact, further analysis reveals that the *branch-loop* blocks identified during the algorithm's major phase are too irregular to trigger a phase change (there is an insufficient number of blocks in the block history threshold (HT)), contrasting with the second major phase of the algorithm, where several blocks are identified and clustered into different phases. Finally, the total number of instructions per phase depicts a higher variation in the proposed implementation (see Figure 6), while Scarphase presents phases with similar sizes, further confirming the variable-grained phase clustering and identification of the proposed architecture.

V. Conclusion

The identification of program phases is of utmost importance, not only to control the scheduling and migration mechanisms in heterogeneous and reconfigurable architectures, but also to promote the application optimization. However, typical coarse-grained procedures cluster an application into fixed-sized phases, which does not allow for an accurate identification of important behavioural patterns. On the other hand, existing fine-grained solutions are often too specialized, problem-specific and characterized by unnecessary high resolutions when considering its usage in typical thread schedulers and reconfiguration engines, undermining the resulting energy efficiency.

The proposed phase clustering and identification architecture aims to solve such issues by providing an online fine-grained phase identification mechanism with varying granularities. It is supported on a lightweight hardware structure that expands the typical BTB logic to include additional phase information. By relying on a low-level identification of the program control flow, the proposed mechanism is able to identify application basic blocks, which are subsequently grouped in program phases of variable length. It attains a convenient segmentation of the program behaviour that is properly matched with the application code and, when compared with state-of-the-art coarse-grained approaches, provides a better application characterization, while attaining similar levels of detail as fine-grain solutions.

Acknowledgment

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under projects PTDC/EEI-ELC/3152/2012 and UID/CEC/50021/2013.

References

[1] J. Lau, S. Schoenmackers, and B. Calder, "Transition Phase Classification and Prediction," in *11th International Symposium on High-Performance Computer Architecture. HPCA-11*. IEEE, 2005, pp. 278–289.

[2] H. H. Najaf-Abadi and E. Rotenberg, "Architectural Contesting," in *15th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2009, pp. 189–200.

[3] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and Exploiting Program Phases," *Micro, IEEE*, vol. 23, no. 6, pp. 84–93, 2003.

[4] A. Sembrant, D. Eklov, and E. Hagersten, "Efficient Software-Based Online Phase Classification," in *International Symposium on Workload Characterization (IISWC)*. IEEE, 2011, pp. 104–115.

[5] R. Cochran and S. Reda, "Thermal Prediction and Adaptive Control Through Workload Phase Detection," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 1, p. 7, 2013.

[6] S. Padmanabha, A. Lukefahr, and et al., "Trace Based Phase Prediction for Tightly-Coupled Heterogeneous Cores," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 445–456.

[7] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. G. Dreslinski, T. F. Wenisch, and S. Mahlke, "Exploring Fine-Grained Heterogeneity with Composite Cores," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 535–547, 2016.

[8] A. S. Dhodapkar and J. E. Smith, "Managing Multi-Configuration Hardware Via Dynamic Working Set Analysis," in *29th Annual International Symposium on Computer Architecture*. IEEE, 2002, pp. 233–244.

[9] X. Yang, S. M. Blackburn, and K. S. McKinley, "Computer Performance Microscopy With SHIM," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2015, pp. 170–184.

[10] M. Rodrigues, N. Roma, and P. Tomás, "Fast and Scalable Thread Migration for Multi-core Architectures," in *13th International Conference on Embedded and Ubiquitous Computing (EUC)*. IEEE, 2015, pp. 9–16.

[11] R. W. Moore, B. R. Childers, and J. Xue, "Performance Modeling of Multithreaded Programs for Mobile Asymmetric Chip Multiprocessors," in *7th International Symposium on High Performance Computing and Communications (HPCC)*, 2015, pp. 957–963.

[12] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-Cores Through Performance Impact Estimation (PIE)," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 213–224.

[13] W.-c. Hsu, H. Chen, P.-c. Yew, and D.-y. Chen, "Phase Locality Detection Using a Branch Trace Buffer for Efficient Profiling in Dynamic Optimization," Intel, Tech. Rep., 2002.

[14] D. Shelepov, J. C. Saez Alcaide, and et al., "HASS: a Scheduler for Heterogeneous Multicore Systems," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 66–75, 2009.

[15] 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*. IEEE, 2001, pp. 3–14.

[16] T. Sherwood, S. Sair, and B. Calder, "Phase Tracking and Prediction," in *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2. ACM, 2003, pp. 336–349.

[17] C. Isci, G. Contreras, and M. Martonosi, "Live, Runtime Phase Monitoring and Prediction on Real Systems With Application to Dynamic Power Management," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2006, pp. 359–370.

[18] M. C. Huang, J. Renau, and J. Torrellas, "Positional Adaptation of Processors: Application to Energy Reduction," in *30th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2003, pp. 157–168.

[19] N. Binkert, B. Beckmann, and et al., "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[20] S. Che, M. Boyer, and et al., "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.