A

Instruction Cache Locking for Improving Embedded Systems Performance

Kapil Anand, University of Maryland, College Park
Rajeev Barua, University of Maryland, College Park

Cache memories in embedded systems play an important role in reducing the execution time of the applications. Various kinds of extensions have been added to cache hardware to enable software involvement in replacement decisions, thus improving the run-time over a purely hardware-managed cache. Novel embedded systems, such as Intel's XScale and ARM Cortex processors provide the facility of locking one or more lines in cache - this feature is called *cache locking*. This paper presents a method in for instruction-cache locking that is able to reduce the average-case runtime of a program. We demonstrate that the optimal solution for instruction cache locking can be obtained in polynomial time. However, a fundamental lack of correlation between cache hardware and software program points renders such optimal solutions impractical.

Instead, we propose two practical heuristics based approaches to achieve cache locking. First, we present a `static` mechanism for locking the cache where the locked contents of the cache are kept fixed over the execution of the program. Next, we present a `dynamic` mechanism which accounts for changing program requirements at runtime. We devise a cost-benefit model to discover the memory addresses which should be locked in the cache. We implement our scheme inside a binary rewriter, thus widening the applicability of our scheme to binaries compiled using any compiler.

Results obtained on a suite of MiBench benchmarks show that our `static` mechanism results in 20% improvement in the instruction-cache miss rate on average and upto 18% improvement in the execution time on average for applications having instruction accesses as a bottleneck, compared to no cache locking. The `dynamic` mechanism improves the cache miss rate by 35% on average and execution time by 32% on instruction-cache constrained applications.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Binary Rewriting, Caches, Cache Locking,Embedded Systems

ACM Reference Format:
Kapil Anand and Rajeev Barua, 20XX. Instruction Cache Locking for Improving Embedded Systems Performance. *ACM Trans. Embedd. Comput. Syst.* V, N, Article A (January YYYY), 25 pages.
`DOI`:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Modern embedded systems employ several memory technologies to meet stringent run-time and power consumption constraints. SRAM and DRAM are the two most common memories used for storing program code and data. Due to the relative cost and performance of these memories, a large amount of DRAM is often complemented with a small-size on-chip SRAM. A proper use of SRAM in embedded systems is imperative in meeting run-time and energy constraints.

SRAM is most commonly managed in the form of a hardware cache. A cache dynamically stores a subset of frequently used data or instructions following a fixed replacement policy.

Various different approaches have been suggested to enable software involvement in the management of on-chip memory. One approach involves the addition of a lightweight software-controlled memory which relies on explicit compiler support for data allocation. Another approach involves explicit modifications to the cache memory structure and availability of programmer level cache control instructions to enable direct software involvement in cache replacement decisions.

On similar lines, several embedded systems such as Intel's XScale and ARM's Cortex processors provide a facility of locking one or more lines in the cache - this feature is called *cache locking*. An address, once locked in the cache, always results in a hit on subsequent accesses unless an unlocking operation is explicitly carried out. Hence, a software application can influence replacement decisions made by the cache and thereby alleviate potential mistakes resulting from cache hardware. As an example, if a soon-to-be-accessed element is susceptible to replacement according to the underlying cache replacement policy in favor of an element that will not be accessed soon, locking this element in the cache will result in a better cache performance.

However, current methods regarding instruction cache locking are geared towards improving real-time predictability of applications [Puaut and Decotigny 2002; Puaut 2002; Falk et al. 2007; Vera et al. 2003]. These methods employ instruction cache locking for adapting the cache to multi-task real-time systems.

We presented the first method in literature [Anand and Barua 2009] employing instruction cache locking as a mechanism for improving the average-case runtime of general embedded applications, thus widening its applicability beyond hard real-time systems. Our scheme is implemented inside a binary rewriter; hence is applicable to binaries compiled using any compiler or software development toolchains and to programs whose source code is not available e.g. legacy code or third party software. Cache locking technique can be applied to both instruction and data caches but in this paper, we limit ourselves to the problem of instruction cache locking.

Liang and Mitra [Liang and Mitra 2010] extended our earlier work [Anand and Barua 2009] and presented an optimal algorithm for static instruction cache locking. However, both these methods only explore static cache locking, where instructions are locked once before the start of a program and remain locked during its entire execution.

In this work, we extend our earlier method and propose a novel dynamic cache locking algorithm, where the addresses locked in the cache are updated dynamically during the execution of a program. Our mechanism identifies the program points with significant shift in program locality and employs a cost-driven model to compute the set of lines which should be locked at each such program point. The input program is instrumented to achieve the locking of required lines at each program point. This mechanism accounts for changing program requirements at runtime and dynamically modifies the cache content.

Several existing instruction cache locking mechanisms [Arnaud and Puaut 2006; Liu et al. 2009] have also presented techniques to divide a program into a set of regions and for computing the contents of cache in each region. However, these techniques for computing the regions are either closely integrated with the mechanism of minimizing the worst case execution time (WCET) of a program or employ a simplistic approach of assuming each basic block as a separate region [Liu et al. 2012].

Instead we propose a novel region partitioning algorithm based on a popular scratchpad allocation mechanism [Udayakumaran et al. 2006] that has been shown to improve the average-case runtime of applications and operates independently of the underlying execution-time minimization algorithm. A program is partitioned into multiple regions based purely on the program call-graph and control-flow graph information. Further, our experiments demonstrate that our earlier static cache conflict model [Anand and Barua 2009] can be successfully employed for determining the cache locking contents in each region for minimizing the average-case runtime.

We also demonstrate that an optimal solution to dynamic instruction cache locking can be obtained in polynomial time. However, as we will discuss in later sections, it is extremely challenging to implement this algorithm in practice due to a fundamental lack of correlation between cache hardware decisions and software program points. Hence, we propose a heuristic based approach for deriving a solution.

The rest of the paper is organized as follows. Section 2 describes the underlying cache locking interface. Section 3 overviews related work and lists the advantages of our method. Section 4 presents a small example to depict the benefit of instruction cache locking. Section 5 formalizes the cache locking problem and its complexity. Section 6 presents our solution for static cache locking while Section 7 presents the dynamic counterpart. Section 8 presents an overview of our implementation framework. Section 9 presents our method's results for different cache and architecture configurations on a variety of benchmarks. Section 10 concludes.

2. CACHE LOCKING INTERFACE

There are two most common kinds of locking mechanisms present in modern embedded systems - way locking and line locking. *Way locking* is a coarse grain approach to cache locking where locking is available at the granularity of ways of a set-associative cache. Locking a particular way in cache implies the way is locked in each set of the set-associative cache. This kind of locking is present in ARM's Cortex processors [ARM 2004] and ARM11 family of processors [ARM 2007].

Line locking is a more fine-grained approach to cache locking. In this interface, the locking mechanism is available at the granularity of a single cache line as opposed to a single way. In this interface, it is possible to have a different number of locked lines in different sets of the cache. Intel's XScale [Xscale 2007], ARM9 family and BlackFin 5xx family processors [BlackFin 2009] support this kind of locking mechanism.

In this paper, we explore the line locking interface present on embedded systems. These platforms provide special co-processor-based lock instructions for locking an address specified as their argument in the cache. In such processors, way 0 of the cache can't be locked; we respect this constraint in deriving our results. However, we emphasize that our method does not require any such constraint and can be applied for locking lines in all the ways of any set.

3. RELATED WORK

There are many existing methods targeting improvement of on-chip memory performance through software involvement. Research in this direction can be broadly categorized in two approaches: (i) approaches involving an additional software-controlled memory apart from, or instead of, the cache; and (ii) approaches involving direct modifications of the cache memory structure.

The first category of methods involve modifications to the memory hierarchy by introducing additional software-controlled memories such as Scratchpad memory (SPM) and loop caches. Various different kind of methods have been suggested for managing the data to be placed in SPM [Sjodin et al. 1998; Banakar et al. 2002; Panda et al. 2000; Verma et al. 2004a; 2004b; Steinke et al. 2002; Avissar et al. 2002]. A *loop cache* [Gordon-Ross et al. 2002] is a small instruction buffer which can be pre-loaded with frequently executed loops and functions thus accelerating their access-time during program execution. SPMs and loop caches are used in industry primarily when the runtime behavior of applications is predictable; or to improve real-time performance. Caches are better at tracking runtime behavior; hence they are widely employed in many non real-time and soft real-time systems.

Even though cache locking tries to achieve the same goal of improving local memory performance, its management strategy is inherently different from allocation problems for the above software-controlled memories. There are two reasons for that. First, when a cache locking method decides to lock a line in the cache, other lines that conflict with it can no longer reside in cache in case of a direct-mapped cache, or have reduced number of slots available in case of a set-associative cache. This opportunity cost does not occur, and is not modeled, by methods for SPM allocation. In contrast, it is modeled

in our method. Correctly modeling this opportunity cost is crucial – employing a SPM allocator oblivious to this cost for locking could exclude heavily used lines from cache, leading to poor runtime. Another reason that SPM allocators are not suitable for cache locking is that a particular element can be placed at any location in SPM, whereas the cache hardware decides the location of each element in a cache. This results in entirely different kinds of constraints for the cache locking problem. The energy model in terms of cache hits and misses suggested in [Verma et al. 2004a] for cache-aware SPM allocation is somewhat similar to the time model we present in our paper but their method addresses a completely different problem.

In the second category, there are methods that involve modifications to the cache hardware to equip software to dynamically modify cache replacement decisions. Chiou et al [Chiou et al. 2000] introduce column caching, to provide software an ability to dynamically partition the on-chip memory into scratchpad memory; Sartor et al [Sartor et al. 2005] propose an extension of each cache line with evict-me and kill-me bits; along with a compile time locality analyzer to determine their values. These methods provide interesting ideas for improving cache performance but rely on hardware modifications that are unavailable in any commercial processors. In contrast, our method is a software-only scheme applicable to a variety of commercial processors.

Jones et al [Jones et al. 2011] proposes a new hybrid hardware and linktime assisted approach to tagless instruction caching that removes the need for tag checks entirely for the majority of cache accesses. Similar to the above methods [Chiou et al. 2000], they rely on hardware modifications not present in existing processors. In contrast, our method is a pure software method that works on existing processors. In addition, the main focus of their work is to reduce power consumption in the cache as opposed to our goal of improving average-case runtime of applications.

Research has been carried out to exploit the cache features present in existing hardware - locking is one such kind of feature available in modern embedded systems. Hollander et al [Beyls and D'Hollander 2005] suggested reuse-distance-based methods for generating cache hints for memory access instructions, available in EPIC architectures, resulting in improved data cache performance. In contrast, we don't target the hardware with cache hints; rather we target cache locking hardware.

Instruction cache locking has primarily been employed as a mechanism for adapting the cache to multi-task real-time systems. In multi-task systems, the presence of caches leads to unpredictability and results in extreme over-estimation of worst case execution time, as each access can result in a miss in the worst case [Puaut and Decotigny 2002]. I-cache locking has been employed in such scenarios to provide predictability; thus improving the worst case estimation. The objective of the cache-content selection problem in such scenarios is to improve the worst case system behavior according to some of real-time schedulability metrics as described in [Puaut and Decotigny 2002; Puaut 2002; Falk et al. 2007; Vera et al. 2003; Arnaud and Puaut 2006; Campoy et al. 2001]. In contrast, our objective of cache-content selection is to improve average-case runtime of embedded applications which is completely different objective, requiring a very different strategy.

There has been very little research on using cache locking for performance improvement of general embedded applications. Hu et al [Yang et al. 2005] presented a method for data cache locking in Itanium and XScale processors based on the length of the reference window for each data-access instruction. In contrast, we present a locking scheme for the instruction cache. Further, their method does not involve finding the optimal number of cache lines to be locked in the cache; rather they rely on locking every possible line which can be locked in cache. The over-aggressive locking might provide negative results and does not ensure that the locked cache would give perform better than cache with no locking. Our method suitably addresses these limitations.

Earlier, we had presented the first method [Anand and Barua 2009] in literature employing instruction cache locking as a mechanism for improving the average-case runtime of general embedded applications. However, our previous work only explored a static solution to cache locking. Liang and Mitra [Liang and Mitra 2010] extended our work and presented an optimal strategy for static instruction cache locking. In this work, we extend our previous work [Anand and Barua 2009] and propose a dynamic solution for cache locking, which accounts for changing program requirements at runtime and updates the locked contents dynamically with program execution.

Several instruction cache locking mechanisms for real-time systems [Arnaud and Puaut 2006; Puaut 2002] have suggested an approach for dynamically updating the contents of cache by dividing the program statically into different regions. As mentioned in Section 1, the region partition algorithm employed by such mechanisms is completely different from our region partition algorithm. The existing methods obtain regions using an iterative mechanism where each such iteration is guided by a WCET metric. They do not present any generic mechanism of replacing this underlying metric. Hence, the only possible way of using their region formation method is to employ their initial regions. As per the method presented by [Arnaud and Puaut 2006], the first step in their method is to consider each basic block as a separate region. Choosing each basic block as a separate region is intuitively a very high overhead operation. Instead of choosing this over-simplified method for region formation, we adapt a region formation algorithm from a popular scratchpad allocation method [Udayakumaran et al. 2006] that has been shown to improve the average-case runtime of applications.

Liu et al [Liu et al. 2012] present a dynamic method for instruction cache locking to improve average-case performance. Their method implicitly divides a program structure into regions by considering each branching node as a potential candidate for reloading corresponding subtree rooted at this node. However, we believe that their method does not accurately capture the impact of dynamic cache locking for two reasons. First, as mentioned in Section 6 in their paper, their method does not model cache conflicts. Consequently their basic method is only applicable to fully-associate caches. Second, in order to apply their method to set-associate caches, they rely on using several compiler optimizations such as code-reordering [Gordon-Ross et al. 2005] and code placement techniques [Zhao et al. 2005] to eliminate the cache conflicts. Their cache-locking method implicitly includes these orthogonal cache optimization strategies, whose impact improves their performance results as well. They do not provide the exclusive impact of cache locking without the above optimizations. The goal of our method is to provide a transparent cache locking scheme that works on pre-compiled production binaries, including third party proprietary binaries, for which source code is not available, and hence compiler optimization is not possible. From their paper, it cannot be deduced that their dynamic cache locking mechanism will necessarily improve the performance over a state-of-the-art static cache locking method when such orthogonal compiler optimization schemes are not employed. In contrast, our proposed dynamic method improves upon the state of art cache locking mechanism [Liang and Mitra 2010] in terms of application runtime.

We summarize the benefits of our scheme: (i) Ours is the first method for employing instruction-cache locking as a mechanism for improving the average-case runtime of general embedded applications, thus widening its applicability beyond hard real-time systems. (ii) Ours is the first dynamic method for instruction cache locking, enabling better results than static schemes. (iii) We provide a profile-based method and derive the cost-benefit from actual cache statistics; thus our method is guaranteed to improve over the performance of cache without locking. (iv) Our method has been implemented inside a binary rewriter, widening its applicability to binaries compiled using any compiler. (v) Our method has an inherent mechanism that greedily determines the number

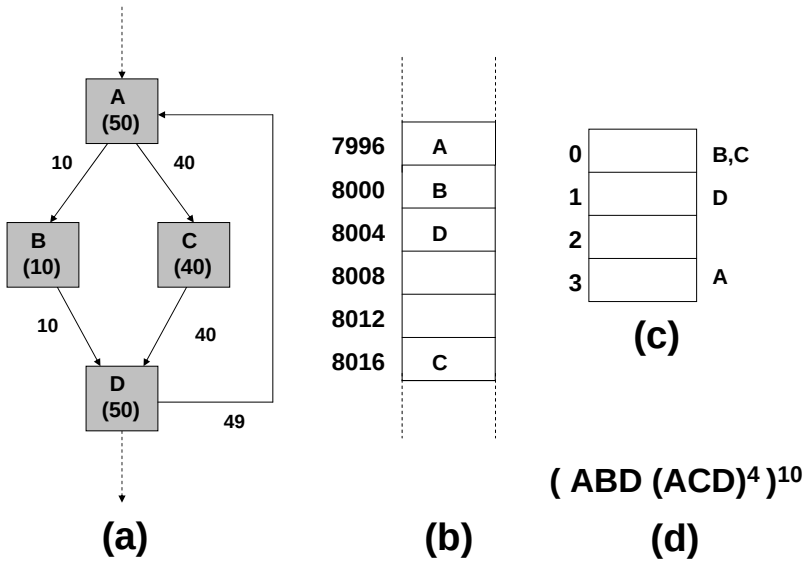


Fig. 1: (a) Weighted CFG of a small part of a program. A, B, C and D are instructions of 4 byte each. (b) A hypothetical memory layout of the above instructions. (c) A dummy 16-byte direct mapped instruction cache. The alphabets at right hand side of each cache line show the instructions which are mapped to the line according to the cache mapping function. (d) The execution trace of this part of the program.

of cache lines to be locked - it does not lock each possible cache line, as suggested by some previous methods. (vi) Cache locking is already available on existing hardware and thus our method does not entail any new hardware modifications, making our approach readily applicable.

4. MOTIVATION

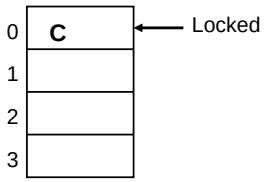
In this section, we present the potential benefits of instruction-cache locking in improving cache efficiency via a small example. Figure 1 shows a weighted control-flow graph (1(a)) and execution trace (1(d)) of a small part of a program; its hypothetical memory layout (1(b)) and a dummy cache configuration (1(c)). The nodes and edges of the control-flow graph are labeled with their execution frequencies as observed during a profile run of the program. The execution trace (1(d)) of the program reveals that a single execution of node B is followed by four instances of node C. This sequence of execution of node B followed by node C is repeated 10 times during the execution of the program. For simplicity, we assume that nodes A, B, C and D contain only a single instruction each and the instruction cache is a tiny 16-byte direct mapped cache with one word per line. Each word and instruction is assumed to be of 4-byte size.The addresses are mapped to the cache lines according to the standard modulo-based cache mapping function:

$$Set = (addr)\ mod\ \frac{Cache\text{-}Size}{Associativity * Words\text{-}Per\text{-}Line} \tag{1}$$

According to the above cache mapping function and the memory layout, instructions B and C share the same line in the cache. During the execution of the above program,

Node	Number of Misses without locking	Number of Misses with locking
A	1	1
B	10	10
C	10	1
D	1	1
Total	22	13

(a)



(b)

Fig. 2: (a)Number of misses observed for each node with and without locking. (b) Locking of node C in set 0 of cache.

node B and node C alternately keep replacing each other in the cache, resulting in a large number of cache misses. The second column in Figure 2(a) shows that this cache configuration leads to 22 misses for this sample program.

Next, assume the presence of locking functionality inside the instruction cache. If node C is locked into cache line 0 then C would not be replaced by node B during the execution of the program. Node C would observe only one compulsory miss while number of misses for B would remain the same. The third column in Figure 2(a) shows the number of misses observed by each node when node C is locked in cache as shown in Figure 2(b). With cache locking, we observe only 13 misses, down from 22 misses in cache without locking. This example highlights the potential of I-cache locking as an effective mechanism for reducing cache misses.

5. THEORETICAL ANALYSIS OF CACHE LOCKING

The cache-locking problem involves selecting the memory addresses which should be locked in the cache during each time interval such that the total number of instruction cache misses over the lifetime of the program is minimized. The solution to this problem is influenced by the behavior of the cache mapping function. In a set-associative cache, an address is mapped to the cache line according to the cache mapping function (1). For a given memory address, this function returns the cache set where the address is mapped in the cache. A particular memory address always gets mapped to the same set in the cache, given by the above function. Thus, given the full range of instruction-memory addresses in the current program, the list of addresses which get mapped to a set during the lifetime of the program can be accurately obtained for each cache set. Once this mapping of addresses to the corresponding set is obtained, *each cache set can be independently analyzed to determine the memory addresses to be locked in that set*.

At each time instant `T`, the cache locking problem has two objectives: (i) determine the number of lines, `L`, that should be locked in this set; and (ii) select these `L` virtual cache lines out of the complete pool of cache lines in this set.

In his seminal paper [Belady. 1966], Belady proposed an optimal offline replacement policy for virtual memory pages, which has been subsequently widely applied for cache analysis as well. Belady's algorithm achieves the lowest possible cache miss rate. Other faster algorithms [Temam 1998] have also been proposed to achieve the optimal cache miss rate. Collectively, the class of such algorithms is referred to as `OPT` algorithm [Temam 1998].

Belady's original algorithm was based on the restriction that each block is placed in the cache when it is referenced. A cache is said to allow bypassing when this restriction is not imposed. As demonstrated by later methods [Brehob et al. 2004; Temam 1998], an algorithm similar to OPT algorithm can be formulated to handle the scenarios when bypassing in allowed in the cache. In our discussion, we define OPT-B as the class of OPT algorithm that assume the presence of bypassing in cache.

As expected, the above research methods [Brehob et al. 2004; Temam 1998] have demonstrated that the solution provided by OPT-B algorithm observes lesser number of misses as compared to the solution provided by OPT algorithm. We demonstrate that OPT-B algorithm can be employed to provide an optimal solution for cache locking problem in each cache set.

The OPT-B algorithm analyzes the cache accesses in a trace in the execution order. The resulting replacement decisions can either be employed for improving cache performance in future executions or for comparing different cache strategies. Intuitively, given a trace of block accesses for the program, the OPT-B algorithm is based on evicting the block which will be referenced furthest in the future. Consider an address X that is referenced twice in a program trace at times t1 and t2, $t2 \geq t1$.

According to OPT-B algorithm, the decision for keeping X in the cache during the time interval {t1,t2} is taken at t2. At time t2, if the number of elements that have already been selected to be in the cache during time interval {t1,t2} is less than cache capacity, then X is added to the list of elements which must be kept in the cache during the {t1,t2}. On the other hand, if the total number of elements already in the cache during time interval {t1,t2} is equal to the cache capacity then X is not kept in cache during this interval.

We observe that the ability to lock an address in the cache provides a tangible mechanism to implement the solution proposed by OPT-B algorithm. For example, in order to keep an element X in the cache during the time interval {t1,t2}, X can be locked in the cache at t1 and unlocked at t2. Consequently, cache locking mechanism actuates the implementation of OPT-B algorithm. Based on this observation, we state the following important lemma.

LEMMA: *An optimal solution for cache locking can be derived in a polynomial time, assuming perfect prior knowledge of memory accesses.*

PROOF: OPT-B is a polynomial time algorithm for obtaining an optimal solution for cache performance. In other words, the OPT-B algorithm minimizes the number of misses in the cache. The cache locking problem shares the same goal of minimizing the number of misses in the cache. The capability of locking a line in cache enables the implementation of each step of OPT-B algorithm in a constant time. Hence, OPT-B is an optimal solution for the cache locking problem as well. The polynomial time complexity of OPT-B results in a polynomial-time optimal algorithm for cache locking.

A perfect (or complete) knowledge of future memory accesses enables OPT-B algorithm to make optimal replacement decisions. Extending this optimal replacement algorithm to the cache locking problem implicitly models the opportunity cost arising due to precluding the remaining elements from the cache during cache locking since OPT-B considers *every* cache line as a possible candidate for locking.

However, it will be extremely challenging to obtain a practical implementation of above cache locking method. As per the OPT-B algorithm, a new decision for evicting a cache line can be taken at any instant during the program execution. If a program address is executed n times, then n different locking decisions can be taken about this particular program address. The mechanism in Section 2 specifies that cache locking instructions corresponding to each particular address need to be inserted at fixed program points. This would require implementation of a state machine that implements

the underlying locking decisions governed by OPT-B. Further, this decisions for this state machine would only be optimal for the particular input whose profiling data is used for OPT-B.

The underlying reason for the above limitation is a fundamental lack of correlation between cache replacement decisions and program points. A cache employs a pure hardware mechanism, transparent to the software layer. At a particular program point, the cache hardware can take distinct decisions based on the history of dynamic accesses at that program point. Hence, it is extremely challenging to implement the optimal cache eviction method inside a program using cache locking instructions.

This is not surprising, since the original OPT algorithm also faces similar practical restrictions. An OPT algorithm cannot be implemented in an actual cache. However, it is still considered as an important analytic tool in comparative studies [Burger et al. 1996; Tam et al. 1999]. The above result establishes that bounds provided by OPT-B algorithm is applicable to the performance of cache locking methods also.

In order to handle the above limitations, we propose two distinct heuristic based solutions to relate cache locking decisions to program points.

→ **Static Cache Locking**: We formulate a static solution to instruction cache locking where instructions are locked once before the start of the program and remain locked during the entire execution of the program. Hence, in this solution, the beginning of the program is the only point where cache locking decisions can be taken.

→ **Dynamic Cache Locking**: In this formulation, we obtain an hybrid between static locking and OPT-B algorithm, where we allow the cache locking decisions to be made at several judiciously chosen program points. Such program points are chosen based on a possibility of a large change in program locality.

Both the above solutions face another pragmatic challenge. As mentioned in Section 2, special locking instructions are provided in target platforms which upon execution lock the elements at specified addresses in the cache lines. The solution to cache locking problem provides a set of addresses which should be locked in the cache at each program instant. However, a direct instrumentation of the program with the instructions for locking these addresses changes the program layout, invalidating the results provided by our algorithm. Hence, the code layout of the updated program should be exactly the same as the original program.

Fortunately, there are existing methods for general modifications of program binaries without modifying the program layout. We adopt the *trampolines* approach suggested in [Buck and Hollingsworth 2000] for modifying the binaries with the extra locking instructions.

First, we insert dummy placeholders at the required program points in the binary. After determining the addresses to be locked in the cache at required program points, the actual locking instruction corresponding to the required lines are inserted at the end of original program layout as a new *trampoline*. A jump to this trampoline is inserted at the placeholders present at corresponding program points. After executing the locking instructions, the trampolines transfer the control back to just after the jump to the trampolines. The remaining placeholders are replaced by NOP instructions. This approach guarantees that the layout of the program is not altered as a result of insertion of locking instructions.

6. STATIC CACHE LOCKING

In this section, we formalize the cache locking problem as an optimization problem and explain our static cache locking algorithm in detail.

Since elements in the cache are locked at the granularity of cache lines and not individual memory addresses, addresses need to be analyzed in terms of cache lines. In

order to represent this situation mathematically, we introduce a new concept of virtual cache line. Given an instruction address, addr, the Virtual Cache Line is defined as

$$\texttt{Virtual Cache Line} = \left\lfloor \frac{\texttt{addr}}{\texttt{Words-Per-Line}} \right\rfloor \tag{2}$$

The remaining analysis for cache locking is carried out in terms of virtual cache lines. We introduce the following definitions to ease the explanation

---

N: Associativity of the cache; s: A cache set
X_s: Set of virtual cache lines which get mapped to set s
M: Cardinality of set X_s.
K: Hardware specified limits on maximum number of lines which can be locked in a set
L: Number of lines to be locked, $L \leq K$

The static cache locking problem has two objectives (i) determining L: the number of lines which should be locked in this set (ii) selecting L virtual cache lines, out of M candidates, which should be locked in the set.

If L lines are locked in this set, L locked virtual cache lines result in L compulsory misses and no other misses are observed for these lines. The remaining M - L virtual cache lines from set X_s perceive the cache as a N - L set associative cache. In case the total number of virtual cache lines sharing this particular cache set is more than the associativity of the cache, which would definitely be true for large programs, this decreased associativity might result in an increased miss rate for the remaining lines.

The number of solutions to the cache-locking problem is exponential since there are an exponential number of ways to choose up to K lines to lock out of M contenders. As demonstrated by Liang and Mitra [Liang and Mitra 2010], this is a classical NP Hard combinatorial optimization problem with a complex integer linear programming based solution. Further, finding an exact solution is complicated by the fact that the increased miss rate for remaining M - L virtual cache lines cannot be accurately determined unless we know which virtual cache lines are locked in the current set of the cache, which is one of the objectives of this optimization problem. In addition, Liang and Mitra [Liang and Mitra 2010] demonstrated that the approximate solution to this problem performs almost equivalent to the optimal solution at a much lower computational cost. Consequently, we explore an approximate solution for this problem.

6.1. Cache Locking Algorithm

Here, the solution for one cache set is considered in detail; the same method is employed repeatedly for each set. Our solution is based upon the total time taken to access each virtual cache line during the lifetime of the program. We introduce a time model for representing the total time taken to access a particular virtual cache line during the lifetime of the program in presence of locking.

LOCKLIST: The running list of virtual cache lines locked so far in the set.
LL: The number of elements in list LOCKLIST.
$\texttt{HIT}_{\texttt{LOCKLIST}}(x_i)/\texttt{MISS}_{\texttt{LOCKLIST}}(x_i)$: Total number of hits/miss obtained for a virtual cache line x_i during the lifetime of the program assuming that the lines in list LOCKLIST were locked in the current set.
$F(x_i)$: The total number of accesses to a virtual cache line x_i.
T_{HIT}/T_{MISS}: Hit and Miss latency of the cache expressed in processor cycles.

Mathematically, this model is described as

$$\begin{aligned} Time(x_i|LOCKLIST) = HIT_{LOCKLIST}(x_i) * T_{HIT} \\ + \quad MISS_{LOCKLIST}(x_i) * T_{MISS} \end{aligned} \tag{3}$$

In our notation, `Time(A|B)` is the total time to access virtual cache line `A` during the lifetime of the program given that all the virtual cache lines in mathematical set `B` have already been locked in `A`'s cache set. This notation is borrowed from conditional probability. `LOCKLIST` is initialized as an empty list. Every time a line is selected to be locked, the `LOCKLIST` is updated with the line. The analysis presented is only applied to $x_i \notin$ `LOCKLIST`.

In order to find the virtual cache lines which should be locked in this set, we introduce a cost-benefit model based on the above time model to find the net benefit (benefit - cost) of locking a particular cache line. The following relation between number of accesses, number of hits and number of misses always holds true, irrespective of the lines currently locked (`LOCKLIST`) in the set:

$$F(x_i) = HIT_{LOCKLIST}(x_i) + MISS_{LOCKLIST}(x_i) \ \ \forall \ LOCKLIST \tag{4}$$

Using the above relation and the time model from equation (3), the original access time for virtual cache line x_i, assuming that virtual cache lines in `LOCKLIST` are already locked in this set, can be represented as:

$$\begin{aligned} Time(x_i|LOCKLIST) = HIT_{LOCKLIST}(x_i) * T_{HIT} \quad + \\ (F(x_i) - HIT_{LOCKLIST}(x_i)) * T_{MISS} \end{aligned} \tag{5}$$

If line x_i is locked in cache, only one miss (a compulsory miss) would be observed for this line. All the remaining accesses to this line would definitely result in a hit. Thus the new access time for this line would be given by following relation:

$$\begin{aligned} Time(x_i|(LOCKLIST + x_i)) = T_{MISS} \quad + \\ (F(x_i) - 1) * T_{HIT} \end{aligned} \tag{6}$$

Subtracting equation (6) from equation (5), the potential benefit of locking a particular line x_i can be expressed as:

$$\begin{aligned} BenLock(x_i) \ &= \ Time(x_i|LOCKLIST) \\ &- \ Time(x_i|(LOCKLIST + x_i)) \\ &= \ (F(x_i) - HIT_{LOCKLIST}(x_i) - 1) \\ &* \ (T_{MISS} - T_{HIT}) \end{aligned} \tag{7}$$

In order to calculate the cost of locking a line, we only consider the opportunity cost of locking a line and not the actual cost of executing locking. Since we are just considering a static solution, the cost of executing a single locking instruction is negligible and hence does not affect our analysis.

In order to represent the opportunity cost of locking a particular cache line, we need to model the increase in total access time for the remaining virtual cache lines which map to the set under consideration. So far, |`LOCKLIST`| = `LL` virtual cache lines have been selected for locking. Let, $X_s^{LOCKLIST}$ denotes the set of virtual cache lines mapped to the current cache set `s`, excluding the elements in the list `LOCKLIST`. This set can be

computed as follows

$$X_s^{LOCKLIST} = X_s - LOCKLIST \tag{8}$$

The elements in LOCKLIST are already locked in cache, hence they won't observe any opportunity cost. According to above terminology, each line $x_j \in X_s^{LOCKLIST}$ observes $HIT_{LOCKLIST}(x_j)$ hits. Each element belonging to set $X_s^{LOCKLIST}$ is a potential candidate for locking. If line x_i is locked at this step, then each remaining element x_j of set $X_s^{LOCKLIST}$ would observe a lesser number of hits, denoted by $HIT_{LOCKLIST\ +x_i}(x_j)$. This constitutes the cost of locking a particular line x_i. Mathematically, for each $x_j \in X_s^{LOCKLIST}$, the original access time is represented by equation (5). The new access time after locking line x_i can be represented as:

$$\begin{aligned} Time(x_j|(LOCKLIST + x_i)) = HIT_{LOCKLIST+x_i}(x_j) * T_{HIT} \quad + \\ (F(x_j) - HIT_{LOCKLIST+x_i}(x_j)) * T_{MISS} \end{aligned} \tag{9}$$

The increase in access time for one element x_j due to locking the line x_i, denoted by $CostLock_{x_i}(x_j)$, can be represented as

$$\begin{aligned} CostLock_{x_i}(x_j) &= Time(x_j|(LOCKLIST + x_i)) \\ &- (Time(x_j|LOCKLIST) \\ &= (HIT_{LOCKLIST}(x_j) - HIT_{LOCKLIST+x_i}(x_j)) \\ &* (T_{MISS} - T_{HIT}) \end{aligned} \tag{10}$$

The total cost of locking the line x_i can be represented as

$$CostLock(x_i) = \sum_{(x_j|x_j \in X_s \wedge x_j \neq x_i)} CostLock_{x_i}(x_j) \tag{11}$$

The net benefit of locking a particular virtual cache line can be calculated as

$$NetBenefit(x_i) = BenLock(x_i) - CostLock(x_i) \tag{12}$$

A positive NetBenefit for a cache line implies that locking this line would result in a lesser total memory access time for the program. Magnitude of the NetBenefit represents the change in total access time. Thus the cache line with maximum positive benefit is the ideal candidate for locking at this step.

In the above cost-benefit model, determining the exact value of $HIT_{LOCKLIST+x_i}$ is completely infeasible given that the number of profile runs needed would equal the number of virtual cache lines. Thus, an approximate value of $HIT_{LOCKLIST+x_i}$ is obtained by locking a dummy (unused) virtual cache line in the set apart from LL lines already locked. Hence,$HIT_{LOCKLIST+x_i}$ is approximated by computing $HIT_{LOCKLIST+\{Dummy\}}$. Nevertheless, this approximation always provides conservative estimates for future hit rate – in reality, one less virtual cache line would be competing for space in cache – and thus locking a line is guaranteed to show performance improvement.

In order to meet both the objectives of the problem – determining the number of cache lines to be locked in the set and selecting the virtual cache lines to be locked in these lines of the set – we devise a greedy and iterative solution for this problem. Let us examine the steps taken at the $(LL + 1)^{th}$ iteration. At this point, we have a list LOCKLIST of LL virtual cache lines which should be locked in the set. The above model is used to calculate the NetBenefit for each of the virtual cache line $x_i|x_i \in X_s^{LOCKLIST}$. If the net-benefit is negative for all the elements, the locking is discontinued for this set,

N: Cache size in number of lines
K: Number of lines which can be locked in each set
S: Number of sets in the cache.
s_i: Set where memory address x_i gets mapped
X_{s_i}: The set of memory addresses which get mapped to set s_i
$F(x_i)$: Total number of memory accesses to address x_i
LL: Iterator over number of lines locked in one set
$HIT_{LOCKLIST}(x_i)$: Total number of hits obtained for x_i when lines in set $LOCKLIST$ are locked
$MISS_{LOCKLIST}(x_i)$:Total number of miss obtained for x_i when lines in set $LOCKLIST$ are locked
$LOCKLIST(s_i)$:Set of virtual cache lines which should be locked in set s_i; initialized as empty
$NumLockLines(s_i)$: Number of virtual cache lines which should be locked in set s_i.
T_{HIT} / T_{MISS} : Hit/Miss latency in processor cycles

void **Cache_Locking_Algorithm**() :
1. for(each set s_i in range 0 to S -1) do :
2. for(each LL in range 0 to K -1) do :
3. for (each x_i in X_{si}) :
4. $BenLock(x_i) = (F(x_i) - HIT_{LOCKLIST}(x_i) - 1) * (T_{MISS} - T_{HIT})$
5. $CostLock(x_i) = \sum_{x_j | x_j \in X_{s_i} \& x_j \neq x_i} CostLock_{x_i}(x_j)$
6. $NetBenefit(x_i) = BenLock(x_i) - CostLock(x_i)$
7.
8. If there exists a x_k such that $NetBenefit(x_k)$ is maximum and is positive.:
9. Add x_k to $LOCKLIST(s_i)$
10. $NumLockLines(s_i) = NumLockLines(s_i) + 1$
11. $X_{s_i} = X_{s_i} - x_k$
12. else:
13. break;

Fig. 3: Static Cache Locking Algorithm.

implying that it is not beneficial to lock any more cache line in this set. The running list LOCKLIST represents the final list of virtual cache lines which should be locked in this set. If there is at least one element with positive net-benefit, we find the virtual cache line which has maximum net benefit for locking. This line is added to the list LOCKLIST. The above steps are repeated at each iteration until at least one of the following two conditions is true: (i) we reach the limit of maximum cache lines which can be locked in a set or (ii) we reach a point where the net benefit becomes zero for each virtual cache line in this set. At the end of this process, we get the number of cache lines (|LOCKLIST|) as well as memory addresses which should be locked in this set (LOCKLIST).

Figure 3 describes the pseudocode for the cache locking algorithm. The iterative method for cache locking is applied exactly as described earlier. First, the instruction trace of the application is obtained using a processor simulator. Next, this instruction trace is used to obtain cache statistics using a cache simulator. This cache simulation is iteratively applied with increasing number of lines locked per set. Specifically, at each iteration, the cache simulation is repeated for two cases: first by locking the the lines in LOCKLIST; second, by locking the lines in LOCKLIST and an additional dummy line in each set of the cache. The data from these two profile runs is used to identify the line to be locked during current iteration and LOCKLIST is updated accordingly. The iterations are continued until no more lines can be profitably locked.

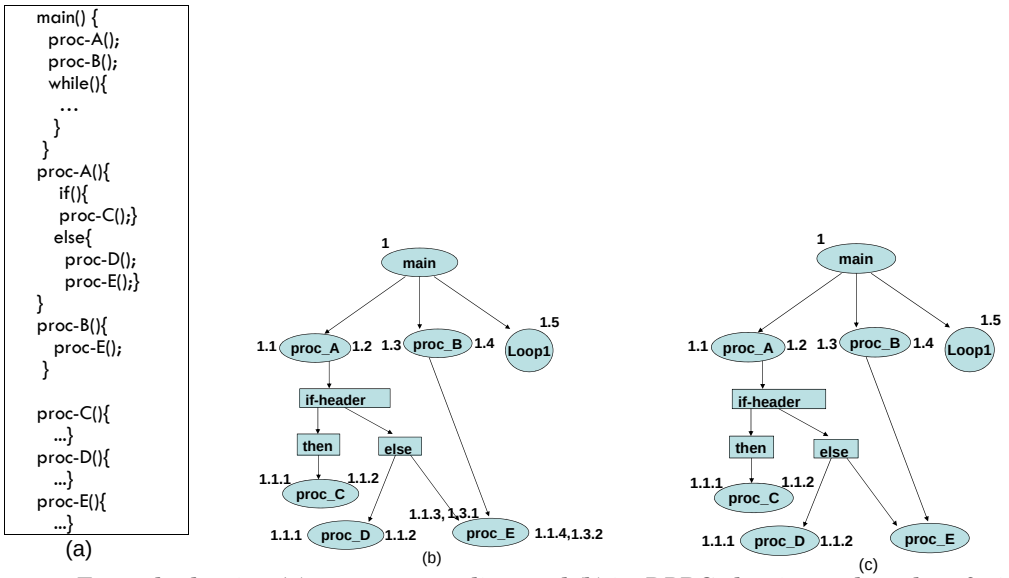


Fig. 4: Example showing (a) a program outline; and (b) its DPRG showing nodes, edges & timestamps (c) modified DPRG nodes and timestamps assuming that execution frequency of proc_E is greater than LIMIT.

7. DYNAMIC CACHE LOCKING

In this section, we discuss our mechanism for dynamically updating the contents locked in a cache. The primary goal is to divide the program into regions to enable solving the problem on multiple smaller regions as compared to the complete program.

The solution consists of the following steps. First, the program code is analyzed to determine a set of promising program points. Second, the program points with high execution frequency are discarded since they will result in high locking overheads. Third, a code region is defined by the code blocks lying between two consecutive program points. Regions correspond to the granularity at which cache locking decisions are made. Next, a heuristic based algorithm is employed to compute the locked content in the cache in each region. The required locking instructions are inserted at the program point corresponding to the beginning point of each region. The cache content is fixed in a particular region, but may change at region boundaries. We first describe our method for determining such program points (and regions) and then propose our solution to determine the locked cache content in each particular region.

7.1. Program Points

The choice of program points is critical to the success of the algorithm. Promising program points are those after which the program has a significant change in locality behavior. Further, the dynamic frequency of program points should be less than the frequency of regions, so that the cost of executing cache locking instructions can be recouped by corresponding improvement in memory performance. Hence, sites just before the start of loops are especially promising program points since they are infrequently executed compared to the insides of loops. Moreover, the loop often re-uses instructions, justifying the cost of locking lines in the cache.

With the above considerations, we employ a modified version of Data-Program Region Graph (DPRG), proposed by Udayakumaran and Barua [Udayakumaran et al. 2006], for determining program points. We modify the original DPRG struc-

ture [Udayakumaran et al. 2006] in three aspects. First, the original DPRG was proposed to solve the data allocation problem, hence it also represents variable accesses. We do not need to represent variables since we are only targeting instruction cache locking. Second, we discard the program points obtained by if-then-else constructs to limit our decision points since these control constructs do not capture change in locality as effectively as loop structures [Hwu and Chang 1989]. Third, we refine the program points obtained through original DPRG structure by discarding the program points with high execution frequency, since locking at those locations results in high overheads. The threshold for refining the program points is heuristically determined using profiling, as explained in later sections. Below, we summarize the DPRG structure and our modifications to this structure.

DPRG defines program points as (i) the start and end of each procedure; (ii) just before each loop (even inner loops of nested loops). In this way, program points track most major control-flow constructs in a program. Program points in a DPRG are the only initial candidate sites for applying cache locking in the ensuing region. This set is further refined and the actual solution regarding the elements to be locked in each region is governed by the method proposed in Section 7.2.

Figure 4 shows an example illustrating how a program points are marked with timestamps. Figure 4(a) shows the outline of an example program. It consists of six procedures, namely `main()`, `proc-A()`, `proc-B()`, `proc-C()`, `proc-D()` and `proc-E()`, one loop, `Loop1` and one `if-then-else` construct.

Figure 4(b) shows the Data Program Relationship Graph (DPRG) for the program in Figure 4(a). DPRG helps in the marking of timestamps and the identification of regions. DPRG is essentially the programs call graph appended with new nodes for loops. Similar to a control-flow graph or a call-graph, DPRG places the control-flow contructs appearing in a program in a left-to-right order. In the DPRG shown in Figure 4(b), there are six procedures, one loop and one if-then-else construct. Separate nodes are shown for the entire if statement (called if-header) and for its then and else parts. Edges to procedure nodes represent calls while edges to loop nodes shows that the loop is nested in its parent. The program points – namely the starts and ends of procedures and loops – are represented by the start of the code in each node. In case of a loop, its program point is outside the loop at its start. Program point corresponding to the beginning of a procedure is inside its body at the start and the program point corresponding to the end of a procedure is present at corresponding callsites.

Figure 4(b) also shows the timestamps (`e.g` 1.1, 1.2) for each program point in the DPRG. As presented by Udayakumaran and Barua [Udayakumaran et al. 2006], the traditional depth-first-traversal is modified in three ways to derive the timestamps. First, then and else nodes of if statements are not assigned any numbers. Second, the children of then and else nodes are assigned the same starting number since only one part is executed per invocation. For example, the children of then and else nodes shown in the figure are both marked with timestamp 1.1.1. Third, it traverses and timestamps nodes every time they are seen, rather than only the first time. For example, `proc-E` is assigned multiple timestamps.

This initial set of program points is further refined by considering the actual execution frequency at each program point. We define a threshold `LIMIT` and discard the program points whose execution frequency is greater than `LIMIT` since the overhead of locking at those points might be too high. The actual value of `LIMIT` is determined through heuristics, as discussed in Section 9. In order to simplify the determination of regions, the program points in the subgraph rooted at such program points are also discarded and the whole subgraph is considered as a single node.

Figure 4(c) shows the resulting program points and corresponding timestamps after applying the above refinement. Each program point in this modified graph denotes

the beginning point of a region. The portions of code lying between two consecutive program points is considered a region. For example, the code lying between main and call to proc-A is considered part of one region. A region can have multiple terminating program points. In Figure 4(c), the code blocks between program points 1.1 (beginning of proc-A) and 1.1.1 (beginning of proc-C and proc-D) is considered a single region.

In our method, timestamps inside the then part of an if-else construct cannot be compared with the timestamps inside the else part of the same if-else construct. Hence timestamps specify only a partial order between regions, not a total order. In such cases, the program point just outside the header is considered the next program point. For example, the code lying between the end of call to proc-C and the end of proc-A is considered a region.

As evident, a code block can simultaneously be part of multiple different regions. For example, proc-E has no corresponding program point since its execution frequency exceeds the threshold. Hence, the code block inside proc-E is considered part of two regions - region between the beginning and end-point of proc-B as well as region between the end of call to procedure proc-D in proc-A and the end of proc-A. The locked content in such code blocks, which are part of multiple regions, is dynamically governed as per the actual program region of execution. If proc-E is executed through the call-site in proc-B, then the contents are determined by the program point at entry of proc-B. On the other hand, if proc-E is executed through the call-site in proc-A, then the program point at end of call to proc-D defines the actual contents at the time of execution.

Although it seems timestamps approximate run-time order, our method does not use the order of timestamps in any way. Instead, once the timestamps are marked, regions are processed independently of each other in any order. The locking at the start of each region only considers the characteristics of that region, and does not depend on the order of timestamps. Since our method only considers the locations of timestamps and ignores their values; hence multiple timestamps at any location with different values are treated the same as a single timestamp at that location.

7.2. Dynamic Locking Algorithm

The above method divides a program into a set of regions, where the program locality is consistent in a region. Each region can be analyzed separately since only one region is active at an execution instant. Consequently, the solution for cache locking in each region is obtained by extending the static cache locking algorithm proposed in Section 6 with some modifications, as discussed next. We introduce the following definitions to ease the description. Similar to static cache locking method, the formulations are for one particular cache set s.

R: Set of program regions; r: An element of set R; p: A program point;
$Exec_p$: Execution frequency of point p; LockInst: Number of cycles for locking a line
Y_r: Set of virtual cache lines accessed in a region r mapped to a particular cache set s[1]
L_r :Lines which should be locked in a set s in a region r

The dynamic cache locking problem has two objectives (i) For each region r, determining L_r (ii) selecting $|L_r|$ virtual cache lines out of $|Y_r|$ candidates which should be locked in the set in this region.

The benefit for locking a line in a region r is same as static cache locking and is given by Equation 7. However, the cost for locking a line in a region is influenced by two

[1]Since we are considering solution independently for each cache set, we have simplified the notation by ignoring the set representation s in the notation for Y_r and L_r

N: Cache size in number of lines; K: Number of lines which can be locked in each set
S: Number of sets in the cache.
R: Set of regions determined by DPRG
Y_r: The set of memory addresses which get mapped to set s_i in a region r
$F(x_i)$: Total number of memory accesses to address x_i
LL: Iterator over number of lines locked in one set in region r
$HIT^r_{LOCKLIST}(x_i)$: Number of hits obtained for x_i when lines in set $LOCKLIST$ are locked in region r
$LOCKLIST^r(s_i)$: Set of virtual cache lines which should be locked in set s_i in region r
$NumLockLines^r(s_i)$:Number of virtual cache lines which should be locked in set s_i in region r
T_{HIT} / T_{MISS} : Hit/Miss latency in processor cycles
$LockInst$: Number of cycles for locking a lines; $\texttt{Exec}_\texttt{r}$: Execution frequency at beginning of region r

void **Dynamic_Cache_Locking_Algorithm**() :
1. for (each region r in set R) do :
2. for (each set s_i in range 0 to S -1) do :
3. for(each LL in range 0 to K -1) do :
4. for (each x_i in Y_r) :
5. $BenLock(x_i) = (F(x_i) - HIT^r_{LOCKLIST}(x_i) - 1) * (T_{MISS} - T_{HIT})$
6. $OppCostDynLock(x_i) = \sum_{(x_j|x_j \in Y_r \& x_j \neq x_i)} CostLock_{x_i}(x_j)$
7. $CostDynLock(x_i) = OppCostDynLock(x_i) + LockInst * \texttt{Exec}_\texttt{r}$
8. $NetBenefit(x_i) = BenLock(x_i) - CostDynLock(x_i)$
9.
10. If there exists a x_k such that $NetBenefit(x_k)$ is maximum and is positive.:
11. Add x_k to $LOCKLIST^r(s_i)$
12. $NumLockLines^r(s_i) = NumLockLines^r(s_i) + 1$
13. $Y_r = Y_r - x_k$
14. else
15. break;

Fig. 5: Dynamic Cache Locking Algorithm.

factors. First, only the program addresses belonging to region r need to be considered for computing the opportunity cost. Second, the locking instructions are now executed each time a program point is executed. Hence, the cost of locking is no more negligible and the model needs to reflect the locking cost.

The opportunity cost of locking a single element is given by Equation 10. However, this opportunity cost is observed only by the elements belonging to region r. Hence, the new opportunity cost is given by the following equation:

$$OppCostDynLock(x_{r_i}) = \sum_{(x_j|x_j \in Y_r \& x_j \neq x_i)} CostLock_{x_i}(x_j) \quad (13)$$

Further, the total cost, considering the actual cost of locking, is defined as follows:

$$CostDynLock(x_{r_i}) = OppCostDynLock(x_{r_i}) + \texttt{LockInst} * \texttt{Exec}_\texttt{p} \quad (14)$$

Hence, the net benefit of locking a line can be denoted as

$$NetBenefit(x_{r_i}) = BenLock(x_i) - CostDynLock(x_{r_i}) \quad (15)$$

The above net benefit heuristic is applied in each region to determine the set of lines to be locked in each region. Fig 5 presents the pseudo code for dynamic cache locking algorithm.

8. IMPLEMENTATION

In this section, we discuss the implementation of binary rewriting scheme for instruction cache locking. Figure 6 presents an overview of our experimentation workflow. Our mechanism for cache locking can be implemented inside any existing binary rewriting framework such as Diablo [De Sutter et al. 2005] or SecondWrite [Anand et al. 2013].

First, the binary rewriter framework is employed to obtain an intermediate representation (IR) from the input binary. Next, the techniques presented in Section 7.1 are employed to determine the DPRG regions/program points in the input binary. Next, the IR is instrumented with dummy placeholders at these program points and the binary rewriter's backend is employed to obtain an instrumented binary.

The instrumented binary is used to obtain an instruction trace of the application using a processor simulator (details below). Next, this instruction trace is used to obtain cache statistics using a cache simulator. This cache simulation is iteratively applied with increasing number of lines locked per set to determine the final list of virtual cache lines to be locked in the cache in each program region.

As mentioned in Section 5, the required locking instructions are inserted in the program using a trampoline mechanism. After determining the addresses to be locked in the cache, binary rewriter is employed again to insert the actual lock instruction corresponding to the required lines in a trampoline. The unlocking operation of the complete cache can be carried out using a single instruction [ARM 2007]. Hence, each trampoline contains a single unlocking instruction at the beginning to unlock the lines from the previous region.

The workflow in Figure 6 corresponds to dynamic locking. In case of static locking, a single placeholder is inserted only at the beginning of the binary, instead of determining the placeholders using DPRG, but otherwise the workflow is similar. This approach enables the application of our cache locking method directly to binaries.

9. RESULTS

The experiment setup consists of a Intel XScale processor core with clock frequency 600 Mhz (PXA27x family), on-chip 16 kB 4-way set-associative data cache, on-chip instruction cache and a unified off-chip memory. The ARMulator software, which is part of ARM Development Suite is used to simulate the processor core. The above

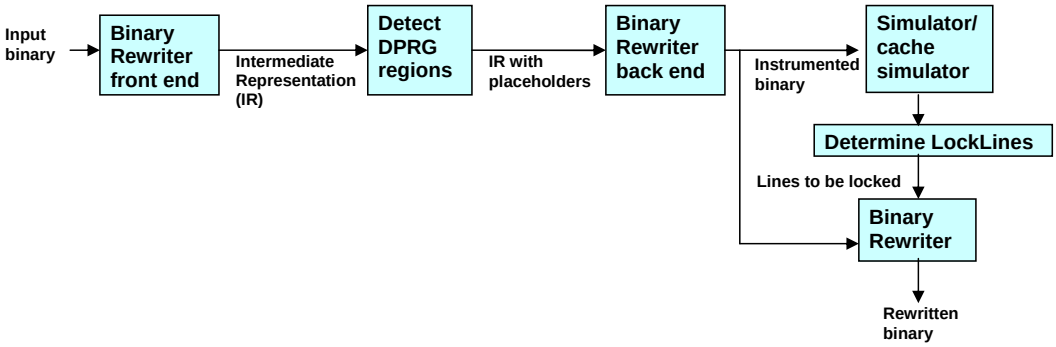


Fig. 6: The Experimental WorkFlow.

architectural parameters can be easily configured in ARMulator. Dinero IV [Edler and Hill 2004], a well-known cache-hierarchy simulator is used to simulate the cache. We modified Dinero to provide the cache statistics at the granularity of virtual cache lines and augmented it with the ability to simulate cache locking.

We configured the ARMulator to simulate a perfect zero-wait memory system. It generates the execution time in terms of processor cycles. The instruction and data miss statistics provided by Dinero are used to calculate the effect of cache misses on execution time and is added to the execution time calculated by ARMulator to obtain the total execution time of the application. A sample memory map file available in ARMulator with average off-chip memory access time of 150ns is chosen to calculate off-chip memory access latency. As per the XScale's architecture manual, each locking instruction is assumed to take four cycles and is considered accordingly while calculating the execution time of resulting binary.

A subset of MiBench benchmarks were selected to substantiate the performance improvement obtained by our method of cache locking. At this point we have simply included all the benchmarks that compiled and ran in our infrastructure in the time available – the benchmarks have not been selected to be favorable to us in any way. Table I lists the benchmarks which are used for carrying out the experiments. All the benchmarks are compiled for 32-bit ARM instructions using the GNU-ARM Baremetal toolchain version 4.3.3 with full optimizations and static linking of libraries.

The results for static cache locking are presented in Section 9.1 while Section 9.2 presents further improvement obtained by dynamic cache locking mechanism. Section 9.2 also compares our static and dynamic mechanisms with the static mechanism suggested by Liang and Mitra [Liang and Mitra 2010]. We refer to the method suggested in [Liang and Mitra 2010] as `OPT-static`. Based on the execution frequency of program regions, the value of `LIMIT` (Section 7.1) was kept to 50 in our experiments.

9.1. Static Cache Locking

Various kinds of experiments are performed with different cache configurations for analyzing the improvement in the instruction-cache miss rate and runtime of the applica-

Application	Lines Of Code	Num of Instr	Num of DynInstr
Sha	207	2501	355452842
Crc	128	1027	75738737
BitCnts	543	3340	149409187
Susan	1456	4040	60516192
Blowfish	3260	2909	868261350
Jpeg	19804	9718	104615385
Dijkstra	268	18612	536074136
Lame	15959	19810	569002359
Gsm	4779	14040	64340338
StringSearch	3072	1839	8051466
QuickSort	79	2298	830913008
Lout	30689	59828	538663
FFT	278	5868	671496345
BasicMath	7367	6375	102147075
Patricia	296	7756	114446172
Rinjdal	1017	4960	578559602

Table I: Application-Table

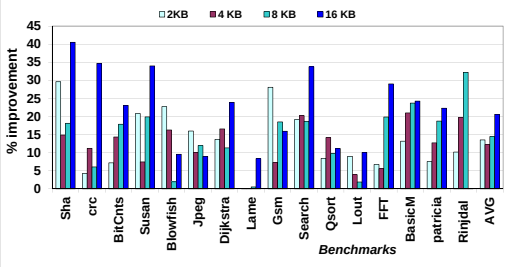


Fig. 7: Percentage improvement in instruction-cache miss rate over cache with no locking for varying sizes of a 2-way set-associative cache.

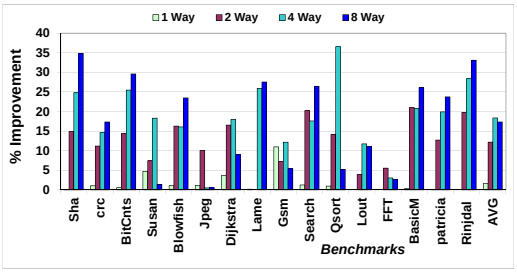


Fig. 8: Percentage improvement in instruction-cache miss rate over cache with no locking for different associativities of the cache. The cache size is kept fixed at 4 Kb.

tions. The cache configuration is varied across two dimensions: size and associativity. The block size is kept fixed at 4 words.

The percentage improvement in the I-cache miss rate with static cache locking compared to the cache configuration without locking is displayed in Figure 7 for different cache sizes. As evident from this figure, the proposed I-cache locking mechanism results in a consistent improvement in the instruction cache miss rate over all the benchmarks and cache sizes. We obtain an average improvement of 15% in the I-cache miss rate for small cache sizes and around 25% for large cache sizes. Interestingly, the improvement in the I-cache miss rate increases with an increase in the cache size for most of the applications. This is expected since a small cache size results in a high opportunity cost in our cost-benefit model as locking a line prevents many other lines from accessing that cache location, resulting into fewer lines being locked in the cache.

Figure 8 displays the variation of I-cache miss rate improvement with variation in associativity of the cache. We see that the improvement in the I-cache miss rate ranges from 15-18% for set-associative caches. Virtually all commercial cached embedded processors support only set-associative caches [2], which establishes our proposed approach as a robust mechanism for improving memory system performance. The average improvement in case of direct mapped cache is, not surprisingly, limited – having only one way in a set amounts to extremely high opportunity cost resulting in very little locking. Our goal is not to get improvements in direct-mapped cache – we never expected to, and such caches are very rare in embedded systems – but the results are presented for completeness, and show that the method never degrades performance, even managing a small improvement for direct mapped caches[3].

Next, the impact of instruction cache locking on runtime performance of various applications is analyzed. Figure 9 shows the savings in runtime obtained by using the instruction cache-locking. Comparing Figure 7 and Figure 9 brings out several interesting observations.

First, even though the cache locking scheme results in considerable improvement of instruction cache miss rate consistently over all the applications, not all applications experience an improvement in runtime performance. The improvement in I-cache miss rates translates to runtime performance improvement only for those applications where the overall I-cache miss rate is high. This is not surprising since a technique

[2]For example, among the ARM processors, only one of the 15 processors listed on ARM's website offers a direct-mapped cache.

[3]For simulation purposes, the architectural constraint of not locking way 0 is relaxed for direct mapped cache.

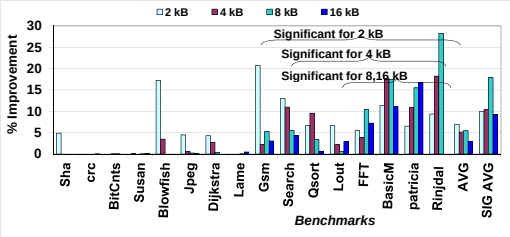


Fig. 9: Improvement in execution time of the applications over cache with no locking for varying size of a 2-way set associative cache

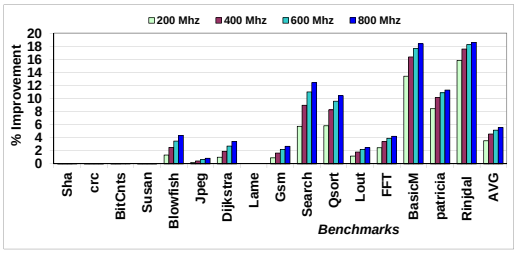


Fig. 10: Variation of execution time improvement for processors with different clock speeds for a 4kB 2 way set associative cache.

like ours to reduce the I-cache miss rate will not help if it is not a problem to begin with.

Revisiting Figure 9, we see that the benchmarks on the right-hand side are marked "significant" for different cache sizes. These are the benchmarks that have a significant I-cache miss rate (which we define as $> 1.5\%$) for that cache size. For the benchmarks with significant I-cache miss rate, the runtime improvement from our cache locking method averages 11.5% for a cache size of 8kB. The averages are shown in Figure 9 and Figure 10 in the last two columns as AVERAGE and SIG-AVERAGE, for all the benchmarks, and those with significant miss rates, respectively. For the benchmarks with very low I-cache miss rates, the benefits from cache locking are, not surprisingly, low – only 1.7 % for a 2kB cache.

The 11.5% runtime improvement with cache locking for benchmarks with a significant I-cache miss rate is encouraging and shows the benefit of our method. For some benchmarks the benefit is even higher – e.g, the Rinjdal benchmark has a runtime gain of 23.5%. Overall we see that about 20-60% of the benchmarks show significant improvement, depending on the cache size and associativity used. The fact that not all benchmarks benefit from cache locking is not an indictment against our method – indeed there is a long history of research into techniques that benefit only a class of applications. For example, faster garbage collectors only benefit benchmarks with heap data, and among those, only those with significant garbage. Nonetheless, garbage collection is still worthwhile. As classes of applications go, benefiting 20-60% of benchmarks significantly is quite good.

Further, we observe that although increasing cache size results in better performance in terms of instruction cache miss rate reduction, the average percentage improvement in execution time decreases with an increase in cache size. An increase in the cache size results in a lower initial miss rate and thus yields smaller runtime benefits from locking.

Next, in order to analyze the applicability of our approach for different processor generations, we analyze the improvement in execution time for various processor frequencies. We vary the processor clock speed while keeping the DRAM latency constant in nanoseconds – this is equivalent to varying the DRAM latency in cycles. We obtain a consistent improvement in execution time with an increase in processor frequency, as displayed in Figure 10. Thus our method can be applied effectively for different generations of processors.

9.2. Dynamic Cache Locking

In this section, we present the results for our dynamic cache locking algorithm and compare the results with our static algorithm (`Static` in figures) as well as the `OPT-static` algorithm suggested by Liang and Mitra [Liang and Mitra 2010].

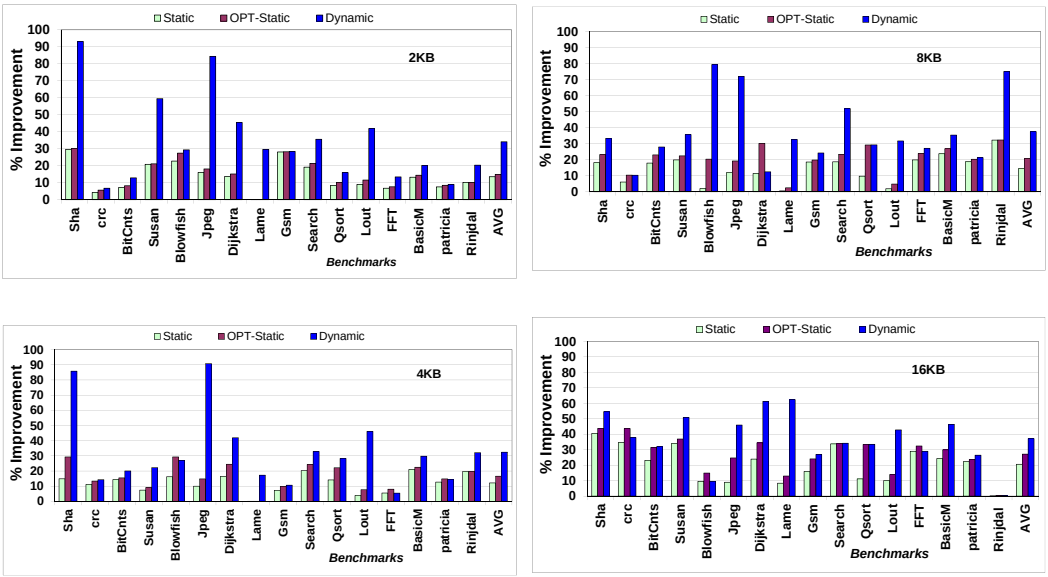


Fig. 11: Percentage improvement in instruction-cache miss rate, compared with static cache locking, for a 2-way set-associative cache of size 2 kb,4 kb, 8 kb and 16 kb.

Figure 11 presents the percentage improvement in the I-cache miss rate with dynamic cache locking, as compared to both static algorithms, for different cache sizes. As evident from this figure, the dynamic I-cache locking mechanism results in a consistent improvement in the instruction cache miss rate over all the benchmarks and cache sizes. We obtain an average improvement in the range of 35% to 40% in the I-cache miss rate for all cache sizes.

Figure 11 shows that `OPT-static` obtains a slightly better I-cache miss rate than our static algorithm, thereby revalidating the results presented in [Liang and Mitra 2010]. However, our dynamic mechanism consistently results in a better I-cache miss rate than both static methods. We also notice a few scenarios (blowfish - 16 kB, dijkstra - 8 kB) where our dynamic version performs worse than `OPT-static` algorithm. We believe this can be overcomed by actually applying `OPT-static`, instead of applying our static version, to determine the lines to be locked within each program region.

The results in Section 9.1 demonstrate that the improvement in the I-cache miss rate due to static cache locking increases with an increase in the cache size for most of the applications. Figure 11 demonstrates that dynamic cache locking mechanism does not display this behavior. It results in a consistent improvement of around 40% across all cache sizes. This is not surprising since the dynamic mechanism overcomes the inherent opportunity cost involved in the static locking mechanism by dynamically adpating the cache content with program demand. An interesting corollary of this result is that the dynamic mechanism is much more effective for smaller cache sizes. For example, for cache size of 4 kB, the dynamic method improves the cache miss rate by 35% as compared to 15% by static method whereas in case of a 16 kB cache, the relative improvement is 37% over 27% improvement obtained by static mechanism.

Figure 12 displays the variation of I-cache miss rate improvement with variation in associativity of the cache. We see that the improvement in the I-cache miss rate due to dynamic cache locking ranges from 32-35% for different associativity of set associa-

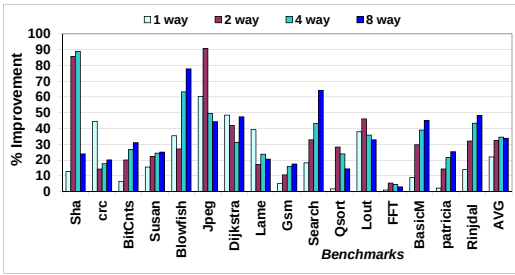


Fig. 12: Percentage improvement in instruction-cache miss rate with dynamic cache locking over cache with no locking for different associativities of the cache. The cache size if kept fixed at 8 kb.

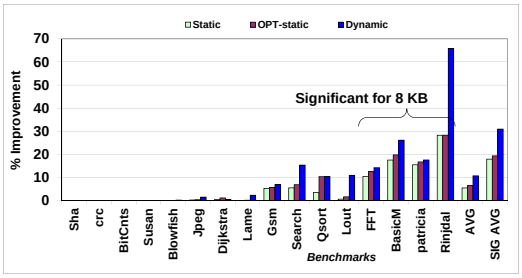


Fig. 13: Improvement in execution time of the applications with dynamic cache locking, as compared with static and optimal static cache locking, for varying size of a 8 kb 2-way set associative cache.

tive caches, as compared to 15-18% improvement obtained by static mechanisms. An interesting feature is that dynamic cache locking is also able obtain 20% performance improvement for direct mapped caches. Recall from Section 9.1, static cache locking was mainly effective for set associative caches. This is due to the reduced opportunity cost in dynamic locking models.

Next, the impact of instruction cache locking on runtime performance of various applications is analyzed. Figure 13 shows the reduction in runtime by using dynamic instruction cache-locking for a particular cache configuration, as compared to static algorithms. Similar to Fig 9, we average the improvement in execution time for all the benchmarks as well as the benchmarks with significant initial miss rate. Figure 13 demonstrates that the improvement in miss rate obtained by dynamic algorithms translate effectively to an improvement in execution time. For benchmarks with a significant I-Cache miss rate, the dynamic mechanism improves execution time by 20% on average as compared to 11.5% and 12.5% obtained by static and `OPT-static` mechanisms respectively.

10. CONCLUSION

In this paper, we have presented an instruction cache locking mechanism for improving the average-case run-time of embedded systems, extending the applicability of cache locking beyond real-time systems. Our instruction-cache-locking scheme is implemented inside a binary rewriter, implying that our scheme can be applied to binaries compiled using any compiler and to legacy codes whose source code is not available. Our results indicate that on average, the proposed cache locking scheme achieves a 35% improvement in instruction cache miss rate and a 32% improvement in run-time performance of instruction cache-constrained applications.

In future work, we plan to extend the cache locking mechanisms for the data cache to further improve the run-time performance of applications.

ACKNOWLEDGMENTS

The authors would like to thank Mr. Joe Bungo from ARM Inc for providing access to the ARM development suite and providing technical guidance regarding the toolchain.

REFERENCES

Kapil Anand and Rajeev Barua. 2009. Instruction cache locking inside a binary rewriter. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems (CASES '09)*. ACM, New York, NY, USA, 185–194. DOI:http://dx.doi.org/10.1145/1629395.1629422

Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 295–308. DOI:http://dx.doi.org/10.1145/2465351.2465380

ARM Revised July 2007. *ARM1156T2-S TechnicalReference Manual*. Arm. http://www.arm.com/products/-CPUs/families/ARM11Family.html.

ARM Revised March 2004. *ARM Cortex A-8 Technical reference manual*. Arm. http://www.arm.com/-products/CPUs/families/ARMCortexFamily.html.

A. Arnaud and I. Puaut. 2006. Dynamic Instruction Cache Locking in Hard Real-Time Systems. In *Proc. of the 14th International Conference on Real-Time and Network Systems (RNTS)*. Poitiers, France.

Oren Avissar, Rajeev Barua, and Dave Stewart. 2002. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems. *ACM Transactions on Embedded Systems (TECS)* 1, 1 (September 2002).

R. Banakar, S. Steinke, B-S. Lee, M. Balakrishnan, and P. Marwedel. 2002. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES)*. ACM, Estes Park, Colorado.

L.A. Belady. 1966. A study of replacement algorithms for virtual storage.. In *IBM Systems Journal*. 5:78–101.

Kristof Beyls and Erik H. D'Hollander. 2005. Generating cache hints for improved program efficiency. *J. Syst. Archit.* 51, 4 (2005), 223–250. DOI:http://dx.doi.org/10.1016/j.sysarc.2004.09.004

BlackFin April 2009. *ADSP-BF533 Processor Hardware Reference*. Analog Devices. http://www.analog.com/-static/imported
-files/processor_manuals/bf533_hwr_Rev3.4.pdf.

M. Brehob, S. Wagner, E. Torng, and R. Enbody. 2004. Optimal replacement is NP-hard for nonstandard caches. *Computers, IEEE Transactions on* 53, 1 (2004), 73–76. DOI:http://dx.doi.org/10.1109/TC.2004.1255792

Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.* 14, 4 (2000), 317–329. DOI:http://dx.doi.org/10.1177/109434200001400404

Doug Burger, James R. Goodman, and Alain Kägi. 1996. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture (ISCA '96)*. ACM, New York, NY, USA, 78–89. DOI:http://dx.doi.org/10.1145/232973.232983

A. Marti Campoy, A. Perez Jimenez, A. Perles Ivars, and J. V. Busquets Mataix. 2001. Using Genetic Algorithms in Content Selection for Locking-Caches. (2001).

Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. 2000. Application-specific memory management for embedded systems using software-controlled caches. In *DAC '00: Proceedings of the 37th conference on Design automation*. ACM, New York, NY, USA, 416–419. DOI:http://dx.doi.org/10.1145/337292.337523

Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. 2005. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst.* 27, 5 (Sept. 2005), 882–945. DOI:http://dx.doi.org/10.1145/1086642.1086645

J. Edler and M.D. Hill. Revised 2004. DineroIV Cache simulator. http://www.cs.wisc.edu/ markhill/DineroIV/.

Heiko Falk, Sascha Plazar, and Henrik Theiling. 2007. Compile-time decided instruction cache locking using worst-case execution paths. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, New York, NY, USA, 143–148. DOI:http://dx.doi.org/10.1145/1289816.1289853

Ann Gordon-Ross, Susan Cotterell, and Frank Vahid. 2002. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. *IEEE Comput. Archit. Lett.* 1, 1 (2002), 2. DOI:http://dx.doi.org/10.1109/L-CA.2002.4

Ann Gordon-Ross, Frank Vahid, and Nikil Dutt. 2005. A First Look at the Interplay of Code Reordering and Configurable Caches. In *Proceedings of the 15th ACM Great Lakes Symposium on VLSI (GLSVLSI '05)*. ACM, New York, NY, USA, 416–421. DOI:http://dx.doi.org/10.1145/1057661.1057760

W. W. Hwu and P. P. Chang. 1989. Achieving High Instruction Cache Performance with an Optimizing Compiler. *SIGARCH Comput. Archit. News* 17, 3 (April 1989), 242–251. DOI:http://dx.doi.org/10.1145/74926.74953

Timothy M. Jones, Sandro Bartolini, Jonas Maebe, and Dominique Chanet. 2011. Link-time Optimization for Power Efficiency in a Tagless Instruction Cache. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 32–41. http://dl.acm.org/citation.cfm?id=2190025.2190051

Yun Liang and Tulika Mitra. 2010. Instruction cache locking using temporal reuse profile. In *Proceedings of the 47th Design Automation Conference (DAC '10)*. ACM, New York, NY, USA, 344–349. DOI:http://dx.doi.org/10.1145/1837274.1837362

Tiantian Liu, Minming Li, and C.J. Xue. 2009. Instruction Cache Locking for Real-Time Embedded Systems with Multi-tasks. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference on*. 494–499. DOI:http://dx.doi.org/10.1109/RTCSA.2009.59

Tiantian Liu, Minming Li, and Chun Jason Xue. 2012. Instruction Cache Locking for Embedded Systems using Probability Profile. *J. Signal Process. Syst.* 69, 2 (Nov. 2012), 173–188. DOI:http://dx.doi.org/10.1007/s11265-011-0650-6

P. R. Panda, N. D. Dutt, and A. Nicolau. 2000. On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems. *ACM Transactions on Design Automation of Electronic Systems* 5, 3 (July 2000).

I. Puaut. 2002. Cache Analysis vs Static Cache Locking for Schedulability Analysis in Multitasking Real-Time Systems. In *Proc. of the 2nd International Workshop on worst-case execution time analysis, in conjunction with the 14th Euromicro Conference on Real-Time Systems*. Vienna, Austria.

I. Puaut and D. Decotigny. 2002. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *Proc. of the 23rd IEEE International Real-Time Systems Symposium*. Austin, TX, USA.

Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang. 2005. Cooperative Caching with Keep-Me and Evict-Me. In *INTERACT '05: Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures*. IEEE Computer Society, Washington, DC, USA, 46–57. DOI:http://dx.doi.org/10.1109/INTERACT.2005.7

Jan Sjodin, Bo Froderberg, and Thomas Lindgren. 1998. Allocation of Global Data Objects in On-Chip RAM. *Compiler and Architecture Support for Embedded Computing Systems* (December 1998).

S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. 2002. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society, 409.

E.S. Tam, J.A. Rivers, V. Srinivasan, G.S. Tyson, and E.S. Davidson. 1999. Active management of data caches by exploiting reuse information. *Computers, IEEE Transactions on* 48, 11 (1999), 1244–1259. DOI:http://dx.doi.org/10.1109/12.811113

Olivier Temam. 1998. Investigating optimal local memory performance. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems (ASPLOS VIII)*. ACM, New York, NY, USA, 218–227. DOI:http://dx.doi.org/10.1145/291069.291050

Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.* 5, 2 (May 2006), 472–511. DOI:http://dx.doi.org/10.1145/1151074.1151085

Xavier Vera, Björn Lisper, and Jingling Xue. 2003. Data cache locking for higher program predictability. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM, New York, NY, USA, 272–282. DOI:http://dx.doi.org/10.1145/781027.781062

Manish Verma, Lars Wehmeyer, and Peter Marwedel. 2004a. Cache-Aware Scratchpad Allocation Algorithm. In *Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society, 21264.

Manish Verma, Lars Wehmeyer, and Peter Marwedel. 2004b. Dynamic overlay of scratchpad memory for energy minimization. In *International conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS)*. ACM.

Xscale May 2007. *3rd Generation Intel Xscale Microarchitecture Developer's manual*. Intel. http://-www.intel.com/design/intelxscale/.

Hongbo Yang, R. Govindarajan, Guang R. Gao, and Ziang Hu. 2005. Improving power efficiency with compiler-assisted cache replacement. *J. Embedded Comput.* 1, 4 (2005), 487–499.

Wankang Zhao, David Whalley, Christopher Healy, and Frank Mueller. 2005. Improving WCET by Applying a WC Code-positioning Optimization. *ACM Trans. Archit. Code Optim.* 2, 4 (Dec. 2005), 335–365. DOI:http://dx.doi.org/10.1145/1113841.1113842