A Survey of Phase Classification Techniques for Characterizing Variable Application Behavior

Keeley Criswell and Tosiron Adegbija, *Member, IEEE*

Abstract—Adaptable computing is an increasingly important paradigm that specializes system resources to variable application requirements, environmental conditions, or user requirements. Adapting computing resources to variable application requirements (or application phases) is otherwise known as phase-based optimization. Phase-based optimization takes advantage of application phases, or execution intervals of an application that behave similarly, to enable effective and beneficial adaptability. In order for phase-based optimization to be effective, the phases must first be classified to determine when application phases begin and end, and ensure that system resources are accurately specialized. In this paper, we present a survey of phase classification techniques that have been proposed to exploit the advantages of adaptable computing through phase-based optimization. We focus on recent techniques and classify these techniques with respect to several factors in order to highlight their similarities and differences. We divide the techniques by their major defining characteristics—online/offline and serial/parallel. In addition, we discuss other characteristics such as prediction and detection techniques, the characteristics used for prediction, interval type, etc. We also identify gaps in the state-of-the-art and discuss future research directions to enable and fully exploit the benefits of adaptable computing.

Index Terms—Phase classification; adaptable computing; workload characterization; variable program behavior; dynamic optimization; edge computing; multithreaded applications; big data; emerging applications

1 Introduction

Much prior research has shown that applications have variable resource requirements throughout execution. Thus, for optimal execution, system resources (e.g., memory resources, clock frequency, functional units, etc.) must adapt to changes in application resource requirements. To enable this adaptability, *application phases* [31] [94] specify execution intervals—typically measured by number of instructions executed or time periods—that exhibit similar execution behavior. Since a phase typically exhibits stable execution characteristics, the resource requirements for that phase are also stable [9] [43] [59] [95] [101]. *Phase classification* groups intervals with similar characteristics to form phases, and represents a vital first step in adaptable computing [48] [57] [58] [75] [86] [93] [101] [108]. Phase classification offers several benefits for adaptable computing in the form of effective configuration of system components (at design time or during runtime), scheduling in heterogeneous systems, design-time rapid system evaluation and simulation, etc.

Fig. 1 illustrates phase classification in the context of an application's execution. Rather than using a single configuration of system resources throughout the application's execution, phase classification determines the different application phases, such that each phase can be executed using the system configuration that most closely meets the phase's resource requirements. Prior work has shown that adapting system resources to application phases—*phase-based optimization*—enables much higher optimization potential than application-based optimization [2] [43] [48] [59] [60] [84] [92] [93] [96] [97]. Phase-based optimization evaluates an application's characteristics and determines the best system configurations that meet each phase's execution characteristics. For example, Gordon-Ross et al. [43] found that phase-based cache optimization could yield 37% and 20% improvements in performance and energy usage, respectively, compared to application-based execution.

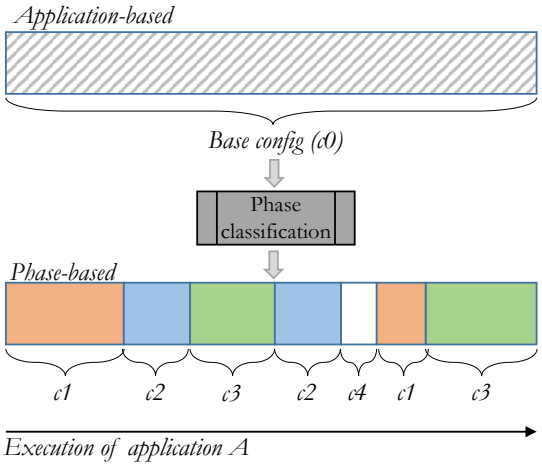

Fig. 1: Illustration of phase-based execution

Similarly, several other optimizations depend on phase classification [32] [59] [60] [93] [75] [57] [58] [86] [101]. For example, Khan et al. [57] used phase classification to facilitate thread-to-core assignment in asymmetric multicore systems, in order to optimize throughput, power, or performance per watt. Dhodapkar et al. [32] used adaptable instruction caches to meet each phase's instruction execution needs for optimized performance efficiency.

The authors are with the Department of Electrical and Computer Engineering, University of Arizona, USA, e-mail: {kcriswell, tosiron}@email.arizona.edu.

The work was done while Keeley Criswell was with the University of Arizona.

Other optimizations, such as just-in-time compilers [6] [7], dynamic instruction optimizations (e.g., [8] [70]), and performance bottleneck analysis [42] also rely on accurate phase classification. Furthermore, phase classification enables rapid system evaluation and shorter simulation time during research. Rather than completely simulating an application, only a few phases that represent the full application's execution characteristics are simulated [95] [96].

Since phase classification is so important in the analysis and optimization of modern computing systems and applications, this paper focuses on surveying recent advances in phase classification. Several phase classification techniques have been proposed that cater to different system scenarios and tradeoffs. Phase classification techniques are usually categorized by the application characteristics used for classifying the phases. For example, Sherwood et al. [95] found that an application's behavior is directly linked to the application code. The authors used the frequency of basic block execution—called *basic block vectors (BBVs)*—to classify an application's phases. A basic block is a section of code with one entry and one exit that is executed from start to finish.

Dhodapkar et al. [33] compared three phase classification techniques that use instruction working sets [32], basic block vectors (BBVs) [96] [97], and conditional branch counts [9]. The authors found that the three techniques agree on phases 85% of the time; BBVs, however, were the most effective at detecting performance changes, finding stable phases, and providing higher sensitivity, i.e., higher probability that the phases are accurately predicted [45]. The authors also found that classifying phases using instruction working sets identified phases that were 30% longer than using BBVs. Gu et al. [46] presented an overview of various phase classification techniques and metrics, such as the minimum possible size of detected phases, online/offline phase classification, and types of application characteristics used for the classification.

In this paper, we discuss recent phase classification techniques, focusing on the advances since the techniques discussed in [33] and [46]. We survey the phase classification techniques with respect to the characteristics they analyze, the types of applications for which the techniques are effective, when phases are classified, etc. We specifically emphasize discussions of new phase classification techniques for multithreaded applications, which have become more mainstream in the past few years. Finally, we explore the gaps in the state-of-the-art, and propose future research directions for addressing those gaps.

The remainder of the paper is organized as follows: in Section 2, we consider some important uses of phase classification. In Section 3, we present a brief overview of phase classification techniques presented prior to 2006. In Section 4 we present a taxonomy of phase classification, and discuss some defining characteristics of phase classification techniques in Section 5. We discuss online and offline phase classifications techniques in Sections 6 and 7, respectively. Within each of these sections, we separate the techniques into those developed for serial and parallel applications. Even though phase classification has been widely studied, there are still some major gaps in the existent techniques, with respect to multithreaded applications, big data applications, Internet of Things computing paradigms, etc. Thus, in Section 8, we discuss some of these current gaps and future research directions to address them.

2 Motivations for Phase Classification

Several optimization techniques rely on accurate phase classification to detect changes in application characteristics. Much research has shown that leveraging phase characteristics enables a finer grained optimization potential by specializing the system configurations to different execution phases. To motivate this survey, we briefly discuss some of the optimizations that use phase classification, including adaptable hardware, thread-to-core assignment, sampled simulations, and hotspot temperature analysis.

2.1 Hardware Resource Adaptability

Phase-based optimizations use adaptable hardware, such as the adaptable cache presented by Zhang et al. [107], to specialize hardware to phases' resource requirements without incurring performance overhead [32] [53] [97] [23] [43]. Adaptable hardware allows specialized hardware configurations (e.g., cache associativity, capacity, and line size [107]; issue width [38]; processor voltage and frequency [55] [83]; instruction windows [20]; and global history register length [56]) for different application phases. Phase-based optimization techniques use phase classification to determine the best points at which hardware configurations must be changed in order to best satisfy application needs.

Adegbija et al. [2] presented a technique that used adaptable caches and phase-based cache optimization to achieve an average energy delay product (EDP) savings of 26%. The authors developed a phase distance mapping (PDM) approach that mapped the difference in characteristics between a new phase and a *base phase* to the configuration space, in order to determine the new phase's best configuration. Key to the effectiveness of PDM for EDP optimization was the phase classification stage. Similarly, Meng et al. [69] used an adaptable cache to reduce processor power consumption. The authors implemented a power manager that monitored the processor's overall power usage to dictate configuration changes.

Although cache optimization is a common focus of phase-based optimization techniques, other adaptable components benefit from phase-based optimization. For example, Dynamic Voltage and Frequency Scaling (DVFS) [55] [83] [104] [10] [17] is commonly used to adapt the clock frequency/voltage to variable runtime execution needs. Other components/configurations that benefit from phase-based optimization include the issue queue [79, 19], reorder buffer [1, 35], pipeline [37, 105], register files [1, 35], etc.

2.2 Thread-to-Core Assignment

Thread-to-core assignments offer another means of optimization. Given a heterogeneous-core system, the effectiveness of specialization using heterogeneity is predicated on the scheduling of execution threads to the core that most closely satisfies the thread's execution requirement. By assigning application threads/tasks to different cores based

on phases, fine-grained energy or temperature optimization can be achieved [86] [57] [101].

In multicore systems, phase changes indicate when the ideal thread-to-core assignments might change. Phase-based thread-to-core assignment policies consider each phase's resource requirements and migrate threads to the cores that most closely match those requirements. For example, Kumar et al. [59] [60] used dynamic phase mapping on heterogeneous multicore systems to reduce the overall energy consumption of a system by 39% on average and achieved a maximum of 31% speedup over a static policy. Their techniques ensured that the threads stayed mapped to the optimal cores throughout application execution despite changing execution requirements.

2.3 Rapid Design Evaluation and Sampled Simulation

During system design (e.g., in computer architecture), the design must be extensively evaluated to determine the system's functionalities and efficiency. Evaluation is typically initially performed through simulations. In several cases, however, simulating entire benchmarks is unfeasible due to prohibitive simulation times, especially when using cycle-accurate simulators. For example, SimpleScalar [18], a commonly cycle-accurate simulator, is capable of executing 400 million instructions per hour. SPEC benchmarks [65], many of which execute well over 300 billion instructions would take about a month to run using Simplescalar (or other similar simulators, such as GEM5 [13]).

Since application phases are typically repetitive throughout the application's execution [95] [96], each distinct phase only needs to be simulated once to estimate the application's overall behavior. Thus, *phase sampling*, predicated by accurate phase classification, can be used to substantially reduce simulation time as compared to running full applications. Sherwood et al. [96] used random linear projection followed by k-means clustering [66] to group phases with similar characteristics. They found that they were able to accurately represent the entire application's execution by simulating a single section of each cluster. Their phase classification technique, called SimPoint [96] [49], is commonly used for sampled simulation [74] [76] [5] [50] [59] [60] [63] [62] [69] [72].

2.4 Hotspot Temperature Analysis

An accurate estimate of the highest possible hotspot temperature on a chip can help circuit designers to reliably verify new circuits and accurately estimate lifetime degradation of chips before the circuits reach market [15]. Srinivasan et al [99] used phases to test system limits. By rearranging application phases, they found that they were able to estimate worst-case hotspot temperatures.

3 Pre-2006 Phase Classification

To provide a background for the more recent phase classification techniques, this section presents an overview of the popular phase classification techniques presented before 2006. Many modern techniques are based on these older techniques or use these older techniques as a baseline for testing modern techniques [76] [75] [93] [84] [57] [58] [89] [108] [16]. We direct the reader to [33] for a survey that details these pre-2006 techniques.

3.1 Basic Block Vectors

Sherwood et al. [96] [97] presented a phase classification technique that uses basic blocks as a microarchitecture independent way to identify phase changes. To implement this classification technique, the frequency information for each basic block—a block of code with one entry and one exit—is stored in Basic Block Vectors (BBVs). The BBVs of different instruction intervals are then compared (e.g., using Euclidean distance) to determine the similarity between the different intervals. Similar intervals are thereafter clustered to form phases. Dhodapkar et al. [33] determined that BBV techniques provide more stable phases and higher sensitivity than other phase-classification techniques. However, BBVs have since been shown to be inaccurate when applications have a large amount of last level cache (LLC) misses [4] [21] [54].

3.2 Instruction Working Sets

Dhodapkar et al. [32] presented a phase classification technique that utilizes the Instruction Working Set for characterizing the different phases. Instruction working sets are application instructions that are executed at least once during application execution. The authors used working set signatures, a lossy-compressed working set representation. In a working set signature, a hash function is used to map working set elements into vectors. The authors recorded working set signature vectors every 100,000 instructions. The proposed technique featured a user-defined threshold value to determine the sensitivity of phase classification. A larger threshold value means that a greater change in application characteristics must occur for a phase change to be detected. As such, there will be fewer phase changes detected, and phase-based optimizations will occur less frequently.

The authors compared the Manhattan distance, or the sum of differences of each element, between the working set signature vectors after every interval. If the Manhattan distance was below the threshold value, the intervals were considered to be similar and belonging to the same phase. Otherwise, when two intervals' working set signatures differed beyond the threshold, the intervals were considered to belong to different phases for which the system resources must be specialized to optimize execution.

3.3 Conditional Branch Counts

Balasubramonian et al. [9] proposed using conditional branches to determine phase changes. They determined phases by the number of conditional branches executed over an interval. A significant change in the number of branches executed between two intervals indicates a phase change. Unlike most phase classification techniques, the authors did not use a fixed threshold value to determine a significant difference between phases. Rather, the threshold value varies dynamically as the application executes. The conditional branch counter technique and the BBV technique detect a similar percentage of phase changes—phase changes detected vs. total phase changes. However, Dhodapkar et al. [33] found the conditional branch counter technique to be less effective at detecting major phase changes, i.e., phase changes during which application characteristics change significantly.

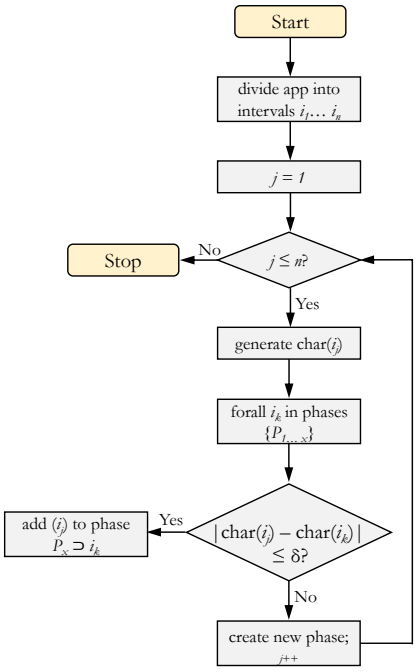

Fig. 2: High level overview of the phase classification process

4 Taxonomy of Phase Classification

Fig. 3 depicts our taxonomy of phase classification techniques. Given the the state-of-the-art in phase classification, our taxonomy is based on several factors, including classification parameters, number of threads handled, the target objective functions, when the classification occurs, and the granularity of classification. The taxonomy also includes existent challenges that still remain to be addressed. These different factors and challenges are described in the rest of the paper.

5 Characteristics of Phase Classification

Most phase classification techniques follow the same general procedure, depicted in Figure 2. First, the application is divided into instruction [54] [76] [93] [75] [58] [89] [86] [108] [101] or time [91] [99] [100] [41] intervals. For each interval i, the application characteristics (*char(i)*) are generated, for example, via design-time simulations or at runtime using statistics from hardware performance counters [22] [40] [47] [48] [75] [76] [86] [91]. Application characteristics that can be used include basic block vectors [94, 95], memory accesses [54] [93] [48], power consumption [99], stalls [101], etc.

The different intervals' application characteristics are then analyzed to determine intervals that differ by less than or equal to a threshold δ. Although the analysis techniques vary for different phase classification techniques, a sample technique is to use the Manhattan distance between two intervals' application characteristics [96] [24] [58] [75] [84] [89] [16] [100] [108]. Intervals with similar characteristics are then clustered to form phases. When new intervals are encountered, the intervals are added to existent phases or used to form a new phase, depending on the disparity between the new interval's and previously encountered intervals' characteristics.

Different factors and characteristics come into play when choosing a phase classification technique, or when designing a new one. As such, it is important for users to understand the different possible phase classification characteristics, and how these characteristics may affect phase classification accuracy. In this section, we discuss three defining phase classification characteristics: *intervals*, *classification metrics*, and *phase detection vs. prediction*.

5.1 Intervals

An interval is the length of time or number of instructions that forms the granularity with which the application characteristics are profiled. Intervals that display similar characteristics are clustered together to form phases. Intervals vary in length depending on the application and the phase classification technique. The granularity, or interval length, of a phase classification technique is most commonly measured in number of instructions.

One way to categorize intervals is with respect to their granularity. *Fine-grained intervals* are shorter (e.g., on the order of hundreds or thousands of instructions) and require that application characteristics be recorded more frequently. As such, application characteristics represent smaller sections of code, and smaller phases can be detected more accurately [16]. However, this enhanced detection comes at the cost of increased overhead—phase classification time and storage space for each interval's information [53] [67]. *Coarse-grained intervals*, on the other hand, are longer and enable faster phase classification, since fewer intervals need to be compared. However, coarse-grained intervals may obscure smaller phases, making them more difficult to detect [16]. Therefore, the interval granularity must be chosen to balance both speed and classification accuracy.

Intervals can alternatively be categorized in terms of the variability of their lengths as *fixed length* or *variable length* intervals. Fixed length intervals [54] [99] [58] [101] [16] [58] [91] [86] [93] [100], as the name implies, remain static throughout phase classification. As a result, they are simpler to implement and are more commonly used. However, because the interval size does not change during phase classification, fixed-length intervals can lead to inaccurate phase classification if the phase behavior changes at a different periodicity than the intervals. Lau et al. [61] found that fixed-length intervals can become mismatched with the naturally-periodic behavior of applications. This mismatch results in phase changes occurring in the middle of an interval instead of near the edge, thus reducing the accuracy of phase classification.

Variable length intervals [75] [48] [25] [108], on the other hand, allow the intervals to closely match the periodicity of the different phases and enable more accurate phase classification. However, variable length intervals are more complex, since they may require more detailed application analysis at design time or an online algorithm with a feedback loop for runtime phase classification [44].

5.2 Classification Metrics

Classification metrics refer to the application characteristics that are used to determine the similarity or variability between intervals. Classification metrics

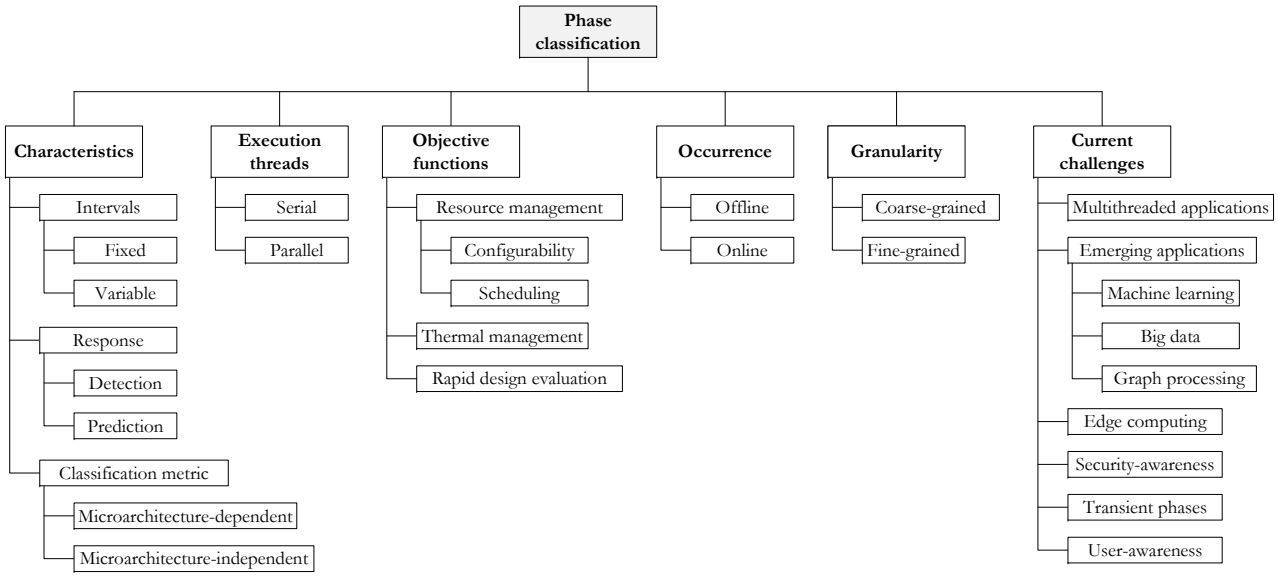

Fig. 3: Taxonomy of phase classification

can be broadly categorized into two: *microarchitecture-dependent* and *microarchitecture-independent* metrics [77, 78]. Microarchitecture-dependent metrics, as the name implies, are application characteristics that are a function of the system microarchitecture on which the application is run. These metrics are typically obtained using hardware performance counters or simulators, and would change if the application is run on a different microarchitecture or configuration. Examples of microarchitecture-dependent metrics include cache miss rates, instructions per cycle (IPC), translation look-aside buffer (TLB) miss rates, branch misprediction rates, etc. These characteristics depend on specific hardware structures (e.g., cache, branch predictor) and would change if different architectures or configurations are used.

Microarchitecture-dependent metrics are straightforward to obtain, given that simulators and hardware performance counters are ubiquitous in modern-day computer systems. In addition, microarchitecture-dependent metrics make phase classification techniques that use them more amenable to being used during runtime. Furthermore, these metrics may better detect dynamic changes in application behavior, such as new data inputs or a previously unknown stimuli. However, a major downside of these metrics is that they can hide underlying inherent program behavior, potentially leading to inaccurate phase classification [52]. It is possible for two applications to exhibit similar microarchitecture-dependent characteristics on one system but behave drastically different on another. To prevent this downside, *microarchitecture-independent* metrics [52, 36, 80] use characteristics that depict the inherent characteristics of the applications being characterized. Examples of microarchitecture-independent metrics include basic block vectors, memory access patterns, working set size, branch behavior, etc. These metrics typically need to be collected using binary instrumentation tools, such as ATOM, Pin, Valgrind, and are easier to collect at design time [85, 73, 103].

5.3 Detection vs. Prediction

Phase classification techniques can be divided into two types—*detection* and *prediction*—depending on when the actual classification occurs. Detection techniques are reactive; they compare application execution metrics after they are recorded—i.e. after the interval has ended—to those of the previous interval [22] [58] [99] [100] [41] [24] [86] [14] [25] [101] [16] [71] [12]. Since the comparison happens after an interval, phase classification occurs after a phase change has occurred [46].

Conversely, the goal of a prediction technique is to classify a new phase *before* the new phase change occurs [93] [47] [89] [88] [26] [29] [28]. These techniques feature a training period, which occurs at design time or during the beginning of application execution. During the training period, information (e.g., execution statistics) about the application's variable phase patterns are gathered. After the training period, a phase prediction technique analyzes newly acquired application characteristics and information about the application's typical phase patterns. These characteristics and analyses are then used to predict phase changes. While the analyses used vary between prediction techniques, the general idea is that predictions can be made about when and how an application's execution characteristics will change based on previously recognized and detected patterns. The biggest drawback to phase prediction techniques is that they are most effective with applications that have predictable phases or applications whose phase structure does not differ with new inputs [93]. Since detection techniques are significantly more common than prediction techniques, this survey focuses on detection-based phase classification techniques.

6 Offline Phase Classification

Offline phase classification techniques use execution characteristics, such as basic blocks and execution traces, which

TABLE 1: Overview of offline phase classification techniques

	Serial	Interval type	Variable [48] [93]; fixed [108] [54] [99] [91] [14]
		Interval width	10M [108] [14] and 1M instructions [54]; 666.6 us [99]; 10ms [91]; context switch points [48]; reuse distance [93]
		Classification metrics	Basic blocks [108] [84] [91]; cache misses [54] [48] [91]; power consumption [99]; memory accesses [93]; instructions per cycle [91]; static/dynamic instruction ratios [14]
		Analysis method	Manhattan distance [108] [14]; basic block execution frequency [84] [91]; digital signal processing [54] [99] [93]; pattern analysis [48]; principal component analysis [52]; genetic algorithm [52]
	Parallel	Interval type	Variable [75] [40]
		Interval width	Several billion instructions [75]
		Classification metrics	Instruction execution [75]; function execution [40]; instructions per cycle [22]
		Analysis method	Manhattan distance [75]; pattern analysis [40]; digital signal processing [22]

are gathered during compile time or through a priori application analysis. These techniques are generally easier to develop than online phase classification techniques, since online techniques have stringent constraints where negative impacts on application execution time must be minimized. In addition, offline techniques need fewer runtime resources for storing application characteristics during classification. Only information about phase IDs and phase change locations need to be stored during runtime.

Table 1 summarizes the surveyed offline phase classification techniques, which we categorized as *serial*—for classifying single-threaded applications—and *parallel*—for classifying multi-threaded applications. This section describes different serial and parallel phase classification techniques.

6.1 Serial Offline Phase Classification

Several techniques have been developed to classify phases in single execution threads. These techniques are typically simple to design, since they must only record and analyze application characteristics for one application thread, without needing to coordinate phase information among multiple threads.

Zhang et al. [108] focused on improving the accuracy of basic block vector techniques for single-threaded applications. The authors examined the execution sequence of fine-grained phases and found that these phases' patterns could be used to predict course-grained phase execution. The authors presented multilevel phase analysis, a technique that combines analyses of different phase granularities. During a training period, their technique identified both fine-grained and coarse-grained phases, and stored a list detailing the sequence of fine-grained phases in each new coarse-grained phase. The authors classified fine-grained phases by comparing the Manhattan distance between basic block vectors. They used an interval size of 10 million instructions while determining fine-grained phases and used outermost loop boundaries to determine coarse-grained phases. After the training period, the classification technique identified x fine-grained phases—the authors found 5 to be sufficient in their experiments. The technique then compared the execution sequence to that of previously-identified course-grained phases. If the technique discovered the same sequence of course-grained phases in the look-up table, it would accurately predict the rest of the fine-grained intervals in the course-grained phase. If the fine-grained phase sequence did not match any sequence in the table, execution continued until a course-grained phase was identified.

Ratanaworabhan et al. [84] also used basic blocks to classify single-threaded application phases. The authors used ATOM [102], an application analysis tool, to identify basic blocks and assign each basic block a unique identification number. During phase classification, the authors identified a phase change when a significant number of new basic blocks executed in a short period of time. They then created a phase identification signature by storing the basic block identification numbers of the new basic blocks that indicated the phase change. By comparing this signature to future sequences of basic blocks, they were able to identify any repetitions of a phase.

Huffmire et al. [54] presented a wavelet-based technique that used a fixed-length interval of 1 million instructions and a digital signal processing technique to classify phases in single-threaded applications. Wavelets, commonly used in digital signal and image processing, are mathematical functions that encode both frequency and spatial information, unlike BBVs, which store only frequency data [54] [95]. For their phase classification technique, the authors used wavelets to store L1 cache access data. They began by gathering an application trace of memory accesses, and then used k-means clustering to analyze the wavelets. Specifically, the authors were interested in analyzing wavelet signatures of all L1 cache accesses to predict L2 cache misses. By comparing these predictions across subsequent intervals, they were able to detect phases.

Srinivasan et al. [99] also proposed an offline phase classification technique that used fixed-length intervals. However, instead of tracking their intervals by number of instructions like most other techniques, they used an execution time—666.6 us. They also used the simulated power traces within the different intervals to determine phases, based on the observation that spatial power dissipation remains unchanged within a phase. From the power trace, they created a power vector that stores a moving average of 500 power values. They detected the beginning of a phase if three consecutive power vectors were similar, and detected the end of a phase if two consecutive power vectors were dissimilar. Conversely, most other offline phase classification techniques use simulated hardware counters or other frequency-based metrics, such as cache miss rates [54] [48] [91] [47] and basic block execution frequency [108] [84] [91]

for characterizing application phases.

Traditional interval lengths—on the order of several thousands to millions of instructions or cycles—are not always acceptable for accurate phase classification. Bui et al. [16] studied past interval lengths and found that traditionally-sized intervals do not always detect a significant number of application phases. This behavior occurs because the interval sizes are too coarse to detect small phases, and phase based optimization opportunities are missed. The authors found that classifying smaller phases can improve the optimization gained by phase-based optimization techniques. They argued that modern hardware optimizations offset much of the overhead from using fine-grained phases and proposed using 'super fine-grained intervals' to better classify phases in single threaded applications. The 'super fine-grained intervals' they suggested are on the order of tens, hundreds, or thousands of application cycles.

Although there are some phase classification techniques that use factors other than hardware data to determine phase changes—such as Srinivasan et al.'s [99] use of power consumption—many phase classification techniques use some combination of hardware data. Gu et al. [48] presented one such technique of hardware data use in applications run by a Java Virtual Machine. To classify phase changes, they analyzed a trace of the number of L1 instruction cache misses. Using pattern analysis, they looked for changes in the density of L1 instruction misses across intervals. Gu et al. aimed to classify variable and course-grained phases.

Shen et al. [93] presented another phase classification technique that utilizes hardware counters—specifically, they used a memory access trace and wavelet analysis to identify phase changes during a training period. During the training period, the authors executed several different iterations of an application, using different inputs each time. They used a digital signal processing technique—a Discrete Wavelet Transform—to analyze a memory access trace and find significant changes in the access pattern. Their technique was designed for applications with large, well-defined phases whose phase sequences do not differ significantly with different inputs. More specifically, ideal applications differ only in the number of times sections of the phase sequence are repeated, referred to as "exponents." A well-defined phase is one that exhibits a significant change in performance characteristics from that of the code surrounding it. To predict phases, the authors analyzed new application inputs and phase pattern exponents. However, their technique does not work if many phase pattern exponents did not change in conjunction with an input parameter.

Sometimes, only a limited knowledge of the executing application exists—the phase classification mechanism may not have access to characteristics such as hardware counters. Shen et al. [91] proposed *active profiling*, a phase classification technique that works on applications in binary form and requires no knowledge of loop or function structure, as in several offline phase classification techniques. The authors found that they could detect phases by controlling the application input. By controlling the input—specifically, by issuing a series of nearly identical requests—they were able to get the application to output an artificially regular pattern, or a particular behavior that repeats a prearranged number of times. To execute phase classification, the authors acquired a basic block trace of an application using the repeated request input. In their active profiling technique, the authors selected basic blocks that executed exactly the same number of times as the input was repeated. Next, they verified that the execution of all the basic blocks of one type were evenly spaced during execution. The authors ignored basic blocks with executions that were not evenly spaced, and removed false positives by running the application with a real input. Basic blocks that were not executed the same number of times as there are input requests were identified as false positives. Of the remaining basic blocks, the first instance of each indicated a phase change.

Thus far, we have discussed phase classification techniques that were designed to work on hardware-only processors or virtual machines. However, hardware/software (HW/SW) co-designed processors behave differently. Brankovic et al. [14] found that traditional phase-classification techniques are unsuited for HW/SW co-designed processors. The authors analyzed BBVs for use as a phase classification metric and found that each BBV's execution time varies significantly more when run on HW/SW co-designed processors versus hardware-only processors. Therefore, the authors theorized that one cannot assume that phases of applications classified with BBVs will behave as expected when run on a HW/SW co-designed processor. For HW/SW co-designed processors, the authors proposed Transparent Optimization Layer Description Vector Phase Classification (TDV). The Transparent Optimization Layer is the software layer in a HW/SW co-designed processor that dynamically analyzes, profiles, translates, and optimizes instructions from a guest instruction set architecture (ISA) to the host ISA. TDV uses intervals and stores 19 execution statistics that contain information about the static/dynamic instruction ratio and the instructions mix of the application to be executed. The technique then compares the statistics for each interval with the statistics from previous intervals to determine the different phases.

6.2 Parallel Offline Phase Classification

Parallel phase classification techniques cater to multithreaded or multiprogrammed applications by determining the application phases while taking into account the individual application threads. In general, due to multithreaded applications' intrinsic characteristics, parallel techniques are more complex than serial techniques. Since executing threads may be competing for the same resources, parallel thread execution can affect commonly used performance metrics such as cycles per instruction (CPI) and cache misses, rendering such performance metrics ineffective for phase classification [75].

In addition, parallel applications can have data-sharing or resource-sharing threads, which complicate phase classification [98] [51]. Although serial phase-classification techniques are simpler, multithreaded applications are becoming increasingly common in emerging computer systems. Thus, phase classification in multithreaded applications remains an important research area that necessitates novel techniques.

Peleg et al. [75] addressed multithreaded phase classification by using changes in code execution frequency to classify phases. Since traditionally used hardware counters such

as cache misses and CPI can be influenced by all threads in a multithreaded application, the authors used a performance metric that would not be influenced by other threads. The proposed technique sampled instruction addresses to capture the profile of the code execution frequency. The authors then created a histogram of sampled addresses to represent the applications' BBV.

Furlinger et al. [40] also used code execution frequency to classify phases. However, the authors recorded connections between functions and manually determined phase start and end locations. They used ompP [39], a profiling tool, to record a call graph—the connections between different functions—of the executing application. For accurate phase classification, the authors modified ompP to track all predecessor nodes in the call graph, and the number of times each predecessor node was executed. The authors defined a predecessor node as a parent node or a sibling node of the same level as the current node. They then manually analyzed the call graph to detect patterns in the execution. Their phase classification technique required the designer to determine which nodes classify as a pattern, and thus could not be automated.

Casas et at. [22] used a signal processing technique—a discrete wavelet transform—to classify phases in multithreaded applications. They found that, by using a signal processing technique on data from a trace file, they could detect only the most relevant frequencies of an applications execution—the frequencies most strongly related to the loops of an applications source code. By using a discrete wavelet transform, they were able to acquire information about the frequency and the locations of the signal. They found several different signals to be suitable for phase classification—the number of processes computing, the total duration of computing bursts, the number of point-to-point MPI calls, the number of collective MPI calls, the instructions per cylce, and the number of outstanding messages at a given point.

7 Online Phase Classification

Although offline phase classification has fewer design constraints than online classification, it is often impractical. Offline techniques require a priori knowledge of the executing applications, and many systems, such as general purpose systems (e.g., smartphones and tablets), may not have complete design time application knowledge. In addition, runtime application variability, such as new data inputs, may cause the application to behave differently. Online phase classification addresses these drawbacks by classifying the phases during runtime. However, runtime applications must execute quickly to avoid adversely impacting the executing application's latency. In addition, online phase classification techniques must be reliable and accurately classify phases with minimal overheads (e.g., energy, storage, etc.). Table 2 summarizes the serial and parallel online phase classification techniques described in this section.

7.1 Serial Online Phase Classification

Hardware characteristics are often used to classify phases in online techniques. Unlike microarchitecture-independent characteristics gathered from traces in offline techniques, online techniques gather hardware characteristics directly from system hardware counters during execution. Srinivasan et al. [101] used hardware characteristics to track the frequency of resource bottlenecks experienced by applications. Specifically, they recorded cache stalls, branch mispredicts, IPC, and resource stalls—the number of cycles the instruction dispatch is stalled due to a blocked instruction queue, re-order buffer, or issue width. They stored this information in a vector referred to as a Bottleneck Type Vector (BTV). As with BBVs, the Manhattan distance between BTVs were compared to determine which intervals behaved similarly to form phases.

Chesta et al. [24] used hardware counters, network bits sent/received, and disk read/write counts to classify phase changes. The authors used hardware counters to record the number of retired instructions, last level cache references and misses, and branch instructions and misses. The proposed technique utilized fixed-length intervals with a sampling frequency of one second. System characteristics were stored in execution vectors and phase changes were determined using the Manhattan distance between two execution vectors. When the distance exceeded a predefined threshold, the phase was considered to have changed.

Gu et al. [47] used microarchitecture-level hardware events to classify and predict longer-than-average phases of different lengths. During the training period for their prediction phase classification technique, the authors recorded the density of L1 instruction cache (i-cache) misses. They then computed the difference of L1 i-cache misses between two intervals. If the difference was more significant—exceeding a higher threshold—than that of the previous two intervals, a new pattern was started. The prediction used a table-based technique that stored information gathered during the training period in a table. The technique then referenced the table during phase prediction. The authors stored identified patterns in a pattern database, or table. Along with the patterns, the table also stored three common *repetition distances*. A repetition distance is the number of instructions between two occurrences of the same phase. Analyzing the repetition distance enabled the proposed technique to predict phase changes before they occurred.

Rather than using hardware characteristics for phase classification, Khan et al. [58] presented a classification technique that counted executed instruction types and stored the information in an *Instruction Type Vector (ITV)*. The authors considered various instruction types: integer ALU, complex, branch, load, store, floating point ALU, multiply, and divide instructions. Their technique utilized fixed-length intervals of 10 million instructions, and counted occurrences of the instruction types within each interval. The authors then used the Manhattan distance between two intervals to cluster the intervals into application phases. One key feature of this technique is that it uses microarchitecture-independent application characteristics for phase classification. However, the technique may accrue additional computation and storage overhead, since the ITV must be computed and the intermediate data must be stored during runtime.

Chiu et al. [25] found that they could use variable-length intervals to improve classification accuracy. The authors proposed a technique that first traced an application's ex-

TABLE 2: Overview of online phase classification techniques

	Serial	Interval type	Fixed [101] [24] [47] [58] [89]; variable [25]
		Interval width	10K-15K instructions [101]; 10M [58]; 100M [89] instructions; 1s [24]; 100K branches [25]
		Classification metrics	Resource bottlenecks [101]; execution vectors [24]; cache misses [47]; instruction type vectors [58]; branch edges/method entry/returns [25]; conditional branches [89]
		Analysis method	Manhattan distance [101] [24] [47] [89]; digital signal processing [58] [25]
	Parallel	Interval type	Fixed [86] [88] [41]
		Interval width	50K - 200K instructions [86]; 100M instructions [88]; execution time [41]
		Classification metrics	Instruction type vector [86]; conditional branch execution [88]; task execution frequency [41]
		Analysis method	Manhattan distance [86] [41]; Clustering algorithm [88]

ecution, profiling the function calls/returns and conditional branch instructions every 100,000 branches. Then, the technique applied a Gaussian mixture model [68] to cluster the different intervals based on the number of executed branches. If a cluster's number of executed branches was different from previous clusters, the cluster was determined to belong to a different phase. The major drawback to this technique is that it requires an offline training period, and cannot be fully implemented as a runtime technique.

Sembrant et al. [89] presented work to reduce the overhead of BBVs. They suggested using Precise Event Based Sampling (PEBS) [27] to directly measure sparse BBVs, or randomly sampled BBV execution frequencies. PEBS, developed by Intel, tracks the number of events that have occurred and records the CPU's state after N events, where N is a value chosen by the user. By declaring a perf_event to record the address of of every Nth branch instruction, the authors were able to record basic block frequency vectors. The authors found that counting only conditional branches sufficed for achieving accurate BBV identification. In addition, the authors developed a C/C++ library, called ScarPhase, that consolidates their phase-classification techniques for easy use.

7.2 Parallel Online Phase Classification

Some phase classification techniques modify a more simple serial and/or offline technique for online use with parallel applications. For example, Rodrigues et al. [86] modified the serial technique presented by Khan et al. [58]. The proposed technique reduced the number of entries in the Instruction Type Vector (ITV) to four—INT (integer), FP (floating point), iBJ (branch), and Mem (load and store). The authors then used the Manhattan distance between ITVs to detect phases. The authors found that they could combine phase classification with dynamic core morphing [87] to significantly improve the performance/watt of most multithreaded workloads.

Sembrant et al. [88] analyzed the phase size in various applications and found that data-parallel applications have shorter phase sizes as the number of threads increases. Because smaller phases indicate that hardware/software changes occur more frequently, runtime phase-guided optimizations often become more costly as the number of threads increases. This makes runtime phase optimization more difficult in highly parallel applications. In addition to their analysis, the authors presented a prediction phase classification technique. Their technique used ScarPhase [89] to classify application behavior. To accomplish phase classification, The authors [88] extended the ScarPhase library to support parallel applications by including the functionality to alternate between threads. ScarPhase classifies a phase for a particular thread and then classifies the phase for whichever thread finishes an interval next.

Ganeshpure et al. [41] designed a phase classification technique to run on an multi-processor system on chip (MPSoC). An MPSoC is a system with multiple processor cores (processing elements—PEs) connected by a Network on Chip. The authors' phase classification technique required one processing element to act as a leader and the rest as followers. The leader could also act as a follower by sharing resources with thread execution. In the classification technique, the followers detected local phases independently and transferred the information to the leader. Each follower stored a Follower Phase Vector (FPV), which is updated every interval. The FPVs stored execution clock cycles for task execution and destination PEs as well as the number of times the task information is encountered. When a new FPV is generated, it is compared to the previous FPV. If the Manhattan distance between FPVs was below a certain threshold, the FPVs were considered matching. A local phase was detected if three consecutive FPV pairs match. If all of the PEs detected local phases before one of the local phases ended, the system detected a global phase. These global phases were then used for thread scheduling.

8 Future Research Directions

Even though, as illustrated in this survey, phase classification is a relatively mature research area, there are still gaps in the state-of-the-art that have not yet been fully addressed. This section briefly highlights some of the existing challenges that must be addressed—and areas of computing in which the challenges exist—to fully leverage the benefits of emerging adaptable systems.

Phase classification for emerging multithreaded applications: One of the most important gaps is the fact that the majority of current effective phase classification techniques are designed for single-threaded applications. However, multithreaded applications are now ubiquitous in modern computer systems, including resource-constrained computing systems. Multithreaded applications are becoming even

more important with the emergence of new classes of big data applications, such as machine learning, graph processing, image processing, and databases. As prior work has shown, adaptability will be a necessary feature for computing architectures that target these application domains [34].

A distinguishing feature of these applications is that they are typically massively parallel with threads that run on large-scale architectures, such as many-core architectures with tens, hundreds, or even thousands of threads [11]. In addition, several of these threads may need to interact with each other to enable functional correctness of the executing applications. As such, to enable optimal performance of the architectures for these applications, novel phase classification techniques must be developed for a wide variety of multithreaded applications, from small-scale applications to large-scale massively parallel multithreaded applications. The few currently existing phase classification techniques for parallel applications typically assume independent application threads without explicit consideration of data sharing. There are currently no known phase classification techniques designed to explicitly cater to multithreaded applications that exhibit data sharing.

Multithreaded data-sharing applications, especially, pose significant new challenges for phase classification. Accurately classifying multithreaded applications' phases is much more difficult due to shared resources, inter-core dependencies, and shared data. Data sharing cores may share working sets; a thread's runtime characteristics may depend on the characteristics of another thread running on another core. These kinds of data sharing multithreaded applications are expected to remain a prominent feature of emerging embedded systems. Currently, there is a critical knowledge gap on the implications of inter-core dependencies and data sharing for runtime phase classification in multithreaded applications.

Embedded, mobile, and edge computing: Despite the prevalence of embedded systems and mobile computing, there are currently no phase classification techniques specifically designed for mobile devices. Previous work [82] suggests that the phases of mobile applications may be different from those of traditional desktop application, but a comprehensive study of mobile applications' phases still remains elusive. While existent techniques may be applicable to resource-constrained embedded systems, most phase classification techniques introduce hardware, runtime, or energy overheads. These overheads may be prohibitive for embedded systems with stringent resource constraints. There is still much room for improvement in minimizing the overheads imposed especially by runtime phase classification.

An emerging area of computing that is amenable to adaptability is *edge computing* [106]. Edge computing has emerged as a paradigm, in the framework of the Internet of Things (IoT), where computation is moved closer to edge data-gathering devices in order to mitigate the bandwidth and latency overheads of transmitting data to the cloud for computation. An important characteristic of edge computing systems is the ability of the systems to be adaptable, not only to applications' changing requirements, but also to execution contexts, environmental factors, etc. [30]. Edge computing design choices like computation migration [90], for example, rely on the classification of application phases or tasks that will run on the edge device vs. the cloud or fog level of the IoT hierarchy. To fully satisfy the adaptability requirements of edge computing, new phase classification techniques must be developed to specifically satisfy the requirements of edge computing systems, including context awareness, low energy consumption, the need for computation migration, etc.

Security-aware phase classification: Phase classification may also come into play in security applications. Due to the fact that computer systems operate in dynamic environments, security mechanisms must also be adaptable to the inherent dynamism of the computing environments [81, 3]. Furthermore, different application phases may have different threat levels, and the phases must be classified in order to enable the design of security mechanisms that guarantee the required levels of protection for the different phases. As such, security-aware phase classification techniques need to be developed to incorporate security objective functions as part of the metrics for evaluating the characteristics of the different phases.

Transient phases: Another area that warrants further study in phase classification is the impact of *transition phases* [64] on the overall classification accuracy. Transition phases refer to the execution periods between stable phases. Most current techniques ignore these transition phases, which, if considered, may substantially change the classification technique's accuracy. Conversely, transition phases, if accurately detected and characterized, may offer additional opportunities for improving the specialization of system resources to application requirements.

User-aware phase classification: Finally, most current phase classification techniques are monolithic. Even though the main goal of phase classification is to exploit the variety of application execution characteristics for system adaptability, several factors can impact the application behavior during runtime. An application's phase characteristics may change drastically throughout execution as a result of multiple factors, such as new data inputs, system execution conditions, or user requirements. The design of phase classification techniques is currently disjoint with these factors, which can limit the achievable optimization from dynamically adaptable computing. Thus, new dynamic phase classification techniques are required to robustly handle and integrate known application and system information with predictive models for runtime application and system behavior changes, and variable user requirements, which may be unknown at design time.

9 Conclusions

The benefits derived from adaptability are directly tied to the accuracy of identifying the points at which the system configurations must be changed. Thus, phase classification is an important initial step in the design of adaptable computer systems that can be specialized to variable application requirements. Phase classification also offers other benefits, including speeding up research simulations, enabling efficient runtime thread-to-core assignments, etc.

In this paper, we presented a survey of phase classification techniques for identifying program phases. We

categorized the different techniques based on several important characteristics, in order to highlight the techniques' similarities and differences. We also highlighted some of the gaps in the state-of-the-art to expose future important research directions on phase classification. We hope that this survey will provide researchers with valuable insights into the state-of-the-art in phase classification, and direction on how to further enhance the benefits of adaptable computer systems for a wide variety of emerging applications.

REFERENCES

[1] J. Abella and A. González. On reducing register pressure and energy in multiple-banked register files. In *Computer Design, 2003. Proceedings. 21st International Conference on*, pages 14–20. IEEE, 2003.

[2] T. Adegbija, A. Gordon-Ross, and A. Munir. Dynamic phase-based tuning for embedded systems using phase distance mapping. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 284–290. IEEE, 2012.

[3] M. Almorsy, J. Grundy, and A. S. Ibrahim. Adaptable, model-driven security engineering for saas cloud-based applications. *Automated software engineering*, 21(2):187–224, 2014.

[4] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 93–104. IEEE, 2004.

[5] E. K. Ardestani, E. Ebrahimi, G. Southern, and J. Renau. Thermal-aware sampling in architectural simulation. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 33–38. ACM, 2012.

[6] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. In *ACM SIGPLAN Notices*, volume 35, pages 47–65. ACM, 2000.

[7] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *ACM SIGPLAN Notices*, volume 37, pages 111–129. ACM, 2002.

[8] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[9] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 245–257. ACM, 2000.

[10] D. Bautista, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato. A simple power-aware scheduling for multicore systems when running real-time applications. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–7. IEEE, 2008.

[11] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *arXiv preprint arXiv:1802.09941*, 2018.

[12] O. Benomar, H. Sahraoui, and P. Poulin. Detecting program execution phases using heuristic search. In *International Symposium on Search Based Software Engineering*, pages 16–30. Springer, 2014.

[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[14] A. Branković, K. Stavrou, E. Gibert, and A. González. Accurate off-line phase classification for hw/sw co-designed processors. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, page 5. ACM, 2014.

[15] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 171–182. IEEE, 2001.

[16] V. Bui and M. A. Kim. Analysis of super fine-grained program phases. *Columbia University Technical Report*, 2017.

[17] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Journal of solid-state circuits*, 35(11):1571–1580, 2000.

[18] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH computer architecture news*, 25(3):13–25, 1997.

[19] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the 11th Great Lakes symposium on VLSI*, pages 73–78. ACM, 2001.

[20] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *International Workshop on Power-Aware Computer Systems*, pages 25–39. Springer, 2000.

[21] T. E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multi-threaded applications. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 2–12. IEEE, 2013.

[22] M. Casas, R. M. Badia, and J. Labarta. Automatic phase detection and structure extraction of mpi applications. *The International Journal of High Performance Computing Applications*, 24(3):335–360, 2010.

[23] L. Chen, X. Zou, J. Lei, and Z. Liu. Dynamically reconfigurable cache for low-power embedded system. In *Natural Computation, 2007. ICNC 2007. Third International Conference on*, volume 5, pages 180–184. Ieee, 2007.

[24] G. L. T. Chetsa, L. Lefevre, J.-M. Pierson, P. Stolf, and G. Da Costa. A user friendly phase detection methodology for hpc systems' analysis. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 118–125. IEEE, 2013.

[25] M.-C. Chiu, B. Marlin, and E. Moss. Real-time program-specific phase change detection for java programs. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, page 12. ACM, 2016.

[26] M.-C. Chiu and E. Moss. Run-time program-specific phase prediction for python programs. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, pages 191–205. ACM, 2018.

[27] I. Corporation. Precise event based sampling (pebs). *Intel 64 and IA-32 Architectures Software Developers Manual*, 3b: system programming guide edition(30.4.4), 2010.

[28] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1396–1410, 2008.

[29] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 157–166. ACM, 2006.

[30] M. D'Angelo. Decentralized self-adaptive computing at the edge. In *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 144–148. IEEE, 2018.

[31] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[32] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 233–244. IEEE, 2002.

[33] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 217. IEEE Computer Society, 2003.

[34] J. R. Doppa, R. G. Kim, M. Isakov, M. A. Kinsy, H. J. Kwon, and T. Krishna. Adaptive manycore architectures for big data computing. In *Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip*, page 20. ACM, 2017.

[35] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, number LABOS-CONF-2006-014, pages 141–152, 2002.

[36] L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 2–12. IEEE, 2005.

[37] A. Efthymiou and J. D. Garside. Adaptive pipeline structures for speculation control. In *Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium on*, pages 46–55. IEEE, 2003.

[38] D. Folegnani and A. González. Energy-effective issue logic. In *ACM SIGARCH Computer Architecture News*, volume 29, pages 230–239. ACM, 2001.

[39] K. Fürlinger and M. Gerndt. ompp: A profiling tool for openmp. In *OpenMP Shared Memory Parallel Programming*, pages 15–23. Springer, 2008.

[40] K. Fürlinger and S. Moore. Detection and analysis of iterative behavior in parallel applications. In *International Conference on Computational Science*, pages 261–267. Springer, 2008.

[41] K. Ganeshpure and S. Kundu. On runtime task graph extraction in mpsoc. In *VLSI (ISVLSI), 2013 IEEE Computer Society Annual Symposium on*, pages 171–176. IEEE, 2013.

[42] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in java workloads. *ACM SIGPLAN Notices*, 39(10):270–287, 2004.

[43] A. Gordon-Ross, J. Lau, and B. Calder. Phase-based cache reconfiguration for a highly-configurable two-level cache hierarchy. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 379–382. ACM, 2008.

[44] A. Gordon-Ross and F. Vahid. A self-tuning configurable cache. In *Proceedings of the 44th annual Design Automation Conference*, pages 234–237. ACM, 2007.

[45] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 122–131. IEEE Computer Society, 1998.

[46] D. Gu and C. Verbrugge. A survey of phase analysis: Techniques, evaluation and applications. *Technical Report SABLE-TR-2006–1*, 2006.

[47] D. Gu and C. Verbrugge. Using hardware data to detect repetitive program behavior. Technical report, Technical Report SABLE-TR-2007-2, Sable Research Group, School of Computer Science, McGill University, Montréal, Québec, Canada, 2007.

[48] D. Gu and C. Verbrugge. Phase-based adaptive recompilation in a jvm. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 24–34. ACM, 2008.

[49] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism*, 7(4):1–28, 2005.

[50] S. Heo, K. Barr, and K. Asanović. Reducing power density through activity migration. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 217–222. ACM, 2003.

[51] O. Herescu, B. Olszewski, and H. Hua. performance workloads characterization on power5 with simultaneous multi threading support. In *Proceedings of the Eighth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 15–23, 2005.

[52] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE micro*, 27(3):63–72, 2007.

[53] M. C. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: application to energy reduction. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 157–168. IEEE, 2003.

[54] T. Huffmire and T. Sherwood. Wavelet-based phase classification. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 95–104. ACM, 2006.

[55] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th annual IEEE/ACM international symposium on microarchitecture*, pages 347–358. IEEE Computer Society, 2006.

[56] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 155–166. IEEE Computer Society, 1998.

[57] O. Khan and S. Kundu. A self-adaptive scheduler for asymmetric multi-cores. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pages 397–400. ACM, 2010.

[58] O. Khan and S. Kundu. Microvisor: a runtime architecture for thermal management in chip multiprocessors. In *Transactions on High-Performance Embedded Architectures and Compilers IV*, pages 84–110. Springer, 2011.

[59] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92. IEEE, 2003.

[60] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 64–75. IEEE, 2004.

[61] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 135–146. IEEE, 2005.

[62] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 236–247. IEEE, 2005.

[63] J. Lau, S. Schoemackers, and B. Calder. Structures for phase classification. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pages 57–67. IEEE, 2004.

[64] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 278–289. IEEE, 2005.

[65] A. Limaye and T. Adegbija. A workload characterization of the spec cpu2017 benchmark suite. In *Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on*, pages 149–158. IEEE, 2018.

[66] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[67] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 14–27. ACM, 2003.

[68] G. McLachlan and D. Peel. *Finite mixture models*. John Wiley & Sons, 2004.

[69] K. Meng, R. Joseph, R. P. Dick, and L. Shang. Multi-optimization power management for chip multiprocessors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 177–186. ACM, 2008.

[70] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W.-M. Hwu. An architectural framework for runtime optimization. *IEEE transactions on Computers*, 50(6):567–589, 2001.

[71] P. Nagpurkar, P. Hind, C. Krintz, P. F. Sweeney, and V. Rajan. Online phase detection algorithms. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 13–pp. IEEE, 2006.

[72] H. H. Najaf-Abadi and E. Rotenberg. Architectural contesting. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 189–200. IEEE, 2009.

[73] N. Nethercote and J. Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[74] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 81–92. IEEE, 2004.

[75] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 150–162. IEEE Computer Society, 2007.

[76] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[77] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 10–20. IEEE, 2005.

[78] A. Phansalkar, A. Joshi, and L. K. John. Subsetting the spec cpu2006 benchmark suite. *ACM SIGARCH Computer Architecture News*, 35(1):69–76, 2007.

[79] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 90–101. IEEE Computer Society, 2001.

[80] J. A. Poovey, T. M. Conte, M. Levy, and S. Gal-On. A benchmark characterization of the eembc benchmark suite. *IEEE micro*, 29(5):18–29, 2009.

[81] J. Portilla, A. Otero, E. de la Torre, T. Riesgo, O. Stecklina, S. Peter, and P. Langendörfer. Adaptable security in wireless sensor networks by using reconfigurable ecc hardware coprocessors. *International Journal of Distributed Sensor Networks*, 6(1):740823, 2010.

[82] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and Y. Handong. Application-specific performance-aware energy optimization on android mobile devices. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 169–180. IEEE, 2017.

[83] R. Rao, A. Srivastava, D. Blaauw, and D. Sylvester. Statistical analysis of subthreshold leakage current for vlsi circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(2):131–139, 2004.

[84] P. Ratanaworabhan and M. Burtscher. Program phase detection based on critical basic block transitions. In *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 11–21. IEEE, 2008.

[85] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, page 22. ACM, 2004.

[86] R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu. Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing. *ACM Trans. Des. Autom. Electron. Syst.*, 18(1):5:1–5:23, Jan. 2013.

[87] R. Rodrigues, A. Annamalai, I. Koren, S. Kundu, and O. Khan. Performance per watt benefits of dynamic core morphing in asymmetric multicores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 121–130. IEEE, 2011.

[88] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase behavior in serial and parallel applications. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 47–58. IEEE, 2012.

[89] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 104–115, Nov 2011.

[90] S. Shahhosseini, I. Azimi, A. Anzanpour, A. Jantsch, P. Liljeberg, N. Dutt, and A. M. Rahmani. Dynamic computation migration at the edge: Is there an optimal choice? 2019.

[91] X. Shen, M. L. Scott, C. Zhang, S. Dwarkadas, C. Ding, and M. Ogihara. Analysis of input-dependent program behavior using active profiling. In *Proceedings of the 2007 workshop on Experimental computer science*, page 5. ACM, 2007.

[92] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. *ACM SIGPLAN Notices*, 39(11):165–176, 2004.

[93] X. Shen, Y. Zhong, and C. Ding. Predicting locality phases for dynamic memory optimization. *Journal of Parallel and Distributed Computing*, 67(7):783–796, 2007.

[94] T. Sherwood and B. Calder. Time varying behavior of programs. *In UC San Diego*, 1999.

[95] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 3–14. IEEE, 2001.

[96] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ACM SIGARCH Computer Architecture News*, volume 30, pages 45–57. ACM, 2002.

[97] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 336–349. ACM, 2003.

[98] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM journal of research and development*, 49(4.5):505–521, 2005.

[99] S. Srinivasan, K. P. Ganeshpure, and S. Kundu. Maximizing hotspot temperature: Wavelet based modelling of heating and cooling profile of functional workloads. In *Quality Electronic Design (ISQED), 2011 12th International Symposium on*, pages 1–7. IEEE, 2011.

[100] S. Srinivasan, K. P. Ganeshpure, and S. Kundu. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(12):1867–1880, Dec 2012.

[101] S. Srinivasan, I. Koren, and S. Kundu. Improving performance per watt of non-monotonic multicore processors via bottleneck-based online program phase classification. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on*, pages 528–535. IEEE, 2016.

[102] A. Srivastava and A. Eustace. *ATOM: A system for building customized program analysis tools*, volume 29. ACM, 1994.

[103] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 39(4):528–539, 2004.

[104] L. Tee, S. Lee, and C.-k. Tsai. A scheduling with dvs mechanism for embedded multicore real-time systems. *International Journal of Digital Content Technology and its Applications*, 5(4), 2011.

[105] A. Tiwari, S. R. Sarangi, and J. Torrellas. Recycle:: pipeline adaptation to tolerate process variation. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 323–334. ACM, 2007.

[106] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 20–26. IEEE, 2016.

[107] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 136–146. IEEE, 2003.

[108] W. Zhang, J. Li, Y. Li, and H. Chen. Multilevel phase analysis. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(2):31, 2015.

Keeley Criswell received her M.S. in Electrical and Computer Engineering from the University of Arizona in 2018 and her B.S. in Physics and Computer Science from Thiel College in 2015.

Tosiron Adegbija (M'11) received his M.S and Ph.D in Electrical and Computer Engineering from the University of Florida in 2011 and 2015, respectively and his B.Eng in Electrical Engineering from the University of Ilorin, Nigeria in 2005.

He is currently an Assistant Professor of Electrical and Computer Engineering at the University of Arizona, USA. His research interests are in computer architecture, with emphasis on adaptable computing, low-power embedded systems design and optimization methodologies, and microprocessor optimizations for the Internet of Things (IoT).

Dr. Adegbija was a recipient of the CAREER Award from the National Science Foundation in 2019 and the Best Paper Award at the Ph.D forum of IEEE Computer Society Annual Symposium on VLSI (ISVLSI) in 2014.