Phase-based Cache Locking for Embedded Systems

Tosiron Adegbija and Ann Gordon-Ross*
Department of Electrical and Computer Engineering, University of Florida (UF), Gainesville, FL 32611, USA
tosironkbd@ufl.edu & ann@ece.ufl.edu
Also affiliated with the NSF Center for High-Performance Reconfigurable Computing (CHREC) at UF

ABSTRACT
Since caches are commonly used in embedded systems, which typically have stringent design constraints imposed by physical size, battery capacity, real-time deadlines, etc., 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 performance and energy consumption. In this paper, we introduce *phase-based cache locking*, which leverages an application's varying runtime characteristics to dynamically select the locked memory contents to optimize cache performance and energy consumption. Experimental results show that our phase-based cache locking methodology can improve the data cache's miss rates and energy consumption by an average of 24% and 20%, respectively.

Categories and Subject Descriptors
B.3.2 [**Hardware**]: Memory Structures: Design Styles – *cache memories.*

General Terms
Design.

Keywords
Cache locking, phase-based tuning, energy savings, configurable caches, dynamic optimization.

1. 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 (e.g., the ARM920T's caches consume up to 44% of the microprocessor's overall energy consumption [15]). Therefore, much research focuses on cache optimizations, such as improved performance and/or reduced energy consumption, while satisfying an embedded system's intrinsic design constraints imposed by physical size, battery capacity, real-time deadlines, consumer market competition, etc.

Cache locking is a popular cache optimization that loads and retains/locks selected contents/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 processors [3]. 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 traditionally benefits execution time predictability when using caches, especially in real time systems where the worst-case execution time (WCET) must be estimated. In these 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 [9] 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 result in reduced dynamic energy since cache locking can reduce cache misses, and thus reduce the energy consumed when 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 performance [21] and/or energy, especially for static cache locking where the locked contents are retained throughout the system's lifetime.

Prior cache locking methods (e.g., [14]) used static cache locking to improve instruction cache predictability in real time systems where the applications and cache contents are known at design time. However, assuming this a priori knowledge limits these methods' applicability to general purpose embedded systems (e.g., smartphones, tablets, etc.), which typically execute a large variety of applications that are unknown at design time. Furthermore, those studies focused on improving predictability without necessarily improving cache performance and/or energy. Alternatively, dynamic cache locking [4][7][21] adjusts the locked contents at runtime to further improve cache predictability and reduce dependence on a priori application and cache content knowledge..

Anand et al. [2], Liang et al. [9], and Liu et al. [10] used cache locking to optimize instruction cache performance in general purpose embedded systems, but none of these works evaluated cache locking's energy benefits. Additionally, since an application typically processes much more data than the number of instructions executed, most prior cache locking methods, 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,

GLSVLSI'15, May 20–22, 2015, Pittsburgh, PA, USA.

http://dx.doi.org/10.1145/2742060.2742076

since complex runtime analysis would be required due to the inherent runtime variability of data caching [21].

Therefore, we propose a new methodology for leveraging cache locking for data cache performance and energy consumption optimizations in general purpose embedded systems. The locked data cache contents are dynamically selected, loaded, and retained at runtime based on the application's intrinsic runtime variable characteristics (e.g., cache miss rates, branch mispredicts, etc.). Unlike instructions, which typically remain fixed during execution, applications process different data streams during runtime, thus our cache locking method dynamically changes the locked contents based on the application's changing data. Prior work showed that *phase classification* can partition an application's execution into execution intervals and group intervals with similar and stable characteristics as *phases*, which typically exhibit data reuse [16]. Our work leverages this data reuse and is based on the premise that cache performance and energy consumption can be optimized if memory blocks with high reuse are locked in the cache, guaranteeing that all accesses to that data are cache hits. Our analyses showed that a few *persistent phases* repeat several times throughout an application's execution, thus we propose to lock those persistent phases' data in the cache, thereby eliminating the conflict misses for those phases.

In this paper, we propose *phase-based cache locking* to dynamically select locked data cache contents for cache performance and energy consumption optimization. We empirically show that cache locking can significantly reduce the data cache's energy consumption when the locked contents are selected to minimize an application's conflict misses.

2. BACKGROUND AND RELATED WORK

Much previous work studied cache locking's execution time predictability benefits and phase classification for exploiting an application's runtime variability in isolation. However, little prior work exploits cache locking for optimizing the cache's performance and/or energy consumption while considering runtime variability. In this section, we present general related work and background on cache locking and phase classification, which we leverage for dynamically selecting the locked contents.

2.1 Cache Locking

Cache locking is primarily used in hard real time systems to improve the cache's predictability and facilitate tighter WCET estimations as compared to a system without cache locking—a non-locking cache. Puaut et al. [14] proposed greedy algorithms for selecting the locked contents in hard real time systems. Vera et al. [21] combined compile-time cache analysis with data cache locking to enable tight WCET estimation in real time systems. Since these works targeted real time embedded systems where the executing applications are typically known a priori, these works have limited applicability to general purpose embedded systems. Furthermore, even though these works improved cache predictability, these works did not explicitly focus on improving the cache performance and the proposed cache locking methods could potentially increase the conflict misses for the memory blocks that are not locked [21].

To improve the cache performance in general purpose embedded systems, Liang et al. [9] presented an instruction cache locking heuristic to select the locked contents in order to realize cache locking's performance benefits by reducing the conflict misses. The proposed heuristic reduced the cache miss rates by up to 24%. Anand et al. [2] 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. [10] proposed an algorithm that dynamically determined the instruction cache's locked contents to improve the average-case execution time (ACET). 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. [5] showed that cache locking could potentially improve cache performance and reduce power consumption. Yang et al. [21] used a dynamic programming algorithm to determine the locked contents in order to improve the data cache's power consumption and performance. However, the authors used a compiler-assisted technique that constrained the proposed method to systems where the executing applications were known a priori.

Our work differs from previous cache locking methods by using dynamic cache locking in the data cache to optimize the cache's performance and energy consumption. We propose 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.

2.2 Phase Classification

Since dynamically leveraging phase characteristics can significantly increase optimization potential by specializing the optimizations to different phases of execution [1][8][17], much prior work explored different phase classification techniques. Sherwood et al. [17] showed that phase classification using basic block distribution was highly correlated with application characteristics, such as cache miss rates, instructions per cycle (IPC), branch mispredictions, etc. Hamerly et al. [8] 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. [16] 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.

3. PHASE-BASED CACHE LOCKING

Our phase-based cache locking methodology selects the locked contents such that the cache's performance and energy consumption are improved compared to a default non-locking cache. Additionally, our methodology determines if an application will benefit from cache locking based on the phases' persistence, such that our methodology never degrades the performance and/or energy consumption as compared to a non-locking cache. In this section, we describe our phase-based cache locking architecture, our methodology for selecting the locked contents, and present our phase-based cache locking algorithm.

3.1 Architecture and Implementation

Our work assumes line locking [7], which is supported in the ARM processor family [3]. 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. Figure 1 depicts our phase-based cache locking architecture for a sample dual-core system, where each core has private level one (L1) instruction and data caches. The phase-based cache locking module (referred to as the locking module for brevity herein) connects directly to each

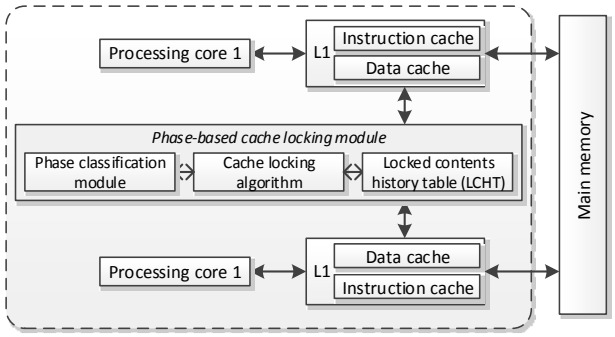

Figure 1. Phase-based cache locking architecture

core's L1 data cache, thus this architecture is extendable to any *n*-core system by connecting the locking module to each core's L1 data cache. Since the locking module contains simple logical operations, the locking module can be implemented using small custom hardware or a lightweight co-processor process to facilitate easy integration into current embedded system microprocessors.

The locking module contains a *phase classification module* to classify the applications' phases and determine the phases' persistence, a *cache locking algorithm*, and a *locked contents history table (LCHT)*. The LCHT is a small hardware or software data structure with per-application entries that retain information (memory addresses and working set sizes) of an application's locked phases' data for subsequent executions of that application. The LCHT's size 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.

Figure 2 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(Ai)* as selected by the cache locking 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 cache miss rate and energy while executing with a non-locking cache for determining if A_i benefits from cache locking (Section 3.3). We estimate the LCHT's overhead in Section 3.4.

Traditional static cache locking restricts the cache's replacement policy from considering locked contents for replacement throughout the system's lifetime. Alternatively, dynamic cache locking inserts special lock and unlock instructions that call the microprocessor's cache locking subroutines. These subroutines disable or enable the cache's replacement policy at the beginning and end of locked contents, respectively. However, since the application code is modified, which alters the application's memory map, data may be

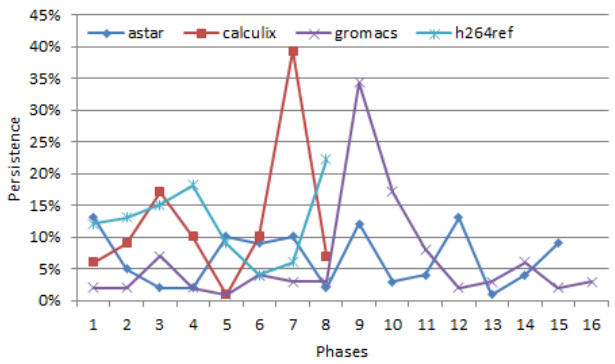

Figure 3. Phase persistence for SPEC2006 benchmarks.

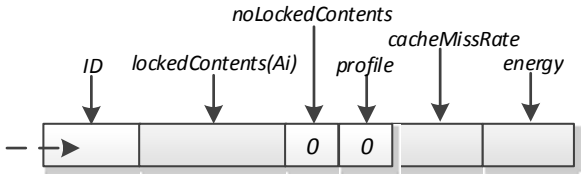

Figure 2. Locked contents history table (LCHT) entry basic structure

written to, and/or read from, the wrong cache sets. Our cache locking methodology avoids application code modification by using the debug registers, which store program counter values that represent the beginning and ending instructions of the locked content's phase. The locking module sets the debug registers such that the exception handler loads and locks the contents in the cache. Using this method does not modify the application's memory map, and can be directly used with any legacy binary. Previous work [4] using a similar method showed that this method accrued negligible runtime overhead.

3.2 Selecting the Locked Contents

To select the locked contents, our phase-based cache locking methodology leverages application execution locality, wherein 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. To ascertain the extent of application execution locality, we analyzed several applications and the applications' phases to evaluate the benefits of locking these phases' data in the cache. Our analysis revealed that for cache locking to provide any cache locking benefits, 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:

$$P_i \in \boldsymbol{A} : I_{Pi} \geq 0.1 * I_{total} \quad (1)$$

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:

$$\rho = \frac{I_{Pi}}{I_{total}} \quad (2)$$

and ρ is a percentage between 0 and 100%. We note that persistence is a necessary, but not sufficient condition for P_i to provide cache locking benefits.

Figure 3 analyzes phase persistence for an arbitrary subset of applications from SPEC CPU2006 [19]. 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%). We evaluated SPEC benchmarks because these benchmarks show a greater variation during execution than typical embedded system benchmark suites, which typically model specific kernels, rather than complete embedded system applications that would be comprised of several of these kernels (e.g., a digital camera's application would contain specific kernels, such as JPEG compression and decompression, MPEG compression and decompression, etc.).

Figure 3 shows that 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. 51% of *gromacs*'s execution is spent

```
1   Inputs: Ai;
2   Outputs: @lockedContents(Ai);
3   if(!(lockedContents(Ai) in LCHT)) {
4        #profile Ai to determine phases and persistence
5             execute(Ai);
6             @phases_Ai, ρ(phases_Ai) <- profile(Ai);
7             #sort phases' persistence in descending order
8             @pers_phases_Ai = sort {ρ{$b} <=> ρ{$a}
9                  } @phases_Ai;
10            foreach phase(@pers_phases_Ai) {
11                 if(sizeOf(lockedContents(Ai)) <=
12                      sizeOf(maxLockedCache)) {
13                           push(@lockedContents(Ai), phase);
14                 }
15                 else {
16                      break;
17                 }
18            }
19            storeLockedContentsLCHT(Ai);
20            profile = 1;
21  }
22  if(lockedContents(Ai) in LCHT &&
23            profile = 1) {
24                 #Ai previously executed once
25                 loadAndLock(@lockedContents(Ai));
26                 execute(Ai);
27                 if(missRates(locking) > missRates(non-locking)
28                      || energy(locking) > energy(non-locking))
29                 {
30                           noLockedContents = 1;
31                 }
32                 profile = 0;
33  }
34  elsif(lockedContents(Ai) in LCHT &&
35       profile = 0) {
36            #locked contents have been determined
37            loadAndLock(@lockedContents(Ai));
38  }
39  if(noLockedContents = 1) {
40       #Ai has no locked contents
41       break;
42  }
```

Algorithm 1. Phase-based cache locking algorithm

in two phases while the remaining 49% of the execution time is spent in the remaining fourteen phases—3% on average for each remaining 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 phase-based cache locking since no phase has a prominent persistence. Our phase-based cache locking methodology identifies these applications, and executes these applications with a non-locking cache to prevent performance and/or energy degradation. Results in Section 4 verify these persistence-based cache locking benefit hypotheses.

Since phase classification has been extensively studied, and data reuse is highly correlated with phase characteristics, the phase classification module profiles the application with a non-locking cache during the application's first execution. The phase classification module uses a phase classification technique similar to [17] to partition the application's execution into phases and uses Equations (1) and (2) to determine the phases' persistence at runtime. Low-overhead, custom hardware profiles the application and groups the application's intervals into phases. The phases are formed at runtime by tracking the program counter (PC) from committed branch instructions and the number of instructions between the current and previous branch to create a basic block vector for each execution interval. Each interval's vector is compared with previous vectors, and similar intervals are grouped into phases. We refer the reader to [17] for additional details.

Without loss of generality and considering general purpose embedded systems, our methodology assumes a system without preemption. However, our methodology could easily incorporate preemption by saving the LCHT's profiling state on application preemption, and restoring the profiling state on resumption. Our future work will evaluate the impact of preemption and context switches on our methodology's effectiveness.

3.3 Phase-based Cache Locking Algorithm

Algorithm 1 depicts our phase-based cache locking algorithm, which takes as input application A_i and outputs an array of A_i's locked contents' memory addresses *lockedContents(Ai)* (lines 1-2). For each application A_i, if the LCHT contains no entry for A_i, our cache locking algorithm profiles and selects A_i's locked contents during A_i's first execution using the non-locking cache (lines 3-21). After A_i completes execution, the algorithm selects A_i's locked contents by sorting A_i's persistent phases by persistence in descending order (lines 7-9), and selects phases for locking in descending order until the total data locked by the selected phases exceeds *maxLockedCache* (lines 10-18). The total data is the working set size of the locked contents, where the working set size is calculated by the number of unique 64-byte 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.

After selecting the locked contents, a new entry for A_i containing A_i's locked contents' memory addresses are added to the LCHT and the *profile* flag is set (lines 19-20). For subsequent executions of A_i, if the LCHT contains an entry for A_i's locked contents and *profile* is set, this is A_i's second execution and the cache locking algorithm locks the selected contents, and determines if A_i will benefit from cache locking after A_i's second complete execution. The cache locking algorithm determines if A_i benefits from cache locking by comparing A_i's cache miss rate and energy consumption while executing with and without locked contents. If cache locking increases the cache miss rate or energy consumption with respect to the non-locking cache, the cache locking algorithm sets the *noLockedContents* flag to '1', implying that A_i does not benefit from cache locking. The cache locking algorithm then sets the *profile* flag to '0' to indicate that cache locking's benefit has been determined for A_i (lines 22-33).

If A_i's locked contents' memory addresses are in the LCHT and the *profile* flag is '0' (i.e., A_i has been previously profiled), *loadAndLock()* triggers the processor's cache locking subroutines (Section 3.1), which load and lock A_i's locked contents in the cache for the duration of A_i's execution (lines 34-38). Alternatively, if *noLockedContents* is set, A_i is executed with the non-locking cache (lines 39-42).

3.4 Computational Complexity and LCHT Hardware Area and Power Overhead

Our phase-based cache locking algorithm sorts the persistent phases N with worst-case time complexity O(N log N) 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.

```
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);
```

Figure 4. Energy model.

To show that our phase-based cache locking methodology constitutes minimal hardware area and power overhead, we estimate the LCHT's hardware/memory requirements. For a 32-entry LCHT, 5 bits store the ID, 32 bits store *lockedContents(Ai),* 1 bit each stores the *noLockedContents* and *profile* flags, and 16 bits each store the cache miss rate and energy. Using these assumptions, we estimate from a synthesizable VHDL implementation and synthesis using Synopsys Design Compiler [20] that the 32-entry LCHT constitutes an area of 2.48 μm^2 and power consumption of 56.72 µW. Relative to a MIPS32 M14K [12] 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.

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

We quantified our phase-based cache locking methodology's performance improvement and energy savings using fifteen benchmarks from the SPEC CPU2006 benchmark suite, compiled to Alpha/OSF binaries and executed using the reference input sets. We used SPEC benchmarks because SPEC applications exhibit greater execution variation (i.e., more distinct phases) than most embedded system benchmarks, and thus more rigorously test our methodology. Since embedded system benchmarks are typically kernels performing a specific task (i.e., few distinct phases), our results are pessimistic. We have verified the suitability of SPEC2006 benchmarks through conversations with personnel in the embedded systems microprocessor manufacturing industry. However, our cache locking methodology presented in this paper and the results are applicable to both embedded system applications and desktop applications. To represent embedded system applications, which are typically much smaller than general purpose applications, we used the first 10 billion instructions [11] from each SPEC benchmark and used SimPoint [8] to classify the benchmarks' phases and determine the phases' persistence.

We simulated cache locking using Simplescalar-AlphaLinux's sim-outorder [18] and drove our simulations using Perl scripts. We modeled an embedded system microprocessor with cache configurations similar to the ARM Cortex A15 [3] microprocessor with 32 KB, 4-way set associative private L1 instruction and data caches with 64 byte line sizes. We used sim-profile to collect information about the phases' memory accesses and data reuse.

Figure 4 depicts the energy model used to calculate the L1 data cache's energy consumption. The model calculates the data 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, similar to previous work [11]. We used Simplescalar to gather cache statistics, such as *totalMisses, totalAccesses, totalWritebacks,*

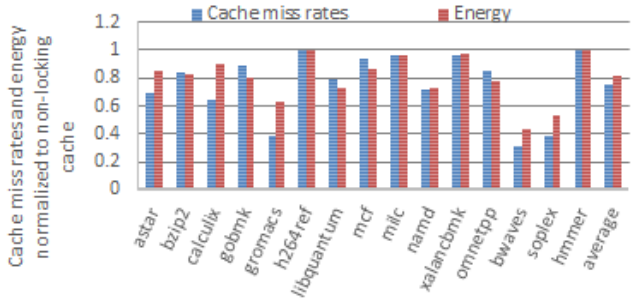

Figure 5. Cache miss rates and energy consumption of phase-based cache locking normalized to a non-locking cache (baseline of one).

etc. 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 [12]. We used CACTI [13] to determine the cache's dynamic energy for 90nm technology.

4.2 Miss Rate and Energy Consumption Analysis Compared to a Non-locking Cache

Figure 5 depicts the data cache's miss rates and energy consumptions for our phase-based cache locking methodology normalized to a non-locking cache (baseline of one). These results depict the system after the cache locking algorithm has selected the locked contents and evaluated the applications' cache locking benefits (i.e., after the first two executions have completed). On average over all of the applications, our cache locking methodology improved the data cache miss rate by 24% compared to the non-locking cache, with improvements as high as 69% for *bwaves* and 61% for both *gromacs* and *soplex*. These applications' cache miss rate improvements were high because a few phases comprised the majority of the applications' execution, and thus substantiates our hypothesis about locking the highly persistent phases. For example, four of *bwaves*'s nineteen phases comprised 47% of *bwaves*' execution. However, locking all four phases would have reduced the effective cache size for the remaining phases and increased those phases' memory blocks' conflict misses. Thus, our cache locking algorithm only locked the two most persistent phases to achieve a 69% miss rate improvement. We observed similar trends for *gromacs* and *soplex*.

However, *h264ref*'s and *hmmer*'s cache miss rates did not improve over the non-locking cache because both applications' phases' persistence were relatively consistent throughout execution, and no one phase was more persistent than any other. For example, since five of *h264ref*'s seven phases comprised 80% of the execution, our cache locking algorithm determined that both applications would not benefit from cache locking, and were executed with the non-locking cache. These results solidify our hypothesis for locking only persistent phases, and if an application's phases do not exhibit persistence, the system should default to a non-locking cache.

With respect to energy consumption, our cache locking methodology improved the data cache's average energy consumption by 20% compared to the non-locking cache, with savings as high as 56% for *bwaves*. Since *h264ref* and *hmmer* did not have any locked contents, phase-based cache locking resulted in the same energy consumption as the non-locking cache. Since our phase-based cache tuning methodology successfully evaluates the benefits of cache locking, and defaults to a non-locking cache when cache locking is not beneficial, there is no cache miss rate and/or energy degradation.

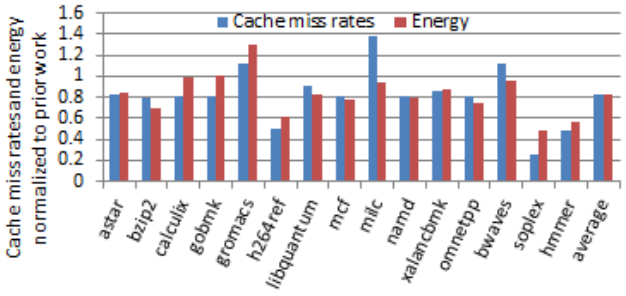

Figure 6. Cache miss rates and energy consumption for phase-based cache locking normalized to prior work (baseline of one).

4.3 Comparison to Prior Work

We compared our phase-based cache locking methodology to prior work that used the cache miss rate to select the locked contents and locked memory blocks with the highest cache miss rates, as determined through extensive runtime analysis [6]. We simulated this method using Simplescalar to provide a close comparison to the state of the art.

Figure 6 depicts the data cache's miss rates and energy consumption when using our phase-based cache locking methodology normalized to prior work (baseline of one). On average over all the applications, phase-based cache locking improved prior work's cache miss rates by 18%, with improvements as high as 74% for *soplex*. Our methodology outperformed prior work for all applications, except *gromacs*, *milc*, and *bwaves* because the memory blocks with the highest miss rates in these applications were included in some, but not all, of the most persistent phases. For example, for *gromacs*, our methodology locked two phases *x* and *y* that comprised of 34% and 17% of the application's execution, respectively. *x* had memory blocks with high miss rates, and these memory blocks were locked by prior work, however, *y*'s memory blocks had very low miss rates, thus, prior work locked different memory blocks that had higher miss rates than *y*'s memory blocks. We can improve our methodology by carrying out runtime analysis to determine the memory blocks' miss rates to lock phases that are persistent and have high miss rates, and we intend to explore this improvement in future work.

Figure 6 also shows that our phase-based cache locking methodology improved the energy consumption as compared to prior work by 17% on average over all of the applications, with improvements as high as 52% for *soplex*. Unlike the cache miss rate comparisons, prior work only outperformed our work for *gromacs* for energy consumption improvement. Prior work locked memory blocks that contributed significantly to *gromacs*'s energy consumption due to those blocks' high miss rates. In general, our methodology also improved over the non-locking cache and outperformed prior work on average. These results show that our phase-based cache locking methodology successfully optimizes the data cache's miss rates and energy consumption without extensive runtime analysis.

5. CONCLUSIONS

In this paper, we proposed phase-based cache locking for improving the data cache's performance and energy consumption in general purpose embedded systems where the executing applications may be unknown a priori. Phase-based cache locking leverages fundamentals of phase classification to dynamically select the data cache's locked contents based on the data's associated phase's persistence, with minimal runtime overhead. Compared to a non-locking cache, our phase-based cache locking methodology improved the data cache's miss rates and energy consumptions by an average of 24% and 20%, respectively. In future work, we will extend our phase-based cache locking methodology to tradeoff the phases' persistence with the phases' memory blocks' miss rates to further improve the cache miss rates and energy consumptions. We will also extend our phase-based cache locking methodology to the instruction cache to optimize the instruction cache's miss rates and energy consumption.

6. ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation (CNS-0953447). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] T. Adegbija, A. Gordon-Ross, and A. Munir, "Phase distance mapping: a phase-based cache tuning methodology for embedded systems," Springer Design Automation for Embedded Systems (DAEM), January 2014.

[2] K. Anand and R. Barua, "Instruction cache locking inside a binary writer," International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES), October 2009.

[3] ARM: http://www.arm.com

[4] A. Arnaud and I. Puaut, "Dynamic instruction cache locking in hard real-time systems," International Conference on Real-Time and Network Systems (RTNS), October 2006.

[5] A. Asaduzzaman, I. Mahgoub, and F. Sibai, "Impact of L1 entire locking and L2 way locking on the performance, power consumption, and predictability of multicore real-time systems," International Conference on Computer Systems and Applications, May 2009.

[6] A. Asaduzzaman, F. Sibai, and M. Rani, "Improving cache locking performance of modern embedded systems via the addition of a miss table at the L2 cache level," Journal of System Architecture, April 2010, pp 151-162.

[7] H. Ding, Y. Liang, and T. Mitra, "WCET-Centric Dynamic Instruction Cache Locking," Design, Automation, and Test in Europe (DATE), March 2014.

[8] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: faster and more flexible program analysis," Journal of Instruction-Level Parallelism, 2005, pp 1-28.

[9] Y. Liang and T. Mitra, "Instruction cache locking using temporal reuse profile," Design Automation Conference (DAC), June 2010.

[10] T. Liu, M. Li, and C. Xue, "Instruction cache locking for embedded systems using probability profile," Journal of Signal Processing Systems, November 2012.

[11] A. Lukefahr, et al., "Composite cores: pushing heterogeneity into a core," International Symposium on Microarchitecture, Dec. 2012.

[12] MIPS32 M14K. http://files.tomek.cedro.info/electronics/doc/mips/MD00688-2B-M14K-APP-01.00.pdf. Accessed 30 July 2014.

[13] N. Muralimanohar and N. P. Jouppi, "Cacti6.0: A tool to model large caches," COMPAQ Western Research Lab, 2009.

[14] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," Real-Time Systems Symposium (RTSS), 2002.

[15] S. Segars, "Low-power design techniques for microprocessor," International Solid-State Circuits Conference Tutorial, Feb. 2001.

[16] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), December 2004.

[17] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," International Symposium on Computer Architecture, December 2003.

[18] Simplescalar ported to Alpha/Linux with Linux System Calls. http://hhnajafabadi.s3-website-us-east-1.amazonaws.com/mase-alphalinux.htm

[19] SPEC CPU2006. http://www.spec.org/cpu2006

[20] Synopsys Design Compiler, Synopsys Inc. www.synopsys.com

[21] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, June 2003.