Instruction Cache Locking inside a Binary Rewriter *

Kapil Anand Rajeev Barua
ECE Dept, Univ of Maryland, College Park
{kapil,barua}@umd.edu

ABSTRACT

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, like 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 the first method in the literature for instruction-cache locking that is able to reduce the average-case run-time of the program. 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 and MediaBench benchmarks show up to 25% improvement in the instruction-cache miss rate on average and up to 13.5% improvement in the execution time on average for applications having instruction accesses as a bottleneck, depending on the cache configuration. The improvement in execution time is as high as 23.5% for some benchmarks.

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. The 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 the frequently used data or instructions following a fixed replacement policy.

Various different approaches have been suggested to enable software involvement in the management of the on-chip memory. One approach involves the addition of lightweight software-controlled memory like Scratchpad memory (SPM) which rely 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 like Intel's Xscale and ARM's latest cortex processors provide the 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. Thus, the software can influence the replacement decision made by the cache and thereby alleviate the potential mistakes resulting from cache hardware management. As an example, suppose 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, then locking this element in the cache will result in a better cache performance.

Current methods regarding instruction cache locking are geared towards improving real-time predictability of applications. We present the first method in literature employing instruction cache locking as a mechanism for improving the average-case run-time 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.

The rest of the paper is organized as follows. Section 2 describes the underlying cache locking inter-

*This work is supported by NSF Grant No 0720683

face. 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 as an optimization problem, presents our cost-benefit model and describes our cache locking algorithm based on the above model. Section 6 explains the experimental environment used for our research. Section 7 presents properties of our benchmarks and our method's results on them for different cache and architecture configurations. Section 8 concludes.

2 Cache Locking Interface

There are two most common kind 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 and ARM11 family of processors.

Line locking is a more fine-grained approach to cache locking. In this interface, the locking mechanism is available at the granularity of single cache line as opposed to single way in way locking. In this interface, it is possible to have a different number of locked lines in different sets of the cache. Intel's Xscale, ARM9 family and BlackFin 5xx family processors 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 like Scratchpad memory (SPM) and loop caches. Various different kind of methods have been suggested for managing the data to be placed in SPM [3, 10, 13, 14, 16]. A *loop cache* [9] 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 where the run-time behavior of applications is predictable; or to improve real-time performance. Caches are better at tracking run-time behavior; hence are widely used 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 the allocation problems for the above software-controlled memories. There are various reasons for that. First ~~of all~~, only ~~the~~ cache-aware approaches of managing ~~the~~ software-controlled memories are applicable to the cache locking problem as other approaches can't guarantee performance improvement in presence of cache. In these approaches, an additional local memory is present apart from the cache. The general approach is to assign some of the conflicting elements to these memories [16]. This allocation of the elements to these memories has no negative affect on remaining cache accesses. On the other hand, locking an element in cache limits the cache space availaible to remaining elements. This opportunity cost is not modeled by the proposed allocation methods and has been captured in the decision problem formulated in this paper. Secondly, a particular element can be placed at any location in above software controlled memory, whereas the cache hardware decides the location of each element in a cache. This results in entirely different kind of constraints for cache locking problem. The energy model in terms of cache hits and misses suggested in [16] 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 which

involve modifications to the cache hardware itself to equip software to dynamically modify cache replacement decisions. Rudolph et al [7] introduce column caching, to provide software the ability to dynamically partition the on-chip memory into scratchpad memory; Wang et al [12] proposed the 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.

Research has been carried out to exploit the cache features present in existing hardwares - locking is one such kind of feature available in modern embedded systems. Hollander et al [4] 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 hardwares.

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 [11]. 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 [2, 6, 8, 11, 15]. In contrast, our objective of cache-content selection is to improve average case run-time 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 [17] 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 doesn't 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.

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 run-time of general embedded applications, thus widening its applicability beyond hard real time systems. (ii) 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. (iii) our method has been implemented inside a binary rewriter, widening its applicability to binaries compiled using any compiler. (iv) our method has an inherent mechanism that determines the optimal number of cache lines to be locked - it does not lock each possible cache line, as suggested by some previous methods. (v) cache locking is already available on existing hardwares 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. For ease of explanation, the instruction cache is a tiny 16-byte direct mapped cache with one word per line. The addresses are mapped to the cache lines according to the stan-

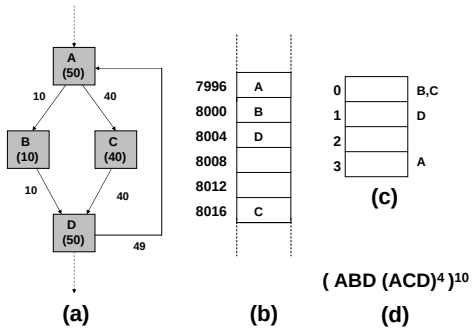
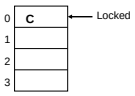


Figure 1: (a) Weighted CFG of a small part of a program. A, B, C and D are instructions of 4 byte each (b) A hypothetical memory layout of the above instructions (c) A dummy 16-byte direct mapped instruction cache. The alphabets at right hand side of each cache line show the instructions which are mapped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

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)

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

dard modulo-based cache mapping function:

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

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 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 instruction cache-locking as an effective mechanism for reducing cache misses.

5 Cache Locking

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

5.1 Overview

The cache-locking problem involves selecting the memory addresses which should be locked in the instruction cache 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.*

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

$$VirtualCacheLine = \frac{addr}{Words\text{-}Per\text{-}Line} \quad (2)$$

The remaining analysis for cache locking is carried out in terms of virtual cache lines.

Assume a N way set-associative cache. For a particular set s, X_s denotes the set of virtual cache lines

which get mapped to this set and suppose M is the cardinality of this set ($|X_s|$). In other words, M virtual cache lines share this particular cache set. Let K be the maximum number of lines which the hardware allows to be locked in one set of the cache ($K \leq N$).

The Cache-locking problem has two objectives (i) determining L ($L \leq K$): 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. In all likelihood, this is a classical NP Hard combinatorial optimization problem, which does not have an exact solution, although we have not attempted to formally prove this. 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. Hence, an exact solution will not only have an exponential number of solutions, but will require a profiling run for each solution to determine the increased miss rate for the remaining unlocked lines, which is completely infeasible. Consequently, we explore an approximate solution for this problem, as presented below.

5.2 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. Mathematically, this model is described as

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

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.). $LOCKLIST$ represents the list of virtual cache lines locked so far in the set where the virtual cache line x_i is mapped. LL denotes the number of elements in this list ($|LOCKLIST|$); in other words, the number of lines locked so far in this set. $LOCKLIST$ is a running list of the lines locked. It is initialized as an empty list. Every time a line is selected to be locked, the $LOCKLIST$ is updated with the line. $HIT_{LL}(x_i)$ and $MISS_{LL}(x_i)$ denote the total number of hits and miss obtained for x_i during the lifetime of the program assuming that LL number of lines were locked in the current set while T_{HIT} and T_{MISS} denote the hit latency and miss latency of the cache, respectively, expressed in processor cycles. 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. To do so, let $F(x_i)$ denote the total number of accesses to a virtual cache line x_i during the lifetime of the program. The following relation between number of accesses, number of hits and number of misses always holds true, irrespective of the number of lines currently locked (LL) in the set:

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

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:

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

If line x_i is locked in cache, only one miss (a com-

pulsory 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:

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

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

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

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_i} denotes the set of virtual cache lines mapped to the current cache set s_i, excluding the LL elements in the list $LOCKLIST$. The elements in $LOCKLIST$ are already locked in cache, hence they won't observe any opportunity cost.

According to above terminology, each line $x_j \in X_{s_i}$ observes $HIT_{LL}(x_j)$ hits. Each element belonging to set X_{s_i} 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_i} would observe a lesser number of hits, denoted by $HIT_{LL+1}(x_j)$. This constitutes the cost of locking a particular line x_i. Mathematically, for each $x_j \in X_{s_i}$, the original access time is represented by equation (5). The new access time after locking line x_i can be represented as:

$$Time(x_j|(LOCKLIST \cup \{x_i\})) = HIT_{LL+1}(x_j) * T_{HIT} \quad + \\ (F(x_j) - HIT_{LL+1}(x_j)) * T_{MISS} \tag{8}$$

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

$$\begin{aligned} Cost_{Lock(x_i)}(x_j) &= Time(x_j|(LOCKLIST \cup \{x_i\})) \\ &- (Time(x_j|LOCKLIST) \\ &= (HIT_{LL}(x_j) - HIT_{LL+1}(x_j)) \\ &* (T_{MISS} - T_{HIT}) \end{aligned} \tag{9}$$

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

$$CostLock(x_i) = \sum_{(x_j|x_j \in X_{s_i} \& x_j \neq x_i)} Cost_{Lock(x_i)}(x_j) \tag{10}$$

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

$$NetBenefit(x_i) = BenLock(x_i) - CostLock(x_i) \tag{11}$$

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 order to meet the 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 $x_i|x_i \in X_{s_i}$. If the net-benefit is negative for all the elements, the locking is discontinued for this set, 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 $LOCKLIST$ and is removed from the locking candidates set X_{s_i}. 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 ($|LOCKLIST|$)

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
$HIT_{LL}(x_i)$:Total number of hits obtained for x_i when LL lines are locked in set s_i
$MISS_{LL}(x_i)$:Total number of miss obtained for x_i when LL lines are locked in set s_i
$LockList(s_i)$: Set of virtual cache lines which should be locked in set s_i
$NumLockLines(s_i)$: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. $BenLock(x_i) = (F(x_i) - HIT_{LL}(x_i) - 1) * (T_{MISS} - T_{HIT})$
5. $CostLock(x_i) = \sum_{x_j | x_j \in X_{s_i} \& x_j \neq x_i} Cost_{Lock(x_i)}(x_j)$
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
9. and is positive.{
10. Add x_k to $LockList(s_i)$
11. $NumLockLines(s_i) = NumLockLines(s_i) + 1$
12. $X_{s_i} = X_{s_i} - x_k$
13. }
14. else {
15. break; //Locking done for this set
16. }
17. }
18. **return**;}

Figure 3: Cache Locking Algorithm.

as well as memory addresses which should be locked in this set ($LOCKLIST$). In other words, we obtain the solution for both the unknowns of cache locking problem. Figure 3 describes the psuedocode for the cache locking algorithm.

In the above cost-benefit model, HIT_{LL+1} cannot be determined precisely till we know which virtual cache line gets locked during the current step of iteration and would be different for each virtual cache line. Determing the exact value is completely infeasible given that the number of profile runs needed would equal the number of virtual cache lines, which is a very large number. Thus, an approximate value of HIT_{LL+1} is obtained by locking a dummy (unused) virtual cache line in the set apart from LL lines already locked. 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.

5.3 Binary Rewriting

In this section, we discuss the implementation of binary rewriting scheme for instruction cache locking.

As mentioned in Section 2, special locking instructions are provided in target platforms which on execution lock the elements at specified addresses in the cache lines. Once the above method determines the addresses to be locked in the cache, the binary rewriter needs to modify the original binary by including these locking instructions and produce a rewritten binary.

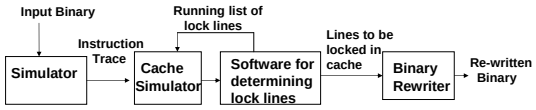


Figure 4: The Experimental WorkFlow

Since our method is based on analyzing the instruction memory addresses of original binary, the code layout of the re-written binary should be exactly the same as original binary. Modifying the program layout might render the above analysis to be incorrect. Fortunately, there are existing methods for general modifications of binaries without modifying the program layout. We adopt the *trampolines* approach suggested in [5] for modifying the binaries with these extra locking instructions. The extra locking instructions are inserted at the end of original program layout as a new *trampoline* and a call to this trampoline is inserted at the entry point of the program. The instructions replaced by the inserted call instruction are inserted in the above trampoline, resulting in a very minimal modification of the original program layout. This approach enables the application of our cache locking method directly to binaries.

6 Experimental Setup

Experiments were conducted according to the work flow presented in Figure 4. In our experiments, 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 (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. 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, or $|LOCKLIST| == K$. The final list of virtual cache lines to be locked in the cache is passed to the binary rewriter which inserts extra instructions for locking virtual cache lines according to mechanism discussed in section 5.3

App	Source	Lines Of Code	#Instr	#DynInstr
BitCnts	MiBench	543	3340	139466730
QuickSort	MiBench	79	2298	341088538
Susan	MiBench	1456	4040	63520600
Jpeg	MiBench	19804	9718	17827512
Lame	MiBench	15959	19810	220428895
Dijikstra	MiBench	268	18612	272657564
StringSearch	MiBench	3072	1839	8206548
Blowfish	MiBench	3260	2909	801766027
Rinjdal	MiBench	1017	4960	657945994
Sha	MiBench	207	2501	375176492
BasicMath	MiBench	7367	6375	298546634
FFT	MiBench	278	5868	129260146
Lout	MiBench	30689	59828	419355251
ADPCM	MediaBench	411	3734	33211608
G711	MediaBench	1173	3641	28707829

Table 1: Application Characteristics

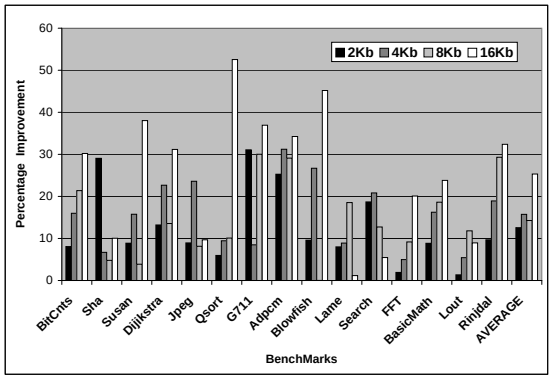


Figure 5: Percentage improvement in instruction-cache miss rate over cache with no locking for varying sizes of a 2-way set-associative cache

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 combined off-chip memory. The ARMulator software, which is part of ARM Development Suite is used to simulate the processor core. The above architectural parameters can be easily configured in ARMulator. Dinero IV [1], 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 total execution time. A sample memory map file available in ARMulator with average off-chip memory access time of 100ns 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.

7 Results

A subset of MiBench and MediaBench benchmarks were randomly 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 1 lists the benchmarks which are used for carrying out the experiments. All the benchmarks are statically compiled with the GNU-ARM toolchain.

Various kinds of experiments are performed with different cache configurations for analyzing the improvement in the instruction-cache miss rate and runtime of the applications. 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 cache locking compared to the cache configuration without locking is displayed in Figure 5 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 6 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 the commercial cached embedded processors support set-associative cache exclusively[1], which establishes our proposed approach as

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

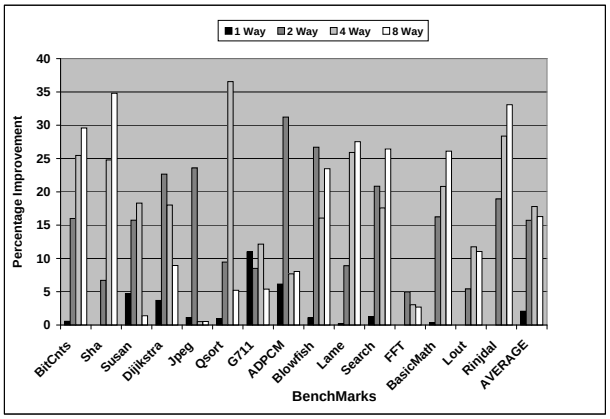


Figure 6: Percentage improvement in instruction-cache miss rate over cache with no locking for different associativities of the cache. The cache size is kept fixed at 4 Kb.

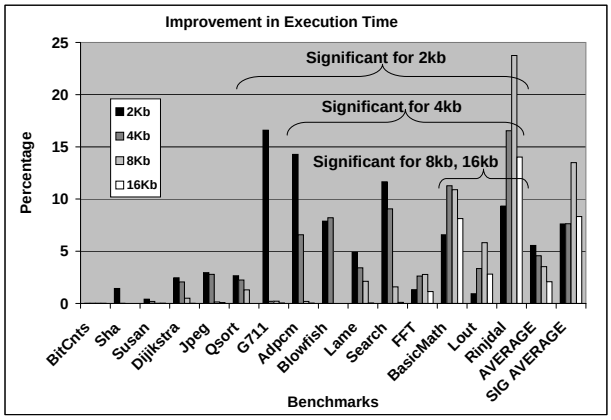


Figure 7: Improvement in execution time of the applications over cache with no locking for varying size of a 2-way set associative cache

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[2].

Next, the impact of instruction cache locking on run-time performance of various applications is analyzed. Figure 7 shows the savings in runtime obtained by using the instruction cache-locking. Comparing Figure 5 and Figure 7 brings out two 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 run-time performance. The improvement in I-cache miss rates translates to run-time performance improvement only for those applications where the overall I-cache miss rate is high. This is not surprising since a technique like ours to reduce the I-cache miss rate will not help if it is not a problem to begin with. We analyzed the initial miss rate of these applications and found a direct correlation between initial miss rate of applications and the execution time improvement obtained by our cache locking mechanism. This relation is depicted in figure 8. The benchmarks in all the figures 5 to 9 are sorted from left to right in increasing order of I-cache miss rate to illustrate the correlation between the miss rate and the overall execution time improvement.

Revisiting figure 7, 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 13.5% for a cache size of 8Kb. The averages are shown in figure 7 and figure 9 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 13.5% run-time 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 – for example, the Rinjdal benchmark has a run-time 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 ap-

cache.

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

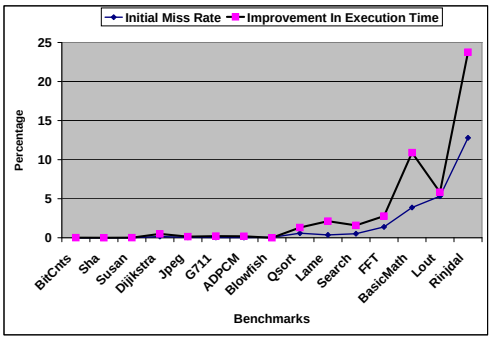


Figure 8: Graph depicting the correlation between the initial miss rate of the applications and execution time improvement obtained by our method. The miss rate and execution time results correspond to 8KB 2 way set associative cache

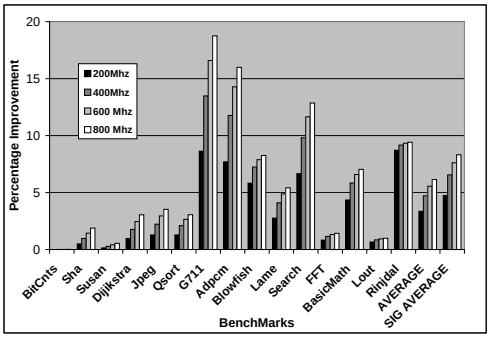


Figure 9: Variation of execution time improvement for processors with different clock speeds for a 2KB 2 way set associative cache

plications [3]. As classes of applications go, benefiting 20-60% of benchmarks significantly is quite good.

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

Finally, 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 9. Thus our method can be applied effectively for different generations of processors.

[3] e.g. faster garbage collectors only benefit benchmarks with heap data, and among those, only those with significant garbage – however garbage collection is still worthwhile.

8 Conclusion

In this paper, we have presented a instruction cache locking mechanism for improving the run-time performance of general 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 25% improvement in instruction cache miss rate and a 13.5% improvement in run-time performance of instruction memory 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.

References

[1] Dinero IV Cache Simulator, http://www.cs.wisc.edu/ markhill/DineroIV.

[2] A. Arnaud and I. Puaut. 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, May 2006.

[3] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, Colorado, May 6-8 2002. ACM.

[4] K. Beyls and E. H. D'Hollander. Generating cache hints for improved program efficiency. *J. Syst. Archit.*, 51(4):223–250, 2005.

[5] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[6] A. M. Campoy, A. P. Jimenez, A. P. Ivars, and J. V. B. Mataix. Using genetic algorithms in content selection for locking-caches, 2001.

[7] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 416–419, New York, NY, USA, 2000. ACM.

[8] H. Falk, S. Plazar, and H. Theiling. 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*, pages 143–148, New York, NY, USA, 2007. ACM.

[9] A. Gordon-Ross, S. Cotterell, and F. Vahid. Exploiting fixed programs in embedded systems: A loop cache example. *IEEE Comput. Archit. Lett.*, 1(1):2, 2002.

[10] P. R. Panda, N. D. Dutt, and A. Nicolau. 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.

[11] I. Puaut and D. Decotigny. 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, December 2002.

[12] J. B. Sartor, S. Venkiteswaran, K. S. McKinley, and Z. Wang. Cooperative caching with keep-me and evict-me. In *INTERACT '05: Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures*, pages 46–57, Washington, DC, USA, 2005. IEEE Computer Society.

[13] J. Sjodin, B. Froderberg, and T. Lindgren. Allocation of Global Data Objects in On-Chip RAM. *Compiler and Architecture Support for Embedded Computing Systems*, December 1998.

[14] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, 2006.

[15] X. Vera, B. Lisper, and J. Xue. 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*, pages 272–282, New York, NY, USA, 2003. ACM.

[16] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the conference on Design, automation and test in Europe*, page 21264. IEEE Computer Society, 2004.

[17] H. Yang, R. Govindarajan, G. R. Gao, and Z. Hu. Improving power efficiency with compiler-assisted cache replacement. *J. Embedded Comput.*, 1(4):487–499, 2005.