BarrierWatch: Characterizing Multithreaded Workloads across and within Program-Defined Epochs

Socrates Demetriades and Sangyeun Cho
Computer Science Department
University of Pittsburgh
{socrates,cho}@cs.pitt.edu *

ABSTRACT

Characterizing the dynamic behavior of a program's execution is essential for optimizing the program on a given system. Once the program's repetitive execution phases (and their boundaries) have been correctly identified, various phase-aware optimization techniques can be applied. Multithreaded workloads exhibit dynamic behavior that is further affected by the sharing of data and platform resources. As computer systems and workloads become denser and more parallel, this effect is expected to intensify the dynamicity of the executed workload.

In this work, we introduce a new relaxed concept for a parallel program phase, called *epoch*. Epochs are defined as time intervals between global synchronization points that programmers insert into their program codes for correct parallel execution. We characterize the behavior of multithreaded workloads across and within epochs and show that epochs have consistent and repetitive behaviors while their boundaries naturally indicate a shift in program behavior. We show that epoch changes can be easily captured at run time without complex monitoring and decision mechanisms and we employ simple run-time techniques to enable epoch-based adaptation. To highlight the efficacy of our approach, we present a case study of an epoch-based adaptive chip multiprocessor (CMP) architecture. We conclude that our approach provides an attractive new framework for lightweight phase-based resource management for future CMPs.

1. INTRODUCTION

Characterizing and understanding the behavior of programs is of vital importance to ensure next-generation chip multiprocessors (CMPs) perform well on their anticipated workloads. In the past, evidences of strong behavioral similarities between programs have driven computer architects to build systems that, on average, work well for a large range of applications [1]. Today, the switch to massively parallel CMP along with the necessity for higher resource utilization at low power calls for a more systematic study of particular workload characteristics and their time-varying behavior.

Shared-memory multithreaded workloads exhibit dynamic behavior that is further affected by the sharing of data and platform resources [2]. In future CMPs, the dominance of shared resources like on-chip network (NoC), cache capacity and memory bandwidth is expected to magnify the impact of a workload's time-varying behavior (potentially producing high oscillations) on performance and power. Hence, knowledge of particular behavior similarities and behavioral shifts during multithreaded execution will offer previously unexploited opportunities for more efficient resource management. For example, *if we had* a good prediction of global traffic volume changes, we could make the right speed-power trade-off and bandwidth allocation on a given NoC. In another example, if the varying sharing and communication patterns could be predicted *a priori*, new thread scheduling policies could be triggered.

The behavioral differences across different time intervals and the repetitive behavioral stability within them have motivated prior work to study techniques and mechanisms that can detect and exploit them by monitoring simple metrics such as IPC or last level cache miss rate [3–5]. Depending on the metric being monitored, however, the detected changes in program behavior are limited to what the chosen metric can sense. To overcome this limitation and create a more fundamental representation of *program phases*, researchers have proposed architecture-independent metrics that capture inherent characteristics of a program by tracking the executed code itself [6–15].

While the architecture-independent phase detection and classification methods work well for single-threaded workloads, they do not directly apply to multithreaded workloads [14, 17]. In particular, the inter-thread interference and data distribution can affect threads' individual behavior, making their phase tracking harder. Also, if phase tracking is done by monitoring the executing code, the relative progresses of co-scheduled threads (and other events that can "un-synchronize" them) will result in an inconsistent program phase representation and thereby hamper phase detection. Lastly, most existing methods are complex and require heavy monitoring mechanisms, hence, their adoption is narrowed mostly to simulation practices [9, 11, 12, 14, 16, 17].

In this work, we explore a new simple parallel program characterization approach using the concept of *epoch* in order to enable lightweight run-time detection and prediction

*This work was supported in part by the US National Science Foundation (NSF) under grants CCF-1059283, CCF-0952273, and CCF-0702236.

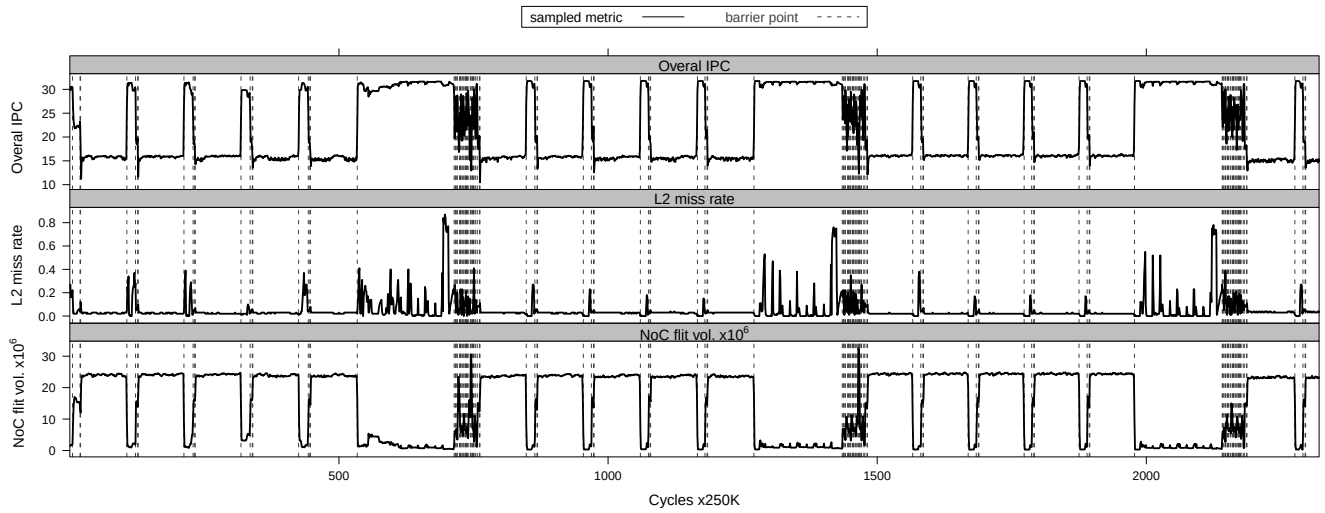

Figure 1: ***Key performance metrics of*** *bodytrack****:*** **Measurements were averaged for each sampling interval in time. The sampling interval size is 250K cycles. 'A,' 'B,' 'C' and 'D' are repeating global synchronization points in the program.**

of the repetitive and varying program behavior. Epochs are defined as program intervals between global barrier synchronization points that programmers have inserted into program codes to ensure the correct parallel execution of the program. The rationale behind our approach is that these synchronization points are likely to define sections of code that perform a unified, logically distinct task while naturally indicating a shift from a certain program phase to another.

Figure 1 illustrates the time-varying behavior of *bodytrack*, a common parallel application for tracking human body movements [18], on a 16-core shared memory CMP model (details of the system are in Section 3). The top graph shows the average system throughout (in IPC), the middle shows the overall L2 cache miss ratio of the 16 parallel threads, and the bottom the NoC traffic volume (in number of flits) as measured over the course of their entire execution. The graphs capture the natural relation of epochs to the repetitive and varying behavior of the program. The vertical dotted lines indicate the global synchronization points (barriers). As shown, there is a natural alignment of the synchronization points with the changes in program behavior, independent of the observed metric. Also, the barrier-bounded intervals (epochs) have a consistent and repetitive behavior across the program execution. Motivated by these observations, we propose watching the barriers ("BarrierWatch") to sense behavioral transitions.

For program behavior analysis, barriers carry four attractive properties: (1) They are tightly program code related and therefore can be watched and identified independently of the underlying architecture; (2) They indicate points that characterize a parallel program in a "global" sense as well as at individual thread level. In other words, they are the only points of execution that can stably define a global program state; (3) They appear in a majority of multithreaded workloads we examined and are easily "watchable" at run time by simple architectural support; and (4) Barriers are relatively rare events and associating information with barriers will lead to smaller overheads than with other finer granularity.

To the best of our knowledge, our work is the first to explore in detail the program-defined epochs and correlate them with the the varying behavior of the parallel multithreaded workloads. In summary, we make the following contributions in this paper:

- We examine the epoch-related properties of a number of parallel applications and we characterize their epoch-level behavior on an elaborate CMP model (Sections 2 and 4). Our characterization confirms the correlation between epochs and varying program behavior by identifying prominent behavioral similarities across repetitive instances of each epoch while pointing out significant changes that occur across their boundaries.

- We propose *BarrierWatch*, a lightweight technique for supporting epoch-based adaptation (Section 5). The proposed technique watches for barriers to detect the beginning and end of each epoch and associates them with appropriate adaptation decisions; and

- We highlight the efficacy of our proposed technique with an elaborate case study of an epoch-based adaptive CMP architecture (Section 6).

We anticipate that our approach will provide an attractive new framework for lightweight phase-based resource management for future large-scale CMPs.

2. PROGRAM EPOCHS

In this section, we first define program epochs and qualitatively discuss what benefits our definition brings. Next, we introduce a set of multithreaded benchmarks used in our study. Finally we present a high-level epoch profiling result of the benchmarks.

Programmers insert synchronization points (e.g., barriers and locks) into the code to avoid race conditions and ensure the validity of the data values during a parallel execution. This work focuses on the global synchronization points (barriers) that synchronize all running threads in a program globally. Global barrier points can be found within parallel

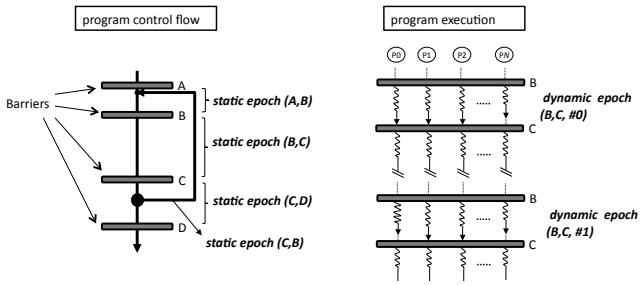

Figure 2: Static (left) vs. dynamic epoch (right).

Benchmark	Num. static epochs	Repeatable epochs	Program input size	Num. dyn. epochs
bodytrack	4	4/4	simlarge	19
fluidanimate	8	8/8	simlarge	39
streamcluster	21	16/21	simmedium	7,569
barnes	3	2/3	64K (particles)	8
fmm	10	6/10	64K (particles)	33
lu	5	2/5	2,048 (matrix)	258
ocean	24	18/24	1,026 (grid)	707
radiosity	5	2/5	largeroom	17
water-ns	11	5/11	1,000 (molecules)	19

Table 1: Epoch statistics of benchmarks.

sections as well as at the end of parallel sections (joins). In any case, barriers can be seen as points that split the program into subsections that perform boundary-synchronized parallel tasks. We will refer to each of those tasks as an *epoch*. Therefore, an epoch starts when a barrier is released and ends when the next barrier is reached. Each *static epoch* is uniquely identified with a tuple containing the IDs (e.g., PC or a compiler-generated number) of the two enclosing barriers. During execution, a static epoch could be exercised multiple times, creating multiple dynamic instances. A *dynamic epoch* can be identified with the corresponding static epoch ID and how many times the static epoch has been executed so far. Figure 2 depicts the notion of static and dynamic epoch.

In a fork-join model of parallel execution such as the one heavily used in OpenMP programs, execution alternates between parallel and sequential sections repetitively, creating oftentimes epochs with both sequential and parallel regions. To further distinguish those regions, epoch definition can be extended to consdier all fork and join points as epoch boundaries.

It is well known that program codes repeat (e.g., loops). Since the program behavior is strongly related to the code that is executing, the repeated execution of the same code can yield very similar behavior. In general, this repetition has been the basis for predicting a program's future behavior. We find that epochs repeat and that their instances exhibit similar behavior, while their boundaries indicate shifts in program behavior. In the examples of Figures 1 and 2, epochs such as epoch(B,C) are likely to perform a single, logically distinct operation that is repeated throughout the program execution. Consequently, epoch boundaries indicate a transition of the program execution into a different code section that could stress the architecture in a completely different way.

We note here that *epochs are not related with the conventional definition of program phase*, where program is analyzed/monitored and program phases are formed from intervals having strong behavioral similarity [16]. Epochs naturally partition the program code into unique, contiguous, variable-length intervals. At run time, the execution of a program can be viewed as a sequence of dynamic instances of these epochs. Marking epoch boundaries is done automatically, without a need for run-time monitoring, profile- or compiler-based analysis. Nevertheless, program behavior within an epoch may not necessarily be constant (e.g., epoch(C,D) in Figure 1).

The relation of the epochs with the code itself has important advantages. First, analyzing the behavioral repetition across instances of the same static code provides reassurance that observed similarity is an inherent characteristic of the program and not an architecture-dependent coincidence. Second, unlike an arbitrarily chosen fixed-size instruction interval, epoch boundaries can naturally and more precisely align with the phase transitions. Third, barriers, as points in the code, can be easily captured at run time with simple architectural support. Finally, many parallel programs are structured with barriers.

In Table 1 we list a selection of commercial and scientific multithreaded programs that we evaluate in this work. These programs use *pthread*, a standard shared memory programming library that supports a barrier interface. The first three programs are taken from the PARSEC 2.0 benchmark suite [18] and the rest from SPLASH-2 [19]. Other programs from those suites either do not use barrier synchronization, or their barriers do not reside inside repeatable sections. Our proposed approach applies only to shared-memory parallel programs that use barriers.

The table shows each program's basic epoch statistics: the number of distinct static epochs, the fraction of the static epochs that have dynamic instances, and the total number of dynamic epoch instances for the referred input size. The table shows that static epochs, if not all, repeat. Some programs have a large number of dynamic epochs generated from a few epochs, such as *ocean*, *lu*, and *streamcluster*, because these epochs reside inside heavily repetitive loops.

Figure 3 shows the relative size of static epochs and their importance. Epoch size intuitively shows how small or large a static epoch is and corresponds to the average dynamic instruction count of the epoch. A large size does not necessarily reflects a time-dominant epoch, since its contribution to the total execution time depends not only on its size, but also on the number of times it is executed. To better illustrate this relation, we order them from bottom to top, with the most time-dominant epoch being at the bottom. Also, in Figure 4, we show their total contribution to the execution time in a cumulative graph, using the same order.

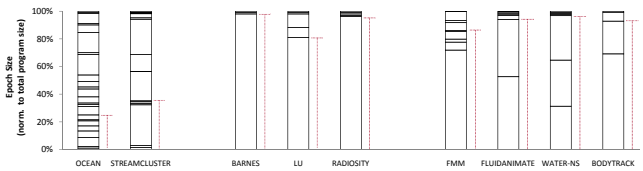

Figure 3: ***Relative static epoch sizes (normalized to the program size)*****: Epochs are ordered from bottom to top based on their execution time contribution. The bars show the fraction of epochs that contribute more than 5% of the total execution (most important epochs).**

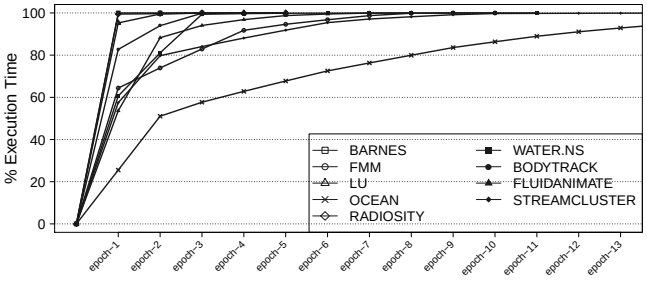

Figure 4: ***Execution time breakdown*: Time spent by each static epoch. Epochs are sorted based on their execution time contribution.**

Based on the profile results, we separate the benchmarks into three categories. In the first category, programs such as *ocean* and *streamcluster* spend most of their execution in a number of small epochs that are highly repetitive. In the second category (*fmm*, *bodytrack*, *fluidanimate* and *water-ns*), a large part of the execution time is consumed by a single epoch while the remaining execution time is split rather equally to the rest of the epochs. Finally, *barnes*, *lu* and *radiosity* belong to a different category where a single static epoch monopolizes the execution time. The large range of epoch granularity reflects the structural heterogeneity across and within different programs. At the two extremes, very fine or very coarse epochs may exhibit unstable or multi-phase behavior, respectively. Nonetheless, the epoch-level characterization in the following sections shows the promising aspects of our approach.

3. METHODOLOGY

Experimental Setup. For various measurements we employ an elaborate CMP simulator built on Simics [20]. Our simulator is a detailed timing simulator that models a 16-core tiled CMP with a 4×4 2D mesh network-on-chip (NoC), similar to the machine models used in recent studies and commercial developments [21–24]. Each tile of the processor incorporates a compute core, an L2 cache bank, coherence support and a NoC router. A compute core is a two-issue in-order processor and has private L1 caches. Per-tile L2 caches form a globally shared logical L2 cache. Cache coherence is maintained by a distributed directory-based coherence protocol with MESI states, modeled after [25]. Table 2 summarizes our architecture configuration parameters.

We collect information for the programs listed in Table 1. For compilation, we use `gcc` version 4.2.0 with the `-O3` optimization level and the SunOS pthread library version 5.10. All programs were executed from start to finish with the listed problem size, referenced by [18, 19]. Statistics were collected starting from the beginning of the parallel section till the end. We use all available processor cores by spawning 16 threads in all experiments. Each thread was bound to a specific core for stable and repeatable measurements.

Performance Metrics. To characterize the program behavior across epochs, we use system-wide global metrics that measure overall system throughput (sum of IPCs), L2 cache miss ratio, NoC network traffic volume and cache-to-cache transfer hit ratio (denoted as "C2C-transfer hit ratio"). NoC traffic includes memory controller requests, remote L2 requests, cache-to-cache requests and all control and coherence messages. C2C-transfer hit ratio is defined as the fraction of L2 misses that were satisfied on chip, by the cache-to-cache transfer protocol. Although further architecture-dependent and independent metrics can be used, we limit our study to the above metrics as they are sufficient to prove the concept of this work.

Parameter	*Value*
Processor model	in-order
Issue width	2
L2 Cache	
Line size	64 B
Size/Associativity	128 KB, 8-way
Tag latency	2 cycles
Data latency	6 cycles
Replacement policy	LRU

Parameter	*Value*
L1 I/D Cache	
Line size	64 B
Size/Associativity	8 KB, direct-map
Load-to-Use latency	2 cycles
Network on Chip	
Topology	4×4 2D mesh
Hop latency	3 cycles
Main mem. latency	300 cycles

Table 2: Machine architecture configuration.

Variation Metrics. To demonstrate the relation between epochs and time-varying behavior of the program we need statistical methods that can characterize the variation across different epoch instances. All our measurements on variation are based on standard deviation.[1] Because most of our metrics are on a ratio scale, standard deviation indicates the variation in terms of percentage. Thus, different variation around different means can be directly compared. To meaningfully include in our displays deviations of non-ratio metrics (global IPC and traffic volume) together with ratio metrics, we apply to them a rescaling factor based on the variation of their values.[2] For the epoch-level analysis, we exclude any small epoch instances that appear as "noise" between longer epochs during execution. Such very short time intervals cannot reach a stable and unbiased microarchitectural state (i.e., they experience only the warm-up period), and therefore performance metrics cannot precisely capture their actual characteristics. We consider 50K dynamic instructions as an acceptable size for an epoch instance to be included in our analysis. As an exception, we include the small epochs that are repeated multiple times back-to-back. These epochs can be reliably measured because their repetition creates a robust microarchitectural state. All other static epochs and their dynamic instances are included, independently of their granularity.

4. EPOCH LEVEL CHARACTERIZATION

4.1 Variation across an epoch's instances

To characterize an epoch's behavior as a reoccurring pattern, we measure the variation of four different architecture metrics across the dynamic instances of each epoch. Figure 5 plots this variation for all four epochs of bodytrack.

The results show low variability for all the metrics across the instances of each static epoch. The percentages can directly evaluate the *similarity* or *stability* across an epoch's dynamic instances for the specific metric. For example, IPC varies around 2% across the instances of epoch-4. Assuming that 2% is a negligible variation, those instances can be considered similar to each other. Epochs with low variation for all the metrics reflect an epoch with a stable dynamic be-

[1]Since our samples are relatively few, we use an unbiased estimator for the standard deviation with a degree of freedom $N-1$.

[2]We prefer this method rather than using Coefficient of Variation (CoV) since CoV can only be used to compare the amount of variance between populations with means that are close together.

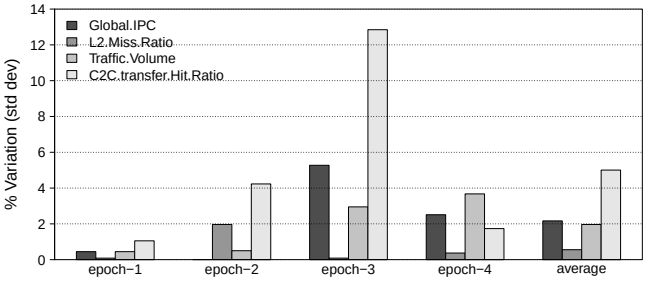

Figure 5: ***Variation across each epoch's dynamic instances for bodytrack*: The variation measures the standard deviation across the instances of each epoch.**

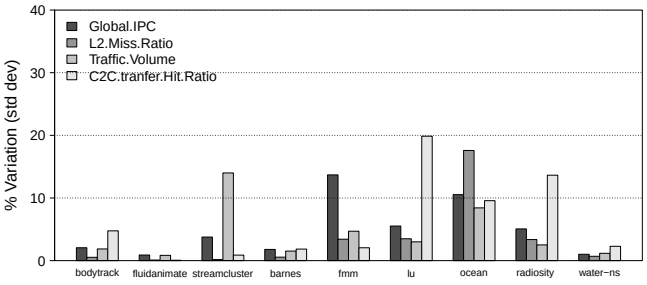

Figure 6: ***Variation across each epoch's dynamic instances*: The lower the variation, the more effective is the epoch granularity in recognizing the cyclic patterns in program behavior.**

havior. Comparing the variation between different epochs gives their relative stability. For example, the performance variation across the instances of epoch-3 is larger than that of epoch-1. Relatively higher variation is observed for C2C-transfer hit ratios, mainly because this metric is strongly affected by the prior behavior to the barrier, e.g., the cache state the previous epoch leaves. This preceding state could be different for each instance of an epoch. The best performance stability is demonstrated in epoch-1, which is the most time-dominant epoch in bodytrack.

To extend our observations to other programs as well, we summarize in Figure 6 the average variation that is observed among the dynamic instances of each epoch, for each program. The average variation is weighted by the number of the dynamic instances of each epoch.[3] As the figure shows, most programs have very limited variation for all the metrics, highlighting the effectiveness of epochs defining periodic execution intervals that exhibit similar program behavior.

4.2 Variation across different epochs

Figure 7 shows the variability in behavior among different epochs of bodytrack, as measured by each metric. In this example, each point reports the mean behavior of each epoch across its different instances. The error bars indicate the variation among the instances of the epoch, based on the variation analysis in the previous subsection.

The existence of high behavioral variability across different epochs (compared with low variability across instances

[3]Valuable only from a statistical point of view; epochs with many instances are more unbiased estimators of the variation than epochs with fewer instances.

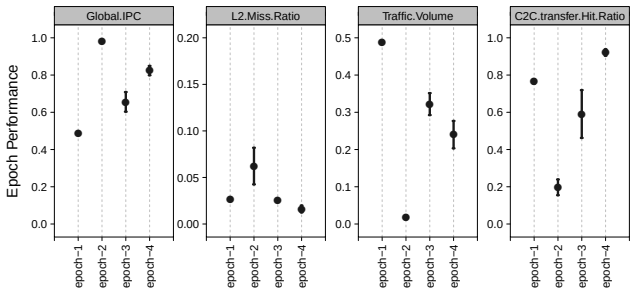

Figure 7: ***Epoch-level Behavior of bodytrack*: The behavior across different epochs varies significantly, compared with the variation that exists among epoch instances (denoted with error bars).**

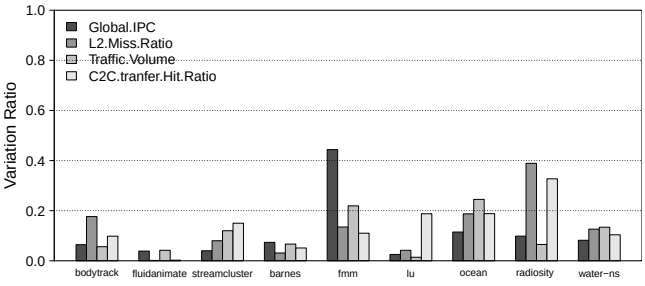

Figure 8: ***Ratio between the average variation across epoch instances and the variation across the different epochs*: The smaller the ratio, the sharper the behavioral shifts on epoch boundaries.**

of an epoch) indicates a fundamental correlation between epoch boundaries and changes in program behavior. A single metric that can capture this property (e.g., global IPC in the example of Figure 7) is adequate to show that there is such correlation. However, showing more than one metrics is desirable since the same single metric may not be suitable for capturing the behavior changes in every application, even if such a metric exists (e.g., L2 miss ratio is fairly insensitive in the same example). In addition, the sensitivity analysis over a number of metrics against different epochs provides useful insights into how possible epoch-based optimizations can be directed and implemented on a target architecture. For example, an adaptive system can exploit the large variability of a metric across different epochs to improve the performance or power consumption of individual epochs.

To quantify how different epochs exhibit distinguishable behavior, we need to meaningfully compare the heterogeneity that exists across different epochs with the homogeneity that exists across instances of the same epoch. To measure this quantity, we use the ratio between the weighted average of the variation that is accounted for the instances of each epoch (see error bars in Figure 7), and the variation that exists across the different epochs. The last part calculates the variation between the mean values taken from each epoch, compared with the grand mean of all the epochs' instances of the program. This ratio is related with a statistical metric that is widely used in statistical analysis of variance (ANOVA) and calculates the significance of the variation between populations that appear to have similar variation

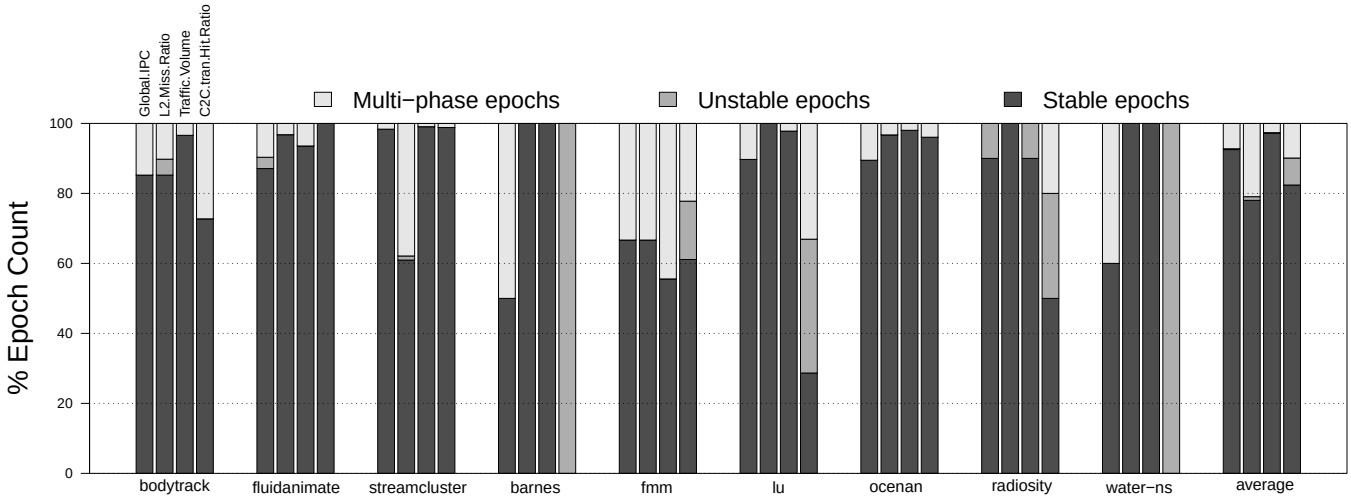

Figure 9: ***Program Behavior within Epochs (breakdown)*: Epochs show stable, unstable, or multi-phase behavior.**

within them [26]. Figure 8 shows the results for all the programs, and for each metric. If the behavior change across different epochs is significantly more marked than across their instances, we should see, at least for some metrics, a small ratio between those standard deviations. The smaller the ratio, the better the distinction is.

The results show that all the examined parallel programs have at least some metrics with a very small ratio, indicating the power of epochs to distinguish the program behavior into well-defined intervals. The relatively high ratio in fmm, radiosity and water-ns comes from the fact that their epochs are generally more similar than in other programs and not due to high variability among their epochs' instances (which is shown to be low in Figure 6).

4.3 Program behavior within epochs

A program may exhibit further repetitive and shifting behaviors within epochs. For example, while most epochs in bodytrack have stable behavior, the L2 cache miss rate in epoch(C,D) (recall Figure 1) oscillates rapidly and as a result produces a highly "unstable" epoch. To gain more insight into the program behavior within epochs, we evaluate the relative stability or instability within their boundaries.

To perform our analysis, first we sample the behavior of each epoch using the four performance metrics. Sampling is done using fixed-length time intervals (250K cycles). Then, we split the epoch into *phases*[4], formed from consecutive intervals having stable behavior. To characterize the behavioral stability within the epoch, we measure the length of each phase as well as the average phase length of the epoch. Based on those measurements, we classify each epoch as *stable*, *unstable* or *multi-phase*. A stable epoch has a single relatively large phase (our criterion is >80% of the epoch length) whereas unstable epochs have frequent changes of program behavior, and therefore will have many short phases. Consequently, we determine an unstable epoch if it has a short average phase length (<5% of the total epoch length). The rest are classified as multi-phase epochs, expected to have more than a single stable phase.

Figure 9 presents the result of our classification using the above criteria. On average, more than 80% of all epoch instances have stable behavior. Some epoch instances show unstable C2C-transfer hit ratio, mainly because this metric is more sensitive to thread interference and fine-grain communication patterns.

Although the behavioral changes within an epoch are not automatically inferred at the epoch granularity, the stability or instability, as consistent repetitive behavioral patterns, can be highly predictable. For example, consecutive instances of epoch(C,D) in bodytrack can be correctly predicted as unstable. We have observed that most program epochs have a strong tendency to repeat their internal pattern in their dynamic instances. Few exceptions were observed in epoch instances having initialization parts or other relatively short epochs. Note that labeling an epoch as "unstable" in a courser sense could have a much higher benefit than tracking rapid changes of very short regions. For example, attempting to continuously adapt and reconfigure a system within unstable regions can lead to unpredictable and undesirable results. In contrast, associating unstable epochs with appropriate system configurations will provide more effective overall impact.

4.4 Summary

Our epoch characterization shows that: (1) Epochs define intervals that repeat in a consistent and predictable way and therefore they provide a reliable granularity in which the cyclic pattern of program behavior can be observed; (2) Different epochs tend to have different behavior and therefore they provide an attractive granularity in which the program can be characterized. Consequently, epoch boundaries are likely to naturally indicate changes of program behavior; and (3) Most epochs exhibit stable behavior within their boundaries. In general, internal behavior patterns reoccur and thus can be accurately predicted.

5. RUN-TIME SUPPORT

A run-time system capable of tracking changes of program behavior can trigger a search for an optimal configuration

[4]Here we follow the traditional notion of program phase, defined as a collection of consecutive intervals with no significant performance changes (set to <5%).

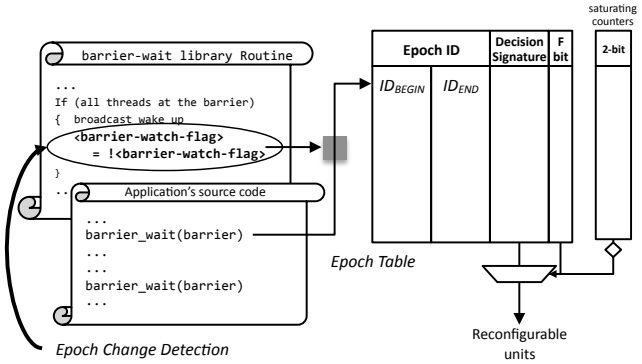

Figure 10: ***Support for Run-time Epoch Detection and Epoch-based Adaptation*: The BarrierWatch technique detects epoch transitions by capturing the barrier points. The Epoch Table associates epochs with decision signatures to be used for adaptive optimization.**

that will adapt the system to a performance and/or power optimal state. The adaptation can be achieved through configurable hardware mechanisms [3, 4, 6–8]. Using an epoch-based approach, a system can dynamically detect a change of program behavior by "watching" for a barrier point. Then a search for the best configuration can be triggered to create and store a decision signature for the running epoch. Upon new iterations, the system could use the epoch ID to automatically reinstate the optimal configuration using the decision signatures stored by the previously seen epochs.

5.1 Epoch change detection

A pass from one epoch to the next occurs when all the executing threads of a running program have reached a barrier. We capture this global synchronization point by watching a hardware flag that is toggled whenever a barrier is released. The "`barrier-watch-flag`" is exposed to the barrier synchronization routine where is set/reset only by the last thread reaching the barrier. (Figure 10). To support a hardware flag, a trivial modification is required in the existing libraries that implement the barriers. Note that no changes are required to the source code of application. Whenever the `barrier-watch-flag` signals a release of a barrier, a barrier identity ($\text{ID}_{Barrier}$), which is unique for each static barrier call, is passed as a search key to the *epoch table*. The barrier identity is the return address of the barrier routine (obtained from the call stack), assuming that the barrier routine call is not enclosed in tight wrapper functions.

5.2 Epoch table

The epoch table is a fully associative array that keeps the epoch IDs of the already executed epochs along with a decision mask and a filter bit for each ID. It can be implemented either as a software routine and linked with the program binary during compilation, or in hardware for fast adaptation decisions. A modest number of entries are sufficient to accommodate all the static epochs of a program (10 on average and 24 in the worse case for the evaluated programs). The ID field holds a tuple of two 8-bit barrier IDs (ID_{BEGIN} and ID_{END}) that correspond to the $\text{ID}_{Barrier}$ at the entry and exit point of the epoch. On a lookup operation, the search key is compared with each ID_{BEGIN} of the table for a match. If the entry is found, the *decision signature*, which is assumed to keep the optimal configuration for that epoch, will become available. The system can then use the decision signature to configure the hardware as directed. The use of the filter bit is to prevent, when necessary, the extraction of the decision mask. This might be desirable in several cases. For example, the decision mask might not yet exist or might not yet have reached a confident stage.[5] In another case, discussed further in Section 5.3, the filter bit suppresses the detection of very small epochs.

At the first instance of each epoch, no matching record is found in the table. This "compulsory miss" will allocate a new table entry, store the $\text{ID}_{Barrier}$ to the ID_{BEGIN} field and trigger an external process that will enable some kind of monitoring policy for the running epoch (e.g., enabling some hardware counters). At the end of the epoch, when the next barrier is reached and before the new lookup operation is performed, the new $\text{ID}_{Barrier}$ is stored in the ID_{END} field and the execution is transfered to a decision algorithm that evaluates the monitoring results, decides the best configuration and saves the decision signature in the table entry.

The reason for keeping both ID_{BEGIN} and ID_{END} in the ID field is two-fold. The first has to do with the possibility of having two different epochs with the same ID_{BEGIN}. In that case, the ID_{END} is required to identify a false positive match. The second is that knowing the ID_{END} of the current epoch allows us to predict the next epoch in the future. Assuming that the current epoch ID is detected (i.e., the ID_{BEGIN} is known), a lookup at the same entry of the table can retrieve the ID_{END}, which is the ID_{BEGIN} of the following epoch.

Two different epochs can have the same ID_{BEGIN} if a control point can lead the execution path into different barriers during different iterations. Such an example is illustrated in Figure 2, where a branch after barrier C redirects the execution to a previous point multiple times, before proceeding. In this case, epoch(C,B) and epoch(C,D) have the same ID_{BEGIN}. Such a case happens if an application uses barriers at loop iteration boundaries.

Assuming that epoch(C,B) is recorded in the epoch table, epoch(C,D) will be incorrectly identified as epoch(C,B) during its first execution. To identify a false positive match, the ID_{END} is compared with the $\text{ID}_{Barrier}$ when the epoch reaches its exit point. The first false positive match of the (few) epochs like epoch(B,B) is inevitable and a new table entry will be allocated to accommodate the new epoch, as if there was no match. The new entry will have the same ID_{BEGIN} but a different ID_{END}. The filter bit might need to remain "set" until more iterations of the epoch are monitored and the appropriate decision signature is determined.

The possibility of having more than one epochs with the same ID_{BEGIN} in the epoch table will result in lookups with multiple matches. To avoid new false positives, we need to accurately predict and pick the entry with the correct epoch ID. A simple 2-bit branch prediction like scheme [27] is likely to work well in this case, since the reoccurring sequence of such epochs is based on a few easy-to-predict branch decisions.[6] To explore such a scheme, a 2-bit saturating counter is added per entry. At the entry point of each epoch, if the

[5] Depending on the adaptation approach, a decision algorithm may need more than one iterations to determine the signature.

[6] Although more than two matching entries can exist, no such case was observed throughout our experiments.

Benchmark	Num. dyn. epochs	Num. mis-detected epochs	
		w/o pred.	w/ 2-bit pred.
bodytrack	19	4	1
fluidanimate	39	0	0
streamcluster	7,569	2,139	4
barnes	8	0	0
fmm	33	1	1
lu	258	1	1
ocean	707	40	4
radiosity	17	2	2
water-ns	19	4	2

Table 3: Number of mis-detected epochs with and without 2-bit prediction. Remaining epochs are detected correctly if they already existed in the epoch table.

$ID_{Barrier}$ matches two different entries, then the entry with the higher counter is picked. At the exit point, the prediction is evaluated by looking for a match with the ID_{END} of the picked entry. Based on the outcome of the evaluation, the counters of both candidate entries are updated.

Table 3 shows the effectiveness of the prediction method. Overall, the 2-bit predictor significantly reduces the mis-detected epochs. There are plenty of prediction models that can be used to further improve the accuracy of epoch detection or handle multi-match cases [5, 28]. However, fully exploring a large design space of such prediction mechanisms is beyond the scope of this work.

The size of the epoch table is determined by the number of static epochs of the program. This number is typically small mainly because programming tactics generally encourage balanced and limited use of global synchronization points. Therefore, a possible hardware implementation can offer fast table lookups with negligible area and power overheads. A table with 24 entries (worst case in our experiments) occupies less than 0.1KB of capacity, and can be integrated on chip. If more space is required, a simple displacement policy such as LRU is likely to perform well, since there may be epochs that are used only once.

5.3 Avoiding small epochs

Adaptation actions during epochs that appear as "noise" between longer epochs may have no practical benefit. Therefore, it is desirable for the epoch detection mechanism to recognize them. A small epoch can be detected either by the external process, which is called during the first instance of the epoch, or by a dedicated hardware. In both cases, the detection involves a calculation of the time difference between the entry and exit point of the epoch and a comparison with a predefined threshold (e.g., 50K cycles). After an epoch is labeled as small, it can be avoided either by enabling the filter bit of the corresponding entry or by not saving it in the epoch table. Using the first approach, future instances of the same epoch will automatically be classified as small and no optimization action will be taken.

6. A CASE STUDY

In this section, we evaluate the effectiveness of epoch-based adaptation in a CMP architecture with a case study. The adaptation aims to optimize the energy and performance trade-off using the Dynamic Voltage/Frequency Scaling (DVFS) technique applied to the NoC [29]. NoC DVFS extends the concept of per-core DVFS to per-router DVFS for congestion and energy management. In this study, we consider a simple NoC DVFS configuration in which all the routers of the network comply to the same voltage/frequency setting at a given time. Although this strategy is less flexible than per-router DVFS, it requires much smaller design and implementation effort.

Caption	Frequency	Voltage
$f_{100\%}$	3 GHz	0.8 V
$f_{75\%}$	2.25 GHz	0.65 V
$f_{50\%}$	1.5 GHz	0.5 V
$f_{25\%}$	0.75 GHz	0.35V

Table 4: Frequency/voltage levels. The linear relation between frequency and voltage is consistent with estimates in [30, 31].

Changes in the NoC's clock frequency shift performance and power to opposite directions. Decreasing the frequency leads to lower power consumption in the routers, but affects negatively their congestion and processing speed. In addition, operating the NoC in a lower frequency than the processor cores can potentially incur network overloading and result in significant performance penalties. Thus, careful control of the NoC frequency is needed to exploit the energy/performance trade-off.

We note that the purpose of this case study is not to propose a complete solution for the energy/performance management of the NoC; rather, we aim to demonstrate the applicability of our simple and low-cost approach in the context of dynamic adaptation. Fully exploring the design space and comparing against other adaptive techniques is beyond the scope of this work.

6.1 Epoch-based Adaptation

We present two different implementations of epoch-based adaptation. In the first one, off-line profiling determines the frequency/voltage setting best suited for each epoch of the application ("static scheme"). We perform a profile run for each NoC voltage/frequency setting to record the execution time (D_f) and energy consumed (E_f) by each epoch (all the combined dynamic instances of each epoch). Then, we pick, for each epoch, the frequency/voltage level f_x for which $E_{f_x} \times D_{f_x}$ is minimized, i.e., to hit the best trade-off between energy and performance. The selected frequency level for each epoch is then included in the binary of the application as a decision signature. At run time and at the beginning of every epoch instance, the signature is retrieved and the NoC switches to the desirable frequency/voltage level.

In the second implementation, the epoch table and the decision signatures are determined dynamically at run time ("dynamic scheme"). A search for the best frequency level is triggered at the first instance of each epoch. During this period, all possible NoC voltage/frequency settings are examined by switching to a different frequency on a fix time-interval basis within the epoch instance. Because the energy and performance measurements for each frequency have to be taken from different time intervals, we measure energy per instruction (EPI) and cycles per instruction (CPI). Thus, the best frequency f_x for each epoch becomes the one that minimizes $CPI_{f_x} \times EPI_{f_x}$. The selected frequency is then stored in the epoch table as a decision signature as described in Section 5. Upon new iterations of the same epoch, the signature is retrieved to adapt the system accordingly.

The static, profile-based approach is particularly suitable to embedded and special-purpose environments where the

execution environment is known *a priori*. Clearly, the static scheme is more effective than the dynamic scheme since optimal decisions will be readily available at run time. However, the dynamic scheme is more general and is applicable to a wider range of environments. We note here that even in dynamic environments, users often tend to execute a limited number of applications frequently, possibly solving similar problems repeatedly. In this case, the user (or the OS) can choose to store the applications' epoch tables to a persistent storage when they finish execution and pre-load them when the applications run again. Such a strategy can further improve the effectiveness of the epoch-based adaptation since all compulsory misses in the epoch table will be eliminated.

6.2 Experimental Setup

The experiments are performed using the same simulation environment as described in Section 3. In our simulator, the NoC models a wormhole-switched network with deterministic X-Y routing and ACK/NACK flow control. Data packets consist of six 128-bit flits and control messages one flit. Each router models a two-stage router pipeline and has 5 physical channels (PCs) and 2 Virtual Channels (VCs) multiplexed on each PC. We use a buffer size of 2 flits per VC.

For NoC DVFS, we assume four possible clock frequency and voltage levels, as shown in Table 4. $f_{100\%}$ represents the maximum operating frequency of the NoC, which we consider as the baseline frequency (no energy savings). DVFS policies are triggered during epoch transitions; assuming on-chip voltage regulators, we account 100 cycles for switching overhead.[7] For the dynamic implementation, we use 100k sampling intervals and thus the per-epoch monitoring period lasts at least 500K cycles—100k for warm-up + 100k (at least one sample) per frequency level. During this period, we keep high voltage and we switch only frequency, therefore we account zero switching overhead, while the energy consumption for the corresponding frequency level is estimated. Epoch instances are usually larger than 500k cycles, so a large number of samples are taken per frequency. When the first instance is smaller than 500k, the epoch is considered too short and is marked with the filter bit; later instances of the same epoch will be therefore skipped by the epoch detection mechanism.

Power consumption is modeled with dynamic, leakage, and background components. Dynamic power scales with fV^2 and leakage power with V. Background power, representing the cores and the rest of the system, is not in the same clock domain and consumes a constant amount of power. Leakage power consumption is assumed to be twice that of dynamic power when operate at 1GHz, which is consistent with estimates from Kim et al. [31]. Background power is computed assuming that a loaded NoC at 1 GHz consumes 30% of the total system power [32].

6.3 Results

Figures 11, 12, and 13 compare energy savings, execution slowdowns and energy-delay improvement, respectively, between non-adaptive fixed frequency schemes and epoch-based adaptation schemes, for the reference applications. Results are shown with respect to the baseline scheme, where the NoC operates at full frequency $f_{100\%}$.

[7]State-of-the-art on-chip voltage regulators can transition voltage even faster (5ns) [30].

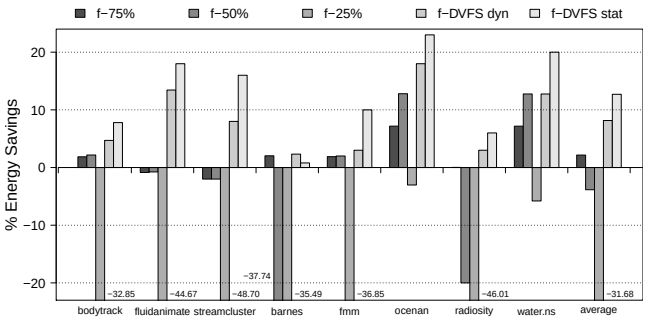

Figure 11: ***Energy savings (relative to the baseline fixed scheme $f_{100\%}$)*: Reducing the frequency/voltage of NoC can result in overall energy savings.**

As the NoC frequency decreases, the power consumed in the NoC is reduced and the overall on-chip energy savings are expected to grow. On the other hand, when NoC operates in lower frequencies, more time is needed for the application to complete execution, making unclear whether the benefits gained by the low-power NoC will result in system-wide energy savings at run completion. For example, the results show that when the NoC operates at $f_{25\%}$, the slowdown is large enough to adverse the NoC power savings into significant sytem energy loss. The same effect is also observed occasionally for other sub-optimal fixed frequencies. On average, all schemes exept $f_{25\%}$ show energy savings at the cost of performance.

Our results indicate the effectiveness of both, static and dynamic epoch-based DVFS schemes, in adapting the system into a more efficient state. In all cases, both adaptive schemes show energy reductions for the least performance degradation compared with the fixed schemes. The static off-line scheme achieves about 13.1% energy savings on average, with 2% slowdown. The savings achieved by the dynamic on-line scheme for roughly the same average slowdown are about 7.8%, which is around 60% of what the off-line achieves. In general, the dynamic scheme follows closely and consistently the effectiveness of the static scheme, indicating the strength of the dynamic approach in capturing correctly the changes in program behavior. In the case of barnes and radiosity, where a single epoch is dominating the execution, a wrong voltage/frequency adaptation decision for that epoch could significantly impact the results. As figure 13 implies, both approaches reach the same decision and successfully pick the optimal frequency.

Overall, our evaluation shows that the BarrierWatch approach is robust in detecting phase changes and adapting an epoch-aware system effectively. The lightweight detection of a phase change using epoch boundaries is limited, however, to the epoch granularity. Further techniques are required if changes within large multi-phase epochs need to be handled. Apart from accurately detecting behavior shifts, the effectiveness of an epoch-based adaptation strategy depends on the accuracy of the decision signatures and the underlying configurable hardware.

7. RELATED WORK

The concept of epoch as a code region was also introduced by Choi et al. [34] to guide a compiler directed cache co-

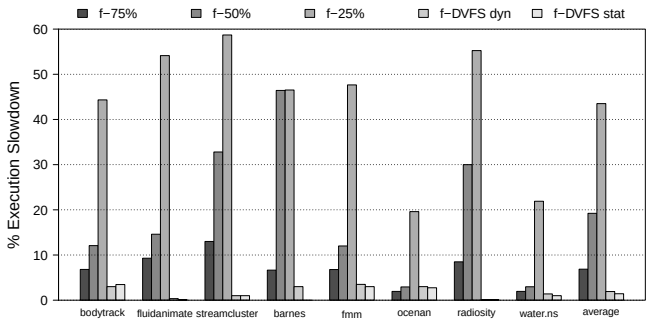

Figure 12: ***Execution slowdown (relative to the baseline*** $f_{100\%}$***)*: Reducing the frequency of NoC can decrease the overall performance.**

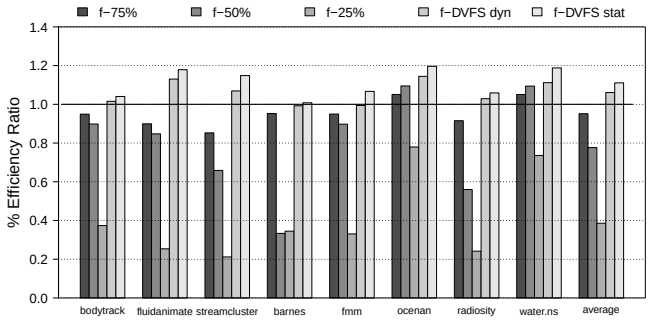

Figure 13: ***Energy-Delay improvement*: The ED improvement is the product of energy and delay, normalized to the baseline** $f_{100\%}$**. The higher the ratio is, the more efficient a given scheme is.**

herence scheme. According to their definition, epochs do not have a static ID and therefore are not reoccuring. Instead, they represent the dynamic sequence of parallel loops and serial sections of a program's execution. This allows them to use epochs for exploiting temporal and spatial locality across task boundaries by detecting accesses to the same data during different epochs. In our work, we use the notion of epochs to characterize the varying program behavior rather than data between consequtive tasks, representing reoccuring program phases with predictable behavioral patterns.

Prior work has focused much less on characterizing and detecting the time-varying behavior of multithreaded workloads than single-threaded ones. Some researchers have recently studied the potential of extending previously established phase detection techniques, primarily developed for single-threaded workloads, to parallel ones [12, 17]. They show how the existing profile-based phase classification methods are inaccurate in the domain of parallel applications and accordingly propose a new set of metrics that can be used to improve the phase-based simulation methodologies.

Single threaded program behavior has been studied and exploited using both, region- and interval-based methods. Code-region methods split the application's code into sections that corresspond to possibly different program phases and ensure that those sections can be effectively tracekd and characterized at run-time. Balasubramonian et al. [3], and Huang et al. [8] propose subroutines as a region granularity in which program phases can be tracked effectively for the purpose of dynamic adaptive optimization. They use a hardware call stack to identify major program subroutines and look for program changes by comparing the program behavior across different subroutines. Magklis et al. [35] consider also loop nests within long-running subroutines as possible phases. However, they rely on a static profile-based method to select the appropriate subroutines, loops, and settings for reconficuration and they expose this information through binary rewritting. Liu et al. [11] use a similar profile-based approach to select representative subroutines and loops to speed up detailed simulation. Other more recent studies use profiling to identify other various locations in the code that indicate phase changes, and propose to incorporate this information into the program binary as software phase markers [10, 13, 15].

Interval methods, on the other hand, divide an execution into fixed-size instruction (or time) windows, characterize past intervals using some architectural or code-based metric, and predict future intervals using history information. Balasubramonian et al. [3] use hardware counters to measure miss rate and branch frequencies and they identify a behavioral change by comparing them with adjustable thresholds on an interval basis. Their method is used to guide dynamic cache reconfiguration to save energy. Dhodopkan and Smith [6] use "working sets signatures" to characterize each interval of execution and identify a phase change by calculating a signature distance between consecutive time intervals. To reduce re-optimization overhead, they store a configuration for every signature and reinstate it when intervals with the same signature occur. Sherwood et al. [16] represent program behavior using interval-based basic block vectors (BBVs) and employ off-line clustering to classify regions of program execution into phases. In a follow-up work, they extend their approach with a hardware support for dynamic phase detection [7]. Lau et al. [14] investigate other code-related metrics for capturing program phases, such as loop, procedure, and register usage vectors. Isci and Martonosi [4], and Duesterwald et al. [5] have thouroughly shown the potential of using hardware performance counters to dynamically detect and predict changes in program behavior. Intervals have been studied in various scales, e.g., from 100K [3] to 10M instructions [16]. Vandeputte and Eeckhout [33] present "phase complexity surfaces" as a method to systematically characterize a program's phase behavior in different time scales.

All above work studies the behavior of single-threaded programs. Our focus in this paper is a simple effective approach to characterizing multithreaded workloads.

8. CONCLUSIONS

In this paper we have presented a characterization study of multithreaded workload behavior across and within program-defined epochs. Being a globally defined code section across all running threads, epochs are a natural granularity for examining the time-varying behavior of multithreaded workloads. Our analysis reveals that epochs repeat with strong behavioral similarity, while their boundaries indicate a significant behavioral change. Based on this observation, we propose BarrierWatch as an effective and lightweight technique for dynamically detecting and predicting changes in program behavior with the epoch granularity. Desirable properties of our approach include: being independent from

the underlying architecture, requiring no monitoring and sampling of fine-grained fixed execution intervals, naturally adopting variable-length intervals, being amenable for low-cost implementation and deployment, and finally being applicable to many multithreaded workloads written with barrier synchronizations.

Detecting changes in program behavior is essential for adaptive program optimization, both static and dynamic. BarrierWatch provides a simple elegant approach to capturing the varying behavior of multithreaded programs. We anticipate that our approach will open up new opportunities for *up-to-bottom* run-time optimizations and effective resource management in future CMPs.

9. REFERENCES

[1] Hennessy, J. L., and Patterson, D. A. *Computer Architecture A Quantitative Approach*, Elsevier, 2007.

[2] Solihin, Y. *Fundamentals of Parallel Computer Arch.*, 2009.

[3] Balasubramonian, R., et al. "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures", *MICRO*, 2000.

[4] Isci, C., and Martonosi, M. "Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data", *MICRO*, 2003.

[5] Duesterwald, E. et al. "Characterizing and Predicting Program Behavior and its Variability", *PACT*, 2003.

[6] Dhodapkar, A. S., and Smith, J. E. "Managing multi-config. hardware via dyn. working set analysis", *ISCA*, 2002.

[7] Sherwood, T. et al. "Phase tracking and prediction", *ISCA*, 2003.

[8] Huang, M. C. et al. "Positional adaptation of processors: application to energy reduction", *ISCA*, 2003.

[9] Eeckhout, L. et al. "Workload design: Selecting representative program-input pairs", *PACT*, 2002.

[10] Lau, J. et al. "Selecting Software Phase Markers with Code Structure Analysis", *CGO* 2006.

[11] Liu, W. and Huang, C. "EXPERT: expedited simulation exploiting program behavior repetition", *ICS*, 2004.

[12] Perelman, E. et al. "Detecting phases in parallel applications on shared memory architectures", *IPDPS*, 2006.

[13] Shen, X. et al. "Locality phase prediction", *ASPLOS*, 2004.

[14] Lau, J. et al. "Structures for phase class.", *ISPASS*, 2004.

[15] Ratanaworabhan, P. and Burtscher, M. "Program Phase Detection based on Critical Basic Block Transitions", *ISPASS*, 2008.

[16] Sherwood, T. et al. "Automatically characterizing large scale program behavior", *ASPLOS*, 2002.

[17] Zhang, Y. et al. "Analyzing the impact of on-chip network traffic on program phases for CMPs", *ISPASS*, 2009.

[18] Bienia, C. et al. "The PARSEC Benchmark Suite: Characterization and Arch. Implications", *Princeton Univ. TR*, 2008.

[19] Woo, S. C. et al. "The SPLASH-2 Programs: Characterization and Method. Considerations", *ISCA* 1995.

[20] Magnusson, S. et al., "Simics: A Full System Simulation Platform", *IEEE Computer*, 2002.

[21] Cho, S., and Jin, L. "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation", *MICRO*, 2006.

[22] Zhang, M., and Asanović, K. "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled CMPs", *ISCA*, 2005.

[23] Tilera TILE64 Processor. `http://www.tilera.com`.

[24] Howard J. et al. "A 48-Core IA-32 Message Passing Processor with DVFS in 45nm CMOS", *ISSCC*, 2010.

[25] Laudon, J., and Lenoski, D. "The SGI Origin: A ccNUMA Highly Scalable Server", *ISCA*, 1997.

[26] David C. Howell. *Fundamental Statistics for the Behavioral Sciences*, 4th Ed., Duxbury Resource Center, 1998.

[27] Lee, J. K. F. and Smith, A. J. "Branch prediction strategies and branch target buffer design", *IEEE Computer*, 1984.

[28] Vandeputte, F. et al. "A Detailed Study on Phase Predictors", *Euro-Par*, 2005.

[29] Mishra, A. et al., "A case for dynamic frequency tuning in on-chip networks", *MICRO*, 2009.

[30] Herbert, S. and Marculescu, D., "Variation-aware dynamic voltage/frequency scaling", *HPCA*, 2009.

[31] Kim, N. S. et al,, "Leakage current: Moore's law meets static power", *IEEE Computer*, 2003.

[32] Kim, J. S. et al,, "Energy characterization of a tiled arch. processor with on-chip networks", *ISLPED*, 2003.

[33] Vandeputte, F. and Eeckhout, L., "Phase Complexity Surfaces: Characterizing Time-Varying Program Behavior", *HiPEAC*, 2008.

[34] Choi, L. and Yew, P., "A compiler-directed cache coherence scheme with improved intertask locality", *SC*, 1994.

[35] Magklis, G., Scott, M., Semeraro, G., Albonesi, D., Dropsho, S., "Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor", ISCA, 2003,