PhLock: A Cache Energy Saving Technique Using Phase-Based Cache Locking

Tosiron Adegbija, *Member, IEEE*, and Ann Gordon-Ross, *Member, IEEE*

***Abstract*— Caches are commonly used to bridge the processor-memory performance gap in embedded systems. Since embedded systems typically have stringent design constraints imposed by physical size, battery capacity, and real-time deadlines much research focuses on cache optimizations, such as improved performance and/or reduced energy consumption. Cache locking is a popular cache optimization that loads and retains/locks selected memory contents from an executing application into the cache to increase the cache's predictability. Previous work has shown that cache locking also has the potential to improve cache energy consumption. In this paper, we introduce phase-based cache locking, *PhLock*, which leverages an application's varying runtime characteristics to dynamically select the locked memory contents to optimize cache energy consumption. Using a variety of applications from the SPEC2006 and MiBench benchmark suites, experimental results show that *PhLock* is promising for reducing both the instruction and data caches' energy consumption. As compared to a nonlocking cache, *PhLock* reduced the instruction and data cache energy consumption by an average of 5% and 39%, respectively, for SPEC2006 applications, and by 75% and 14%, respectively, for MiBench benchmarks.**

***Index Terms*— Adaptable computing, cache locking, configurable caches, dynamic optimization, energy savings, low-power embedded systems, persistent phases, phase-based tuning.**

I. Introduction and Motivation

CACHES are commonly used in embedded systems to bridge the processor-memory performance gap by exploiting the spatial and temporal locality of memory accesses. However, caches can contribute significantly to overall system energy consumption, especially in resource-constrained embedded systems. As the cache's actual power consumption varies in different systems, the cache has been estimated to consume from 16% to 50% of the total chip power in a variety of processors [21], [22]. Therefore, much research focuses on cache optimizations that minimize the energy consumption—without degrading the performance—and satisfy the system's design constraints. Embedded systems are especially viable candidates for these optimizations, since they have intrinsic design constraints imposed by physical size, battery capacity, real-time deadlines, and consumer market competition.

Cache locking is a popular cache optimization that loads and retains/locks selected memory blocks (regions of instruction and/or data addresses) from an executing application into the cache. Cache locking can be done either at system startup (static cache locking) or dynamically during runtime (dynamic cache locking), and is available in modern embedded processors, such as the ARM Cortex A* series processors [1]. These cores support special lock subroutines that lock the selected contents into the cache such that locked contents cannot be evicted by the cache's replacement policy. Since accesses to locked contents will always produce a cache hit, these addresses' access times are predictable.

Cache locking research has traditionally focused on improving the cache's execution time predictability, especially in real-time systems where the worst case execution time (WCET) must be estimated [29]. In real-time systems, the cache contents are typically known statically and cache locking ensures that the memory access times and cache-related preemption delays are predictable for the locked contents, allowing tighter WCET estimation. Previous work [16] showed that cache locking benefits also include improved cache performance in general-purpose embedded systems by eliminating conflict misses and guaranteeing a hit for the locked contents. Additionally, cache locking can reduce dynamic energy since cache locking can reduce cache misses, and thus reduce the energy consumed from accessing lower memory levels and associated stalls.

However, cache locking also reduces the cache's overall utilization. Since portions of the cache are exclusively used for the locked contents, the effective cache capacity is reduced and conflict misses may increase for the memory blocks that are not locked. For cache locking to be effective, the locked contents must represent application regions that significantly affect overall cache performance and energy consumption. If the contents are poorly selected, cache locking can significantly degrade the performance [34] and/or energy.

Anand and Barua [5], Liang and Mitra [16], and Liu *et al.* [18] used cache locking to optimize instruction cache performance in general-purpose embedded systems. However, none of these works evaluated the energy

Manuscript received June 2, 2017; revised August 23, 2017; accepted September 25, 2017. This work was supported in part by the National Science Foundation under Grant CNS-0953447. *(Corresponding author: Tosiron Adegbija.)*

T. Adegbija is with the Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721 USA (e-mail: tosiron@email.arizona.edu).

A. Gordon-Ross is with the Center for High Performance Reconfigurable Computing, University of Florida, Gainesville, FL 32611 USA (e-mail: anngordonross@ufl.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2017.2757477

benefits of cache locking. Additionally, most current cache locking techniques target one cache—either the instruction cache or the data cache—and involve techniques specific to the targeted cache's access characteristics. A technique developed specifically for the instruction cache, for example, would be ineffective for the data cache.

Instruction and data caches exhibit different access characteristics during runtime. Even though applications typically have phases of execution during which the execution characteristics [e.g., cache miss rates (CMRs), branch mispredicts, and instructions per cycle (IPC)] are relatively stable, the instruction and data characteristics of these phases vary. The pattern of instruction variability remains stable throughout the execution, since instructions remain fixed during execution. Data streams, on the other hand, may vary during runtime, changing the application's execution characteristics. In addition, since an application typically processes much more data than the number of instructions executed, an instruction cache locking technique, if applied directly to the data cache, would require a large data cache and/or potentially result in runtime overhead in terms of performance and/or energy. These overheads result from the complex runtime analysis required due to the inherent runtime variability of data caching [34]. Thus, our goal is a dynamic cache locking technique that is low overhead and can be used for both instruction and data caches in general-purpose embedded systems.

In this paper, we propose a new method for using cache locking to achieve energy savings, with minimal performance degradations, in general-purpose embedded systems. We propose *PhLock*[1]—*Phase-based Cache Lock*ing as a low-overhead dynamic cache locking technique for both the instruction and data caches. *PhLock* is motivated by the observation that even though applications typically have several phases during runtime, a few of the phases are usually persistent. Persistent phases feature instructions that are frequently executed or data blocks with high reuse.

PhLock leverages application phase changes and data reuse to dynamically determine and change the locked contents at runtime. *PhLock* is based on the premise that cache energy consumption can be optimized if persistent memory blocks—high-reuse instructions and data blocks—are locked in the cache, guaranteeing that all accesses to those blocks are cache hits. *PhLock* locks those persistent phases' instructions or data in the cache, thereby eliminating the conflict misses for those phases. Using *PhLock*, the locked cache contents are dynamically selected, loaded, and retained at runtime based on the application's intrinsic runtime variable characteristics.

Although the key goal of *PhLock* is to reduce the cache's energy consumption, especially in general-purpose embedded systems, *PhLock* also improves the cache's performance. Our contributions are summarized as follows.

1) We propose *PhLock*, which dynamically determines the locked contents based on applications' persistent phases.
2) *PhLock* reduces the energy consumption, with minimal performance degradation, for both the instruction and data caches as compared to a nonlocking cache. *PhLock* also dynamically determines the benefits of cache locking and only locks the cache when such benefit exists.
3) Using *PhLock*, we analyze the benefits of cache locking in instruction and data caches for improving cache energy consumption and performance, and show that these benefits are dependent on the kind of applications being executed.

[1]Pronounced "flock"

To illustrate *PhLock*'s benefits in different execution scenarios and for a variety of applications, we perform experiments using SimpleScalar simulator [10] and benchmarks from the SPEC CPU2006 [3] and MiBench [14] suites. Results reveal that as compared to a nonlocking cache, *PhLock* reduces the average instruction and data cache energy by 5% and 39%, respectively, for SPEC2006 applications, and by 75% and 14%, respectively, for MiBench benchmarks.

II. Background and Related Work

Much prior work has studied the WCET estimation benefits of cache locking. In addition, much work has studied phase classification for exploiting an application's runtime variability. Our cache locking technique, *PhLock*, uses application phases and the phases' persistence to determine the locked contents during runtime in order to dynamically optimize the cache's energy without degrading the performance. Additionally, our technique caters to general-purpose embedded systems, where the applications are unknown *a priori* and typically execute multiple times throughout the system's lifetime. In this section, we present a general related work and background on cache locking and phase classification, which we leverage for dynamically selecting the locked contents.

A. Cache Locking

Cache locking was primarily developed for hard real-time systems. Cache locking can be full [7], [18], [35] or partial [12], [19], [25]. In full cache locking, the whole cache is locked, such that all accesses to unlocked cache blocks result in misses. Even though full cache locking may be beneficial for higher predictability when the applications are known *a priori*, the performance can be significantly degraded due to increased cache misses. Alternatively, partial cache locking only locks a fraction of the cache; the unlocked portion of the cache works with normal cache replacement enabled. In this paper, we employ partial cache locking, since it has performance benefits as compared with full cache locking.

The chief goal of cache locking is to improve the cache's predictability and to facilitate tighter WCET estimations as compared to a system with a nonlocking cache [23]. Since cache locking reduces the effective cache utilization, one of the major challenges of effective cache locking is determining the locked contents.

In general, cache locking can be broadly classified, with respect to how and when the locked contents are determined and/or changed: static cache locking [13] and dynamic cache locking [26]. In static cache locking, application characteristics are statically analyzed *a priori*, the locked contents are determined and remain constant throughout the execution.

Although static cache locking works well when application characteristics are known at design time [27], these characteristics may change during execution due to changing execution conditions or input data. Assuming *a priori* knowledge of applications makes these methods' inapplicable to general-purpose embedded systems (e.g., smartphones, tablets, and so on), which typically execute a large variety of applications that are unknown at design time. In such systems, static cache locking can significantly degrade the overall performance, energy, and even the WCET estimation.

Puaut and Decotigny [27] proposed greedy algorithms for selecting the locked contents in hard real-time systems. Vera *et al.* [34] combined compile-time cache analysis with data cache locking to enable tight WCET estimation in real-time systems. Since these methods targeted real-time embedded systems where the executing applications are typically known *a priori*, and required design-time analysis of the applications, the proposed methods are inapplicable to general-purpose embedded systems. Furthermore, even though these methods improved cache predictability, they did not explicitly focus on improving the cache's energy; they could potentially degrade the cache's energy consumption as a result of increased conflict misses and lower level memory accesses for the unlocked blocks [34].

In contrast, dynamic cache locking [12], [26], [34] selects the locked contents during runtime. In some cases (see [6], [12]), the locked contents are determined statically, and locked at runtime to account for variable execution conditions. This technique adjusts the locked contents at runtime in order to further improve cache predictability and reduce dependence on *a priori* application analysis and the prior knowledge of cache content. In addition, dynamic cache locking can account for changes in working set size and enable better utilization than static cache locking [23]. Thus, dynamic cache locking is especially useful for general-purpose systems where the applications are unknown *a priori*.

Apart from improving the real-time systems' cache predictability, cache locking can also benefit general-purpose systems for improving the cache's energy consumption without much performance degradation. To improve the cache performance, Liang and Mitra [16] and Liang *et al.* [17] presented an instruction cache locking heuristic to select the locked contents in order to realize the performance benefits of cache locking by reducing the conflict misses. The proposed heuristic reduced the CMRs by up to 24%. Similarly, to improve the instruction cache performance, Anand and Barua [6] proposed a cache locking technique that first analyzed the program code to determine potential program regions for locking, and then used a heuristic to select the appropriate locked contents during runtime, based on changes in program locality.

Anand and Barua [5] used detailed, iterative cache simulations to evaluate the performance benefits for locking different memory blocks. However, due to the detailed cache simulations and number of iterations involved, this method would incur significant runtime overhead if used for dynamic cache locking. Additionally, since the authors used static cache locking, this method is not applicable to systems where the executing applications are unknown *a priori*. Liu *et al.* [18] proposed an algorithm that dynamically determined the instruction cache's locked contents to improve the average-case execution time. However, these works did not evaluate the energy benefits of cache locking, and since these works focused on the instruction cache, the inherent runtime variability of data caches were not considered.

Using simulations, Asaduzzaman *et al.* [7] showed that cache locking could potentially improve cache performance and reduce power consumption. Asaduzzaman *et al.* [8] presented a cache locking technique that locked cache lines that caused the highest number of misses in order to improve the cache's performance. Kang *et al.* [33] used a dynamic programming algorithm to predetermine the locked contents in order to improve the cache's power consumption and performance. However, since the locked contents are predetermined, the proposed method is constrained to systems where the executing applications are known *a priori*.

Our work differs from the previous cache locking methods by using dynamic cache locking in both the instruction and data caches to optimize the cache's energy consumption, without degrading the performance. We propose *PhLock*, a phase-based methodology, that dynamically selects the locked contents, incurs minimal runtime overhead, and makes our work applicable to general-purpose embedded systems, where the executing applications may be unknown *a priori*.

B. Phase Classification

Since dynamically leveraging phase characteristics can significantly increase optimization potential by specializing the optimizations to different phases of execution [15], [32], much prior work explored different phase classification techniques. Sherwood *et al.* [32] showed that phase classification using basic block distribution was highly correlated with application characteristics, such as CMRs, IPC, and branch mispredictions. Hamerly *et al.* [15] created SimPoint, which used machine-learning techniques to identify an application's phases by analyzing basic block vectors that were annotated with the block's execution frequency. Shen *et al.* [30] showed a strong correlation between data locality and an application's phase characteristics, and showed that data reuse patterns could be used to classify phases. Since phase characteristics are strongly correlated with the phases' data reuse patterns, our work leverages phase classification and the phases' data reuse to select an application's locked contents to optimize the data cache's performance and energy consumption.

III. Phase-Based Cache Locking

Fig. 1 presents an overview of our phase-based cache locking methodology, *PhLock*, which selects the locked contents such that the cache's energy consumption are improved compared to a default nonlocking cache. When an application A_i is first executed, using the unlocked cache, the application's phases are classified to determine the phases' persistence, CMRs, and base energy consumption (Section III-C). The *PhLock* algorithm (Section III-E) then determines if the application will benefit from cache locking based on the most persistent phases or phases with the highest CMRs.

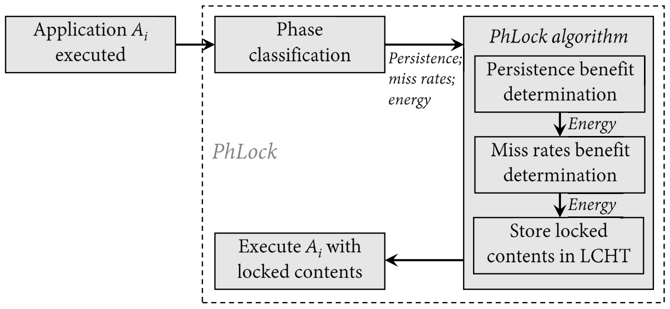

Application A_i executed
Phase classification
Persistence; miss rates; energy
PhLock algorithm
Persistence benefit determination
Energy
Miss rates benefit determination
Energy
Store locked contents in LCHT
PhLock
Execute A_i with locked contents

Fig. 1. High-level overview of *PhLock*'s workflow.

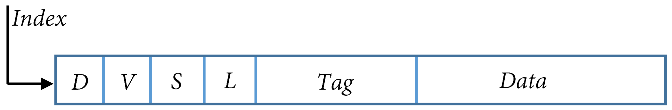


Fig. 2. Cache line with a locking bit. D = dirtybit; V = validbit; S = sharingbit;L = locking bit.

These two options are compared to the nonlocking cache, so that *PhLock* never degrades the energy consumption compared to the nonlocking cache. The locked contents that achieve lowest energy consumption are then stored in a locked contents history table—LCHT—for subsequent executions of A_i (Section III-B).

In this section, we describe the details of *PhLock*: the architecture and implementation, an analysis of persistent phases that motivates our work, our methodology for selecting the locked contents based on phases' persistence, and the *PhLock* algorithm.

A. PhLock Architecture and Implementation

Cache locking can be implemented as line locking [12] or way locking [7]. Line locking enables individual lines to be locked for different cache sets, as opposed to way locking, where all the lines in a particular cache way are locked. Line locking is more widely implemented, and can easily be implemented with low overhead; an additional locking bit is added to the cache block to indicate whether or not the block is locked. Thus, the cache's replacement algorithm skips any blocks with the lock bit set. In this paper, we assume line locking, which is supported in some members of the ARM processor family. Fig. 2 illustrates a cache line with a dirty bit D (we assume a write-back cache), a valid bit V, a sharing bit S, and a locking bit L.

Fig. 3 depicts a high-level overview of the *PhLock* architecture for a sample dual-core system, where each core has private level one (L1) instruction and data caches. The *PhLock* module connects directly to each core's L1 instruction and data cache; thus this architecture can be extended to any n-core system by connecting the locking module to each core's L1 caches. Depending on the system, there could be a dedicated *PhLock* module for each core—at the expense of area and power overheads—or a single global *PhLock* module—at the expense of time overhead, especially in a system with several cores.

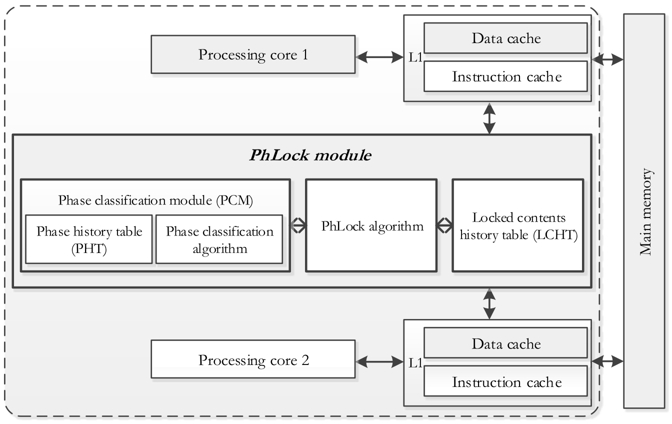


Fig. 3. High-level overview of the *PhLock* architecture.

The *PhLock* module contains a phase classification module (PCM), comprising of the phase classification algorithm and a phase history table (PHT) (Section III-C). The phase classification algorithm classifies the applications' phases and determines the phases' persistence, which are then stored in the PHT. The *PhLock* module also contains the *PhLock* algorithm, which determines the locked contents, and an *LCHT*, which stores the locked contents.

The *PhLock* module can be implemented in software, hardware, or a combination of both. If implemented in software, *PhLock* can use the system's processor to execute phase classification and the *PhLock* algorithm, while the LCHT is stored in SRAM. However, a software implementation may affect the functional applications' caches and runtime behavior due to context switching. These effects could degrade *PhLock*'s optimization potential. To mitigate the software implementation overheads, while maintaining ease of implementation, we propose a combination of software and hardware, wherein low-overhead hardware structures implement the PHT and LCHT, while the rest of the *PhLock* module—phase classification algorithm and *PhLock* algorithm—can be implemented in software.

B. Locked Contents History Table

The LCHT is a small data structure with per-application entries that retain information (memory addresses) of an application's locked phases' instructions and/or data for subsequent executions of that application. The size of LCHT can be dynamic or fixed depending on the memory constraints of the system, and a replacement policy, such as least recently used (LRU), can be used when the table is full. When a new application is executed, an entry is added to the LCHT for that application.

Fig. 4 depicts the LCHT entry's basic structure, which includes: application A_i's identification ID; the memory addresses of A_i's locked contents $lockedContents(A_i)$ as selected by the *PhLock* algorithm; $noLockedContents$ and $profile$ flags, which default to 0' and indicate if A_i benefits from cache locking and if A_i has been profiled, respectively; and two fields to store A_i's CMR and energy consumption

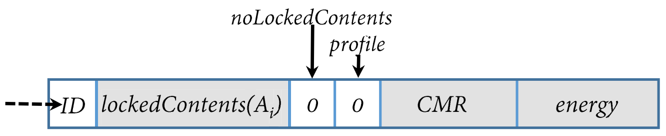


Fig. 4. LCHT entry basic structure.

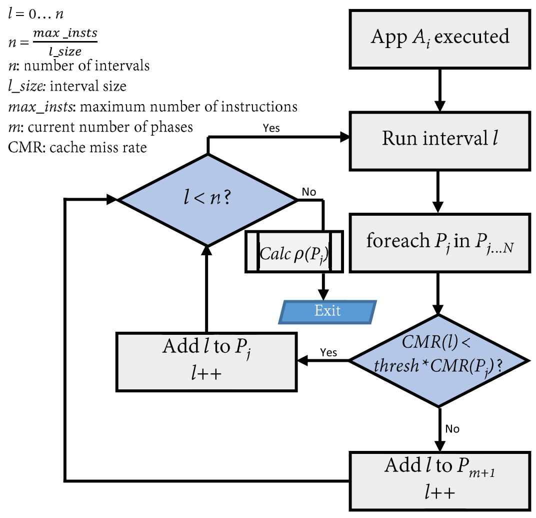


Fig. 5. Runtime phase classification algorithm.

while executing with a nonlocking cache for determining if A_i benefits from cache locking (Section III-E). The CMR is measured using the microprocessor's hardware performance counters, while the energy is calculated using the energy model in Fig. 7. We analyze the overhead of LCHT in Section IV.

C. Runtime Phase Classification

The PCM profiles the application with a nonlocking cache during the application's first execution, and uses (1) and (2) to determine the phases' persistence. For our work, we used CMRs as the execution characteristic for classifying the phases, since our optimization target is the cache. Prior work has shown that the CMRs are sufficient for such classifications [4], [9], [28].

Fig. 5 illustrates the phase classification algorithm implemented by the PCM. When a new application A_i is executed, the PCM partitions the application into intervals $l = 0 \ldots n$, each of size l_size instructions, a designer-specified variable. For our work, we empirically determined that $l\_size = 1\,000\,000$ instructions allowed for balance of fine-grained, low-overhead, and accurate phase classification. A smaller l_size would result in more unnecessary classifications, while a larger l_size may reduce the classification accuracy.

For each application, there are n intervals. The PCM compares each interval's CMRs [$CMR(l)$] with CMRs of phases stored in a *PHT*. The PHT is a data structure in the PCM that stores classified phases' program counters. If the interval's CMR is within a threshold ($thresh$, e.g., 5%) of any of the stored phases, the PCM adds that interval's head program counter to the PHT. Otherwise, the PCM creates a new PHT entry and designates the interval's head program counter as a new phase. The threshold provides a tradeoff between the number of phases and the accuracy of phase classification. A higher $thresh$ value would result in fewer phases, but phases with disparate characteristics may be classified as one phase, while a smaller $thresh$ value would result in more phases.

We assume that phase classification is performed without cache locking; thus, the base energy is also recorded for each application and stored in the PHT. When an application's phases have been classified, each phase's persistence, ρ is then calculated using 2 and stored in the PHT.

D. Persistent Phases

To select the locked contents, *PhLock* leverages application execution locality; the majority of an application's execution, measured by the number of dynamic instructions executed, typically occurs within a few persistent phases that access the same data. The key idea of *PhLock* is that cache lines that represent an application's most frequently referenced data and/or instructions are locked in the cache.

To ascertain the extent of application execution locality, we analyzed several applications in the SPEC2006 benchmark suite and their phases to evaluate the benefits of locking these phases' cache blocks. Our analysis revealed that for cache locking to provide cache locking benefits, a phase P_i's execution must comprise at least 10% of application A's total execution. Based on this observation, we define a phase P_i as persistent if

$$P_i \in \boldsymbol{A} : I_{Pi} \geq 0.1 \times I_{\text{total}} \tag{1}$$

where $\boldsymbol{A}$ represents all of the phases in application A, I_{Pi} is P_i's number of instructions, and I_{total} is A's total number of instructions. We quantify P_i's persistence ρ using the percentage of A's total execution that belongs to P_i, where ρ is given as

$$\rho = \frac{I_{Pi}}{I_{\text{total}}} \times 100\%. \tag{2}$$

We note that persistence is a necessary, but not sufficient condition for locking P_i to provide cache locking benefits.

Fig. 6 analyzes phase persistence for an arbitrary subset of five applications each from SPEC CPU2006 [3] and MiBench [14] benchmark suites. The x-axis depicts the applications' distinct phases (the number of phases per application varies) and the y-axis depicts the percentage of each application's execution that belongs to each phase (total phase execution percentage for each application totals 100%).

Fig. 6(a) shows that, for the SPEC2006 benchmarks, the majority of applications have a few phases that are significantly more persistent than the other phases, suggesting that these application's phases are amenable to cache locking. For example, 56% of $calculix$'s execution is spent in two phases, while the remaining 44% of the execution is spent in the remaining six phases—7% on average for each remaining phase with a 0.03 standard deviation. Fifty one percent of $gromacs$'s execution is spent in two phases, while the remaining 49% of the execution time is spent in the remaining 14 phases—3% on average for each remaining

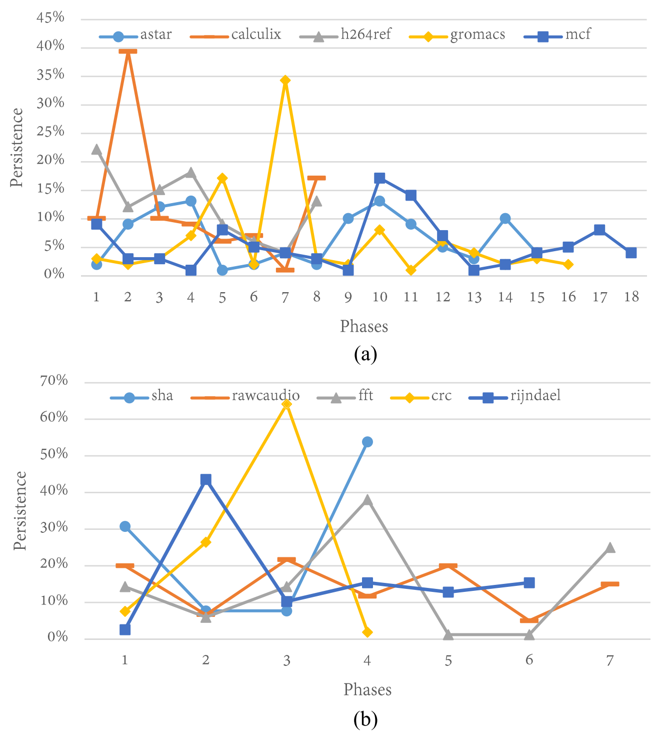


Fig. 6. Phase persistence for (a) SPEC2006 and (b) MiBench benchmarks.

phase with a 0.02 standard deviation. Since only two phases represent nearly half of *calculix*' and *gromacs*'s execution, these applications would benefit the most from cache locking if these two most persistent phases were locked in the cache. Alternatively, since *h264ref*'s execution is relatively evenly spread across all of the application's eight phases, *h264ref* has less potential to benefit from *PhLock* since no phase has a prominent persistence.

Fig. 6(b) shows a similar trend for MiBench. In general, MiBench workloads have fewer phases and less variation between the phases than SPEC2006. However, the MiBench applications also showed a few phases that were more persistent than others. For example, *sha*'s phases 1 and 4 comprise 31% and 54% of the total execution, while the remaining 16% of execution is evenly distributed between phases 2 and 3. Similarly, 44% of *rijndael*'s execution is spent in phase 2, with the remaining execution distributed among the other phases.

PhLock prioritizes the phases with the highest persistence for cache locking. From prior analysis, we also observed that while locking the most persistent phases achieved the most energy savings benefits, some phases benefited more from locking phases with the highest CMRs. Thus, *PhLock* contains two locking modes—locking based on phases' persistence and CMRs—and dynamically determines which mode achieves the maximum energy saving benefits. *PhLock* also identifies phases that will not benefit from locking, and executes those with the base nonlocking cache to prevent performance and/or energy degradation. Results in Section V verify these hypotheses regarding the proposed *PhLock* technique.

E. PhLock *Algorithm*

Without loss of generality and considering general-purpose embedded systems, we assume a system without preemption. However, *PhLock* can easily incorporate preemption by saving the profiling state of LCHT on application preemption, and restoring the profiling state on resumption.

Algorithm 1 depicts our *PhLock* algorithm. The algorithm uses two executions to determine which locking mode achieves maximum benefits: locking based on phase persistence or CMRs. The algorithm takes as input an array of application A_i's phases (at P_i) and the phases' persistence (ρ) and CMRs [$CMR(P_i)$], as determined by the PCM (Section III-C). The algorithm outputs an array of A_i's locked contents' memory addresses, $@lockedContents(Ai)$.

For each application A_i, if the LCHT contains no entry for A_i, *PhLock* indexes the application's phases in descending order of persistence and CMRs (lines 1–5), and selects phases for locking in descending order until the size of the selected memory blocks exceeds $maxLockedCache$ (lines 6–13). The size of the selected memory blocks is calculated by the number of unique 64-B blocks accessed by the selected phases. $maxLockedCache$ is the maximum percentage of the cache that can be locked, and defaults to 50%. We empirically determined that at least 50% of the cache must remain unlocked to minimize conflict misses for the memory blocks that are not locked for an application to benefit from cache locking. We found this to be truer for SPEC2006 applications than the MiBench applications since the SPEC2006 applications have much larger working set sizes than the MiBench applications. However, we kept the value at 50% to cater to the different application working set sizes.

On the A_i's first execution, *PhLock* locks phases based on the phases' persistence ($@lockedContents\_pers$), and stores the $energy$ achieved, $energy\_pers$ (lines 11–22). Similarly, on the second execution, *PhLock* locks phases based on the phases' CMRs ($@lockedContents\_miss$), and stores the $energy$ achieved, $energy\_miss$ (lines 23–27). If both $energy\_pers$ and $energy\_miss$ increase the $energy$ as compared to a nonlocking cache, *PhLock* determines that the cache locking does not benefit the application, and sets $noLockedContents\_Ai$ to 1 (lines 28–30). Otherwise, *PhLock* determines which locking mode achieves the maximum benefits, stores the locked contents ($@lockedContents\_pers$ or $@lockedContents\_miss$) indicating the best locking mode, and sets the $profile$ flag to 1 to indicate that A_i's locked contents have been determined (lines 31–37).

If the LCHT contains an entry for A_i's locked contents and $profile$ is set (i.e., profiling is complete for A_i), $loadAndLock()$ triggers the processor's cache locking subroutines (Section III-A), which load and lock A_i's locked contents in the cache for the duration of A_i's execution (lines 38–41). The $loadAndLock()$ function looks up the LCHT for the memory addresses of the locked contents, and sets the locking bit L for the cache lines corresponding to those addresses. Alternatively, if $noLockedContents$ is set, A_i is executed with the nonlocking cache (lines 42–44).

Algorithm 1 *PhLock* Algorithm

```
Input: $@P_i$, $\rho$, $CMR(P_i)$
Output: @lockContents(Ai)
if !(lockedContents(Ai)) in LCHT then
    #index the phases in descending order of persistence
    @lock_phases_pers = sort $\{\rho\{\$b\} \Leftrightarrow \rho\{\$a\}\}$ $@P_i$;
    #index the phases in descending order of cache miss rates
    @lock_phases_miss = sort $\{CMR(P_i)\{\$b\} \Leftrightarrow CMR(P_i)\{\$a\}\}$ $@P_i$;
    for all $P_i$(@lock_phases_pers) do
        while sizeOf(lockedContents_pers(Ai)) $\leq$ sizeOf(maxLockedCache) do
            push(@lockedContents_pers(Ai), $P_i$);
        end while
    end for
    for all $P_i$(@lock_phases_miss) do
        while sizeOf(lockedContents_miss(Ai)) $\leq$ sizeOf(maxLockedCache) do
            push(@lockedContents_miss(Ai), $P_i$);
        end while
    end for
    execution_Ai = 1;
else if lockedContents(Ai) in LCHT && execution_Ai = 1 then
    #On first execution, lock phases based on persistence
    loadAndLock(@lockedContents_pers);
    execute(Ai);
    execution_Ai = 2;
    $energy\_pers = energy(locking)$
else if lockedContents(Ai) in LCHT && execution_Ai = 2 then
    #On second execution, lock phases based on cache miss rates
    loadAndLock(@lockedContents_miss);
    execute(Ai);
    $energy\_miss = energy(locking)$
    if $energy\_pers > energy(non-locking)$ && $energy\_miss > energy$(non-locking) then
        #No locking benefits
        noLockedContents_Ai = 1;
    else if $energy\_pers \leq energy\_miss$ then
        @lockedContents(Ai) = @lockedContents_pers
    else if $energy\_pers > energy\_miss$ then
        @lockedContents(Ai) = @lockedContents_miss
    end if
    storeLockedContentsLCHT(Ai);
    profile = 1;
else if lockedContents(Ai) in LCHT && profile = 1 then
    #locked contents have been determined
    loadAndLock(@lockedContents(Ai));
    execute(Ai);
else if noLockedContents_Ai = 1 then
    #Ai has no locked contents
    break;
end if
```

IV. Computational Complexity and Hardware Overheads

The *PhLock* algorithm sorts the N phases' persistence and CMRs with worst case time complexity $O(NlogN)$, and selects the locked contents with worst case time complexity $O(N)$. Given that these operations dominate the algorithm, the algorithm results in minimal computational overhead and has good scalability.

PhLock's hardware overheads comprise mainly of the area and power overheads from the LCHT (Section III-A). *PhLock* also includes a PHT as part of the PCM (Section III-C). The structure of PHT can be easily implemented as a software- or hardware-based lookup table. The table's size depends on the total number of distinct phases across all applications running on the system. Much prior work [31], [32] has studied the PHT's structure and shown that these structures have minimal impact on overall system area, performance, or energy consumption. *PhLock* will not increase these overheads with respect to the prior work.

We note that the LCHT can also be implemented in software for easy system integration. However, a software implementation can adversely impact the application's cache and runtime behavior due to context switches. Thus, we detail a hardware implementation of the LCHT.

To adhere to system memory constraints, the size of LCHT can be fixed, and a replacement policy, such as LRU, can be used when the table is full. To show that *PhLock* constitutes minimal hardware area and power overhead, we estimate the LCHT's hardware/memory requirements for a 32-entry LCHT. In this LCHT, 5 b store the ID, 32 b store $lockedContents(Ai)$, 1 b each stores the $noLockedContents$ and $profile$ flags, and 16 b each store the CMR and energy. Using these assumptions, we estimate from a synthesizable VHDL implementation and synthesis using Synopsys Design Compiler [11] that the 32-entry LCHT constitutes an area of 2.48 μm^2 and the power consumption of 56.72 μW. Relative to a MIPS32 M14K [2] 90-nm processor, which has an area of 0.21 mm^2 and consumes 12 mW of power at 200 MHz, the 32-entry LCHT constitutes only 1.3% and 0.5% area and power overheads, respectively.

V. Experimental Results

A. Experimental Setup

We quantified *PhLock*'s energy savings and performance improvement using 20 benchmarks in total: 15 from the SPEC CPU2006 benchmark suite, and executed using the reference input sets, and 5 from the MiBench suite. All benchmarks were compiled to Alpha/OSF binaries. In order to rigorously test our methodology, we used SPEC benchmarks to represent applications with high compute and memory intensity, and significant runtime execution variability (i.e., more distinct phases); we used MiBench to represent embedded systems applications, which typically have fewer phases and less interphase variability. We ran the MiBench benchmarks to completion, and used the first 10 billion instructions from each SPEC benchmark. We used SimPoint [15], with an interval size of 1 million instructions, to classify the benchmarks' phases and calculated the phases' persistence using 2.

We modeled cache locking using Simplescalar-AlphaLinux's sim-outorder. We modeled an embedded system microprocessor with base cache configurations similar to the ARM Cortex A15 [1] microprocessor with 32 KB, 4-way set associative private L1 instruction and data caches with 64-B line sizes. We used sim-profile to collect information about the phases memory accesses and data reuse. We interfaced SimpleScalar with a software implementation of the *PhLock* algorithm using Perl.

Fig. 7 depicts the energy model used to calculate the L1 caches' energy consumption. The model calculates the cache's dynamic and static energy, the energy required to fill the cache on a miss, the energy consumed during a cache write back, and the energy consumed when the processor is stalled during cache fills and write backs. We assumed instruction and data cache access latencies of 1 cycle and a main memory access latency of 80 cycles [20]. We used Simplescalar to gather cache statistics, such as *totalMisses, totalAccesses*, and *totalWritebacks*. We assumed the static energy per cycle to be 25% of the cache's dynamic energy and the CPU idle energy to be 25% of the MIPS M14K processor's active energy [2]. We used CACTI [24] to determine the cache's dynamic energy for 90 nm technology.

```
totalEnergy = dynamicEnergy + staticEnergy +fillEnergy +
            writebackEnergy + cpuStallEnergy;
dynamicEnergy = totalAccesses * accessEnergy;
staticEnergy = (((totalMisses * penalty) +
            (totalHits * hitCycles)) * staticEnergyPerCycle
staticEnergyPerCycle = dynamicEnergy * 0.25;
fillEnergy = (totalMisses * (linesize/wordsize) *
            readEnergyPerWord);
writebackEnergy = (totalWritebacks * (linesize/wordsize) *
            writeEnergyPerWord);
cpuStallEnergy = (((totalMisses * penalty) + (totalWritebacks *
            writebackPenalty)) * cpuIdleEnergy);
```

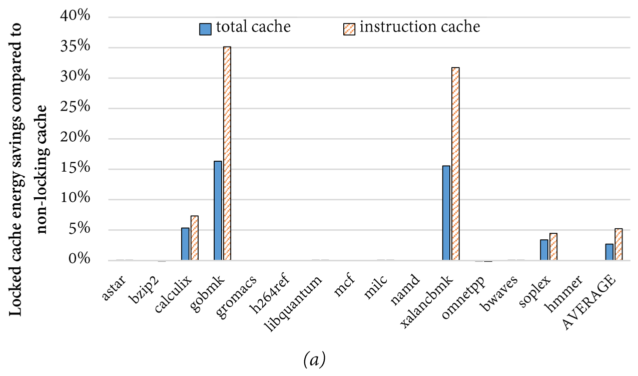
Fig. 7. Energy model.

B. Energy Savings for Instruction and Data Caches

1) SPEC2006: Fig. 8 depicts the instruction and data caches' energy savings achieved by *PhLock* for the SPEC2006 benchmarks as compared to a nonlocking cache. We also evaluated the overall cache impact of locking the instruction and data caches individually. The results depict the system after *PhLock* has selected the locked contents and evaluated the cache locking benefits for the applications (i.e., after the first two executions have completed). Note that these first two executions do not constitute any overhead, since both executions achieve overall energy savings as compared to the nonlocking cache. *PhLock*'s goal, however, is to determine the best cache locking mode—based on phases' persistence or CMRs—that achieves maximum benefits.

Fig. 8(a) shows that on average over all the SPEC applications, *PhLock* reduced the instruction cache and total cache (instruction + data) energy consumption by a modest 5% and 3%, respectively, with savings as high as 35% and 16% for $gobmk$. The modest average overall energy savings resulted from the fact that *PhLock* determined that cache locking offered no benefits for most of the applications; thus, the applications were executed using the nonlocking cache. Most of the applications have an instruction footprint that fits into the unlocked portion of the cache, while a few applications (e.g., mcf and $gromacs$) exhibit extremely low instruction reuse that made a locked cache unviable for efficient execution.

With the SPEC2006 applications, *PhLock* showed a lot more promise for the data cache than the instruction cache. This observation was due to the applications' relatively large working set sizes and the high rate of data reuse. Fig. 8(b) shows that on average over all the applications, *PhLock* reduced the total and data cache energy consumption by 20% and 39%, respectively, with data cache energy savings as high as 79% for $soplex$. *PhLock* had the least total and data cache energy savings for $bzip2$ at 3% and 5%, respectively, due to



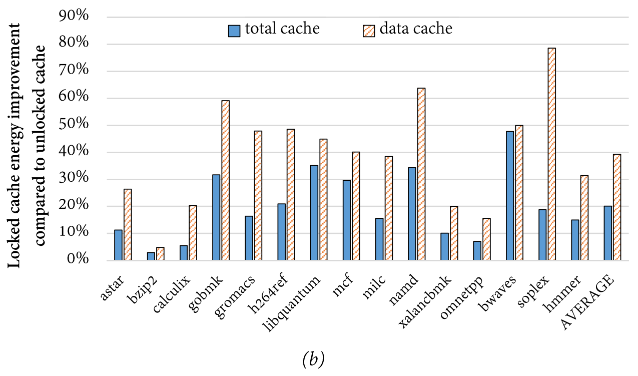


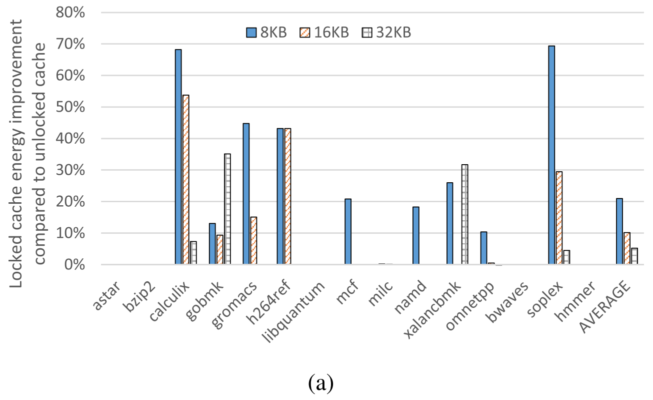
Fig. 8. *PhLock* energy improvement compared to the nonlocking cache for (a) instruction cache and (b) data cache, for SPEC2006 applications.

*bzip*2's small working set size, which allowed most of the data to fit into the locked cache's unlocked portion.

We also observed a difference in how data cache locking affected the total cache energy savings for different applications. The total cache energy savings was higher for applications for which *PhLock* used the phases' CMRs (rather than persistence) for determining locked contents. For example, even though *PhLock* achieved a much higher data cache energy savings for *soplex* (79%) than for *gobmk* (59%), the impact of these savings on the total cache energy was less for *soplex* (19%) than for *gobmk* (32%). *PhLock* used persistent phases to determine locked contents for *soplex*, while CMRs was used for *gobmk* because *gobmk* had a few phases that, though not persistent, had high CMRs. For *gobmk*, locking phases with high CMRs reduced execution stalls resulting from lower level memory accesses; this reduction in lower level memory accesses directly impacted the flow of the instruction stream, and improved overall energy savings. Overall, to determine the locked contents for the instruction and data caches, *PhLock* used persistent phases for six and seven applications, respectively, while the CMRs was used for the rest of the applications.

We also investigated *PhLock*'s energy impact for different cache sizes. Fig. 9 compares the energy improvement achieved by *PhLock* as compared to the nonlocking cache for 8, 16, and 32 KB caches, while running the SPEC applications, with all other configurations constant. We observed that for the instruction caches, the energy improvement decreased as the cache size increased. Fig. 9(a) shows that on average across all the applications, *PhLock* reduced the energy by 21%, 10%, and 5% for the 8, 16, and 32 KB instruction caches, respectively. In contrast, *PhLock*'s energy improvement increased as the data cache size increased. As shown in Fig. 9(b), *PhLock* reduced the energy by 16%, 19%, and 39% for the 8, 16, and 32 KB data caches, respectively.

2) MiBench: Fig. 10 depicts the instruction and data caches' energy savings achieved by *PhLock* for the MiBench benchmarks as compared to a nonlocking cache. Unlike for the SPEC2006 benchmarks, *PhLock* achieved much higher energy savings for the instruction cache than the data cache. Fig. 10 (a) shows that on average over all the applications, *PhLock* reduced the total and instruction cache energy consumption by 61% and 75%, respectively, with savings as high as 81% and 85% for *rawcaudio*. In general, even though MiBench applications have a small instruction footprint, cache misses still occur in a nonlocking cache due to address conflicts. In addition, MiBench applications exhibit a high amount of reuse; thus, *PhLock* achieved significant energy savings by locking high-reuse phases in the cache, thereby significantly reducing the conflict misses.



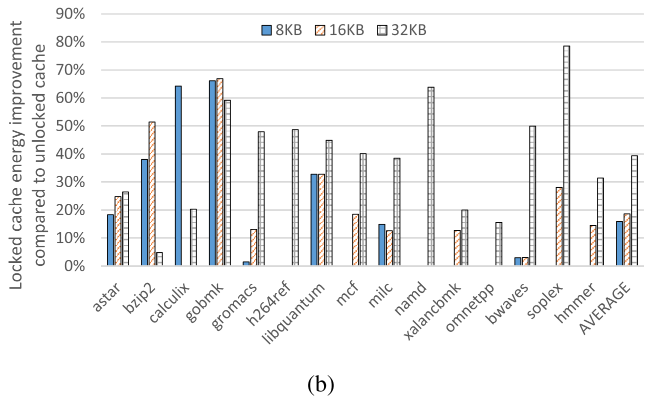
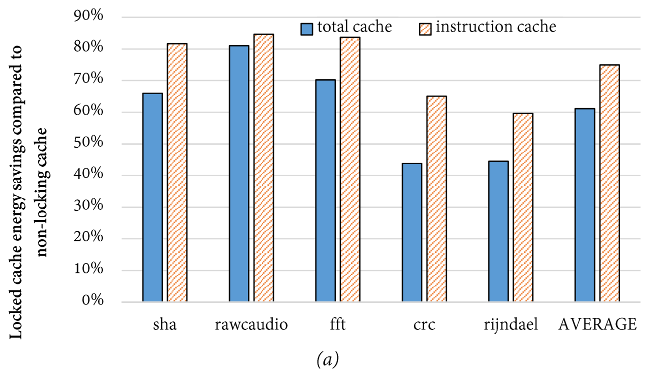


Fig. 9. *PhLock* instruction and data cache energy improvement compared to nonlocking cache for different cache sizes. (a) Instruction cache. (b) Data cache.

Fig. 10(b) shows that *PhLock* achieved data and total cache energy savings of 14% and 4%, respectively, with savings as high as 28% and 9% for *crc*. For most of the applications, locking the data cache had no impact on the instruction cache accesses; thus, the total cache energy savings were modest. *PhLock* achieved much lower energy savings for



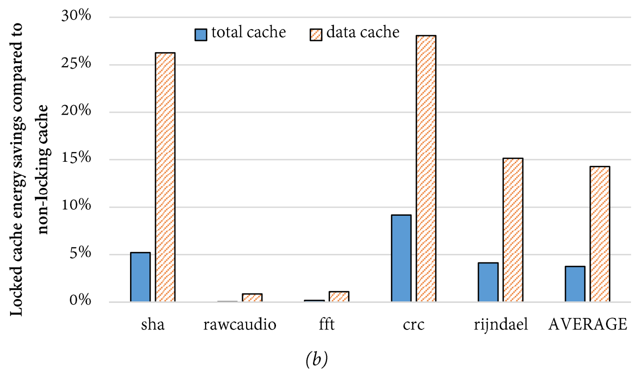


Fig. 10. *PhLock* energy improvement compared to the nonlocking cache for (a) instruction cache and (b) data cache, for MiBench applications.

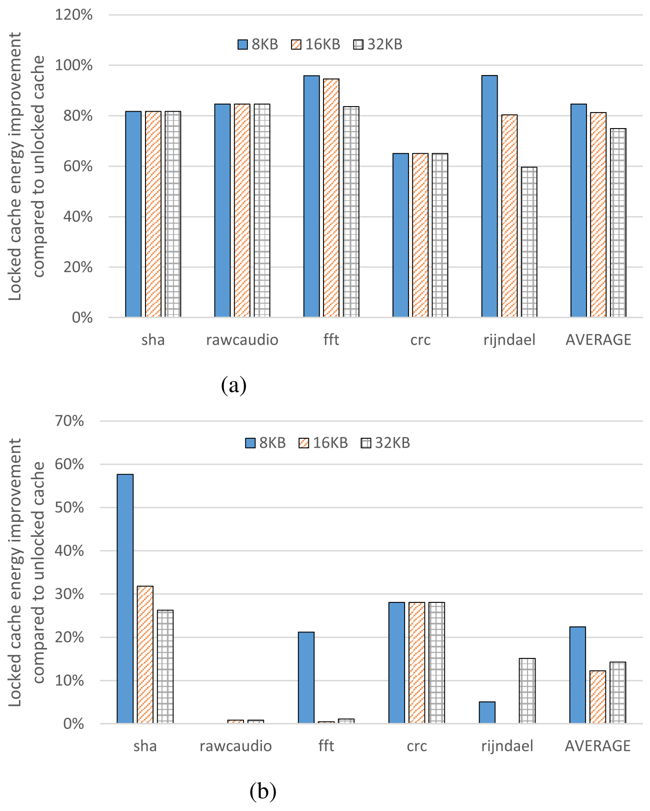


Fig. 11. *PhLock* instruction and data cache energy improvement compared to nonlocking cache for different cache sizes. (a) Instruction cache. (b) Data cache.

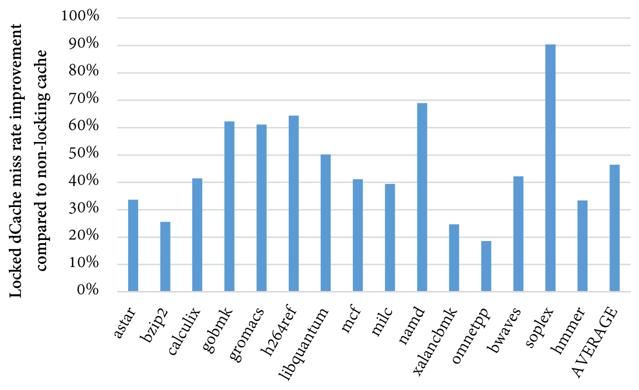


Fig. 12. *PhLock* data CMR improvement compared to nonlocking cache for SPEC2006 applications.

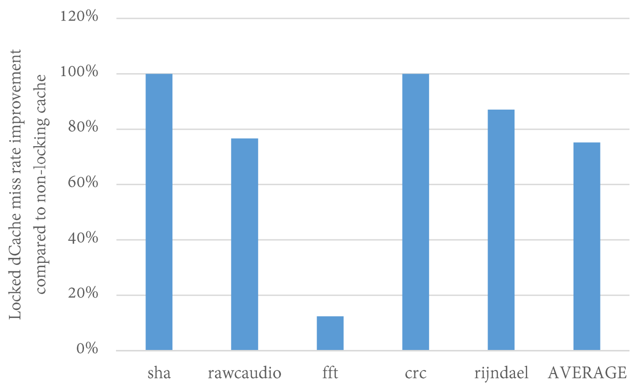


Fig. 13. *PhLock* data CMR improvement compared to nonlocking cache for MiBench applications.

the data cache than instruction cache because the MiBench applications' data exhibit much less runtime variability and reuse than the instructions. The instructions, on the other hand, had a few common kernels that were repeatedly executed; thus locking instructions resulted in more substantial savings than locking data. In general, *PhLock* used persistent phases to determine three applications' instruction cache locked contents and one application's data cache locked contents; the rest were determined using the CMRs.

Similar to the SPEC applications, Fig. 11 shows that *PhLock* also achieves energy savings for different cache sizes while running MiBench applications. Fig. 11(a) shows that *PhLock* reduced the energy by 85%, 81%, and 75% for the 8, 16, and 32 KB instruction caches, respectively. For that 8, 16, and 16 KB data caches, *PhLock* reduced the energy by 22%, 12%, and 14%, respectively.

C. Cache Miss Rates

Even though *PhLock*'s major goal is to save energy, we also analyzed *PhLock*'s ability to reduce the instruction and data CMRs for the SPEC2006 and MiBench applications. Due to

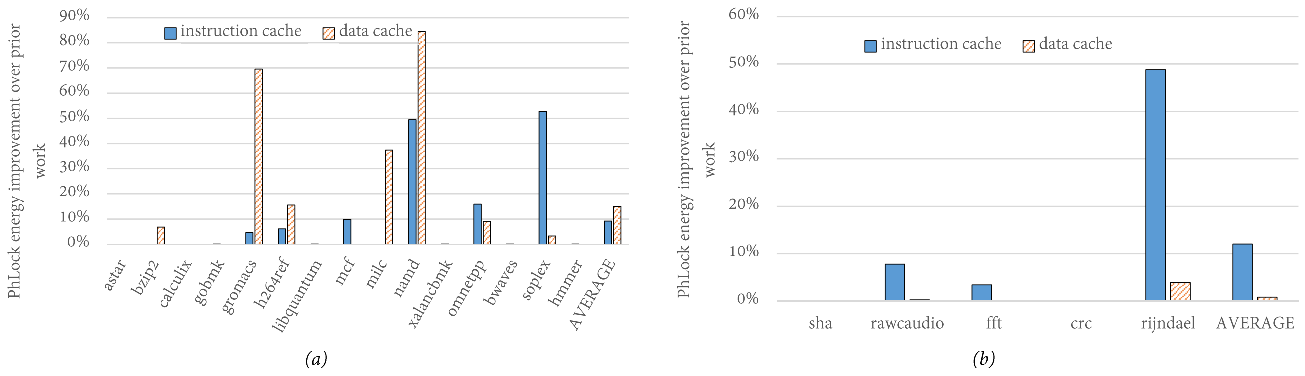


Fig. 14. *PhLock* instruction and data cache energy consumption compared to prior work.

the generally low instruction footprints, *PhLock* had minimal impact on SPEC2006 and MiBench applications' instruction CMRs; *PhLock* set $noLockedContents = 1$ for several applications from both benchmark suites (especially SPEC) and offered no reduction in instruction CMRs, as compared to the nonlocking cache, for those applications.

For both SPEC2006 and MiBench applications, however, *PhLock* achieved significant data CMR reduction. Fig. 12 depicts *PhLock*'s data CMR improvements, for the SPEC2006 applications, as compared to the nonlocking cache. On average over all the applications, *PhLock* reduced the CMRs by 46%, with reductions as high as 90% for $soplex$.

Fig. 13 depicts *PhLock*'s data CMR improvements for the MiBench applications. Similarly, *PhLock* reduced the CMRs by an average of 75%, with up to 100% for sha and crc. We note, however, that even with the nonlocking cache, the miss rates were very low. For example, *PhLock* reduced sha's miss rate from 0.0024 to 0.

In general, these results illustrate that the impacts and benefits of cache locking vary for different kinds of applications. Applications with low compute and memory complexity benefit more from instruction cache locking than more complex and data-rich applications. Thus, cache locking techniques must be able to efficiently select locked contents for both instruction and data caches, with minimal overhead. Overall, the results demonstrate *PhLock*'s ability to determine the locked contents at runtime to efficiently reduce the cache energy consumption through cache locking.

D. Comparison With Prior Work

We compared *PhLock* with prior work that used the CMR to select the locked contents. This technique locked memory blocks with the highest CMRs [8]. To provide a basis for comparison, we simulated this method for both instruction and data caches using SimpleScalar.

Fig. 14 depicts *PhLock*'s instruction and data cache energy improvement compared with prior work [8] for (a) SPEC2006 and (b) MiBench applications. Fig. 14(a) shows that on average across all the SPEC2006 applications, *PhLock* improved instruction and data cache energy savings by 9% and 15%, respectively. *PhLock* outperformed prior work in most applications, with improvements as high as 49% and 85% for $namd$. *PhLock* performed similar to prior work in applications for which *PhLock* determined the locked contents using phases with the highest miss rates.

Fig. 14(b) shows that for the MiBench applications, *PhLock* improved the instruction and data cache energy savings, as compared with prior work, by 12% and 1%, respectively. Unlike for the SPEC2006 applications, *PhLock*'s improvement over prior work was more modest for the data cache. *PhLock* performed similar to prior work for four out of five MiBench applications [hence, 0% improvement for those applications in Fig. 14(b)]. For the instruction cache, however, *PhLock* outperformed prior work in three out of five applications, with improvements as high as 49% for $rijndael$, while *PhLock* performed similar to prior work for the other two applications.

VI. Conclusion

In this paper, we proposed *PhLock* for improving the instruction and data caches' energy consumption in general-purpose embedded systems where the executing applications may be unknown *a priori*. *PhLock* leverages fundamentals of phase classification to dynamically determine the cache's locked contents based on the applications' phase persistence and CMRs. *PhLock*'s goal is to improve the cache energy consumption without introducing significant overheads. We analyzed the impact of cache locking in different kinds of applications, which we experimentally represented using applications from the SPEC2006 and MiBench suites. Compared to a nonlocking cache, *PhLock* improved the instruction and data cache energy consumption by an average of 5% and 39%, respectively, for SPEC2006 applications, and by 75% and 14%, respectively, for MiBench benchmarks. In the future work, we will study the interplay of cache coherence protocols that take cache locking into account and the energy savings achieved by cache locking. We will also explore the synergy of cache locking with other cache optimizations, such as configurable caches.

Acknowledgment

Any opinions, findings, and conclusions of recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] *Arm*. Accessed: Dec. 2016. [Online]. Available: http://www.arm.com

[2] *Mips32 m14k*. Accessed: Apr. 2017. [Online]. Available: https://imagination-technologies-cloudfront-assets.s3.amazonaws.com/documentation/MD00668-2B-M14K-SUM-02.04.pdf

[3] *Spec cpu2006*. Accessed: Jan. 2016. http://www.spec.org/cpu2006

[4] T. Adegbija and A. Gordon-Ross, "Exploiting dynamic phase distance mapping for phase-based tuning of embedded systems," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 363–368.

[5] K. Anand and R. Barua, "Instruction cache locking inside a binary rewriter," in *Proc. Int. Conf. Compil., Archit., Synthesis Embedded Syst.*, 2009, pp. 185–194.

[6] K. Anand and R. Barua, "Instruction-cache locking for improving embedded systems performance," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 3, p. 53, 2015.

[7] A. Asaduzzaman, I. Mahgoub, and F. N. Sibai, "Impact of L1 entire locking and L2 way locking on the performance, power consumption, and predictability of multicore real-time systems," in *Proc. IEEE/ACS Int. Conf. Comput. Syst. Appl. (AICCSA)*, May 2009, pp. 705–711.

[8] A. Asaduzzaman, F. N. Sibai, and M. Rani, "Improving cache locking performance of modern embedded systems via the addition of a miss table at the L2 cache level," *J. Syst. Archit.*, vol. 56, nos. 4–6, pp. 151–162, 2010.

[9] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proc. 33rd Annu. ACM/IEEE Int. Symp. Microarchit.*, 2000, pp. 245–257.

[10] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.

[11] Compiler, Design and User, RTL and Guide, Modeling, *Synopsys*, Synopsys Inc., Mountain View, CA, USA, 2001. [Online]. Available: http://www.synopsys.com

[12] H. Ding, Y. Liang, and T. Mitra, "WCET-Centric dynamic instruction cache locking," in *Proc. Conf. Design, Autom. Test Eur. Exhib.*, Mar. 2014, p. 27.

[13] H. Falk, S. Plazar, and H. Theiling, "Compile-time decided instruction cache locking using worst-case execution paths," in *Proc. 5th IEEE/ACM Int. Conf. Hardware/Softw. Codesign Syst. Synthesis*, 2007, pp. 143–148.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Int. Workshop Workload Characterization (WWC)*, 2001, pp. 3–14.

[15] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SIMPOINT 3.0: Faster and more flexible program phase analysis," *J. Instruct. Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[16] Y. Liang and T. Mitra, "Instruction cache locking using temporal reuse profile," in *Proc. 47th Design Autom. Conf.*, 2010, pp. 344–349.

[17] Y. Liang, T. Mitra, and L. Ju, "Instruction cache locking using temporal reuse profile," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 34, no. 9, pp. 1387–1400, Sep. 2015.

[18] T. Liu, M. Li, and C. J. Xue, "Instruction cache locking for multi-task real-time embedded systems," *Real-Time Syst.*, vol. 48, no. 2, pp. 166–197, 2012.

[19] M. Loach and W. Zhang, "Exploring hybrid cache locking to balance performance and time predictability," in *Proc. SoutheastCon*, Apr. 2015, pp. 1–4.

[20] A. Lukefahr *et al.*, "Composite cores: Pushing heterogeneity into a core," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, Dec. 2012, pp. 317–328.

[21] A. Malik, B. Moyer, and D. Cermak, "A low power unified cache architecture providing power and performance flexibility," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, Jul. 2000, pp. 241–243.

[22] S. Mittal, "A survey of architectural techniques for improving cache power efficiency," *Sustain. Comput., Inform. Syst.*, vol. 4, no. 1, pp. 33–43, 2014.

[23] S. Mittal, "A survey of techniques for cache locking," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 3, p. 49, 2016.

[24] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," HP Lab., Palo Alto, CA, USA, Tech. Rep. HPL-2009-85, 2009, pp. 22–31.

[25] F. Ni, X. Long, H. Wan, and X. Gao, "Combining instruction prefetching with partial cache locking to improve WCET in real-time systems," *PLoS ONE*, vol. 8, no. 12, p. e82975, 2013.

[26] I. Puaut and A. Arnaud, "Dynamic instruction cache locking in hard real-time systems," in *Proc. 14th Int. Conf. Real-Time Netw. Syst.*, 2006, pp. 1–10.

[27] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *Proc. 23rd IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2002, pp. 114–123.

[28] M. Rawlins and A. Gordon-Ross, "An application classification guided cache tuning heuristic for multi-core architectures," in *Proc. 17th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2012, pp. 23–28.

[29] A. Sarkar, F. Müeller, and H. Ramaprasad, "Static task partitioning for locked caches in multicore real-time systems," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 1, p. 4, 2015.

[30] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," *ACM SIGOPS Operat. Syst. Rev.*, vol. 39, no. 11, pp. 165–176, 2004.

[31] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE Micro*, vol. 23, no. 6, pp. 84–93, Nov. 2003.

[32] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," *ACM SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 336–349, 2003.

[33] K. Kang, K.-J. Park, and H. Kim, "Functional-level energy characterization of μC/OS-II and cache locking for energy saving," *Bell Labs Tech. J.*, vol. 17, no. 1, pp. 219–227, 2012.

[34] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 272–282, 2003.

[35] X. Vera, B. Lisper, and J. Xue, "Data cache locking for tight timing calculations," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 1, p. 4, 2007.

Tosiron Adegbija (M'11) received the B.Eng. degree in electrical engineering from the University of Ilorin, Ilorin, Nigeria, in 2005, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Florida, Gainesville, FL, USA, in 2011 and 2015, respectively.

He is currently an Assistant Professor of electrical and computer engineering with the University of Arizona, Tucson, AZ, USA. His current research interests include computer architecture, with an emphasis on adaptable computing, low-power embedded systems design and optimization methodologies, and microprocessor optimizations for the Internet of Things.

Dr. Adegbija was a recipient of the Best Paper Award at the Ph.D. forum of the IEEE Computer Society Annual Symposium on VLSI in 2014.

Ann Gordon-Ross (M'00) received the B.S. and Ph.D. degrees in computer science and engineering from the University of California, Riverside, CA, USA, in 2000 and 2007, respectively.

She is currently an Associate Professor of electrical and computer engineering with the University of Florida, Gainesville, FL, USA, where he is also a member of the NSF Center for High Performance Reconfigurable Computing. She is also a Faculty Advisor of the women in electrical and computer engineering and the Phi Sigma Rho National Society for women in engineering and engineering technology, and an active member of the women in engineering proactive network. Her current research interests include embedded systems, computer architecture, low-power design, reconfigurable computing, dynamic optimizations, hardware design, real-time systems, and multicore platforms.

Dr. Gordon-Ross was a recipient of the CAREER Award from the National Science Foundation in 2010, the Best Paper Awards at the Great Lakes Symposium on VLSI in 2010, the IARIA International Conference on Mobile Ubiquitous Computing, Systems, Services, and Technologies in 2010, and the Best Ph.D. Poster at the IEEE Computer Society Annual Symposium on VLSI in 2014. She is very active in promoting diversity in STEM fields, and has been a Guest Speaker at several international workshops/conferences on this topic, organizes workshops, and participates in local outreach programs at local K-12 schools.