Sustainable Computing: Informatics and Systems 2 (2012) 71–80

Contents lists available at SciVerse ScienceDirect

Sustainable Computing: Informatics and Systems

journal homepage: www.elsevier.com/locate/suscom

Compression-aware dynamic cache reconfiguration for embedded systems☆

Hadi Hajimiri*, Kamran Rahmani, Prabhat Mishra

Department of Computer & Information Science & Engineering, University of Florida, Gainesville, FL, USA

A R T I C L E   I N F O

Article history:
Received 30 September 2011
Received in revised form 17 January 2012
Accepted 30 January 2012

Keywords:
Cache reconfiguration
Code compression
Embedded systems

A B S T R A C T

Optimization techniques are widely used in embedded systems design to improve overall area, performance and energy requirements. Dynamic cache reconfiguration is very effective to reduce energy consumption of cache subsystems which accounts for about half of the total energy consumption in embedded systems. Various studies have shown that code compression can significantly reduce memory requirements, and may improve performance in many scenarios. In this paper, we study the challenges and associated opportunities in integrating dynamic cache reconfiguration with code compression to retain the advantages of both approaches. We developed efficient heuristics to explore large space of two-level cache hierarchy in order to study the effect of a two-level cache on energy consumption. Experimental results demonstrate that synergistic combination of cache reconfiguration and code compression can significantly reduce both energy consumption (61% on average) and memory requirements while drastically improve the overall performance (up to 75%) compared to dynamic cache reconfiguration alone.

1. Introduction

Energy conservation has been a primary optimization objective in designing embedded systems as these systems are generally limited by battery lifetime. Several studies have shown that memory hierarchy accounts for as much as 50% of the total energy consumption in many embedded systems [1]. Dynamic cache reconfiguration (DCR) and code compression are two of the extensively studied approaches in order to achieve energy savings as well as area and performance gains.

Different applications require highly diverse cache configurations for optimal energy consumption in the memory hierarchy. Unlike desktop-based systems, embedded systems are designed to run a specific set of well-defined applications. Thus it is possible to have a cache architecture that is tuned for those applications to have both increased performance as well as lower energy consumption. Since too many cache configurations are possible, the challenge is to determine the best cache configuration (in terms of total size, associativity, and line size) for a particular application. Studies have shown that cache tuning can achieve 53% memory-access-related energy savings and 30% performance improvement [2].

The use of high-level programming languages coupled with RISC instruction sets leads to a larger memory footprint and increased area/cost and power requirements, all of which are important design constraints in most embedded applications. Code compression is clearly beneficial for memory size reduction because it reduces the static memory size of executable code. Several code compression techniques have been proposed for reducing instruction memory size in low cost embedded applications [3]. The basic idea is to store instructions in compressed form and decompress them on-the-fly at execution time. More importantly, code compression could also be beneficial for energy by reducing memory size and the communication between memory and the processor core [4].

Design of efficient compression techniques needs to consider two important aspects. First, the compressed code has to support the possibility of starting the decompression during execution at several points inside the program (i.e., branch targets). Second, since decompression is performed on-line, during program execution, decompression algorithms should be fast and power efficient to achieve savings in memory size and power, without compromising performance. We explore various

☆ This is an extended version of the paper that has appeared in the proceedings of International Green Computing Conference (IGCC) 2011 [22]. The IGCC paper presented some initial results for employing dynamic cache reconfiguration and code compression for embedded systems with one level (L1) cache. This article considers the effects of code compression on DCR in the presence of both level one (L1 data and L1 instruction) and unified level two (L2) caches. Specifically the following major contributions are new: (i) we have used a highly configurable two-level cache hierarchy and added Section 4 to propose multi-level cache tuning heuristics for DCR; (ii) we have added Section 5.3 to present new experimental results for exploration of two-level cache hierarchy.

* Corresponding author.
E-mail addresses: hadi@cise.ufl.edu, hadi.hajimiri@gmail.com (H. Hajimiri), kamran@cise.ufl.edu (K. Rahmani), prabhat@cise.ufl.edu (P. Mishra).

2210-5379/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.suscom.2012.01.003

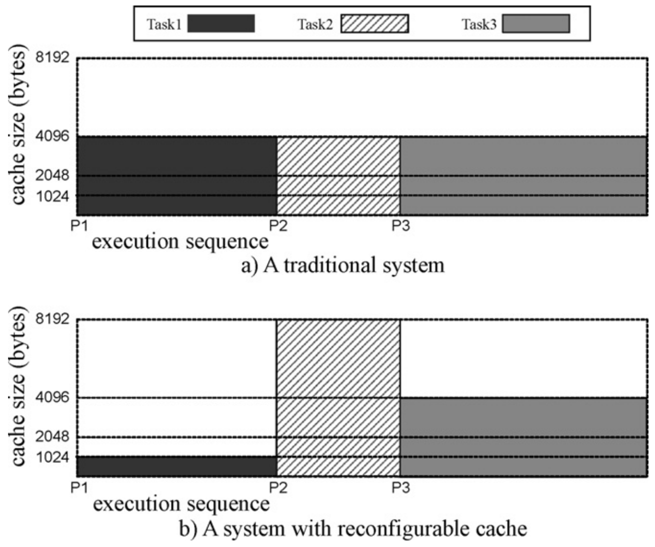


Fig. 1. DCR for a system with three tasks.

compression techniques (including dictionary-based compression, bitmask-based compression and Huffman coding) that represent a trade-off between compression performance and decompression overhead.

It is expected that by compressing instructions the cache behavior of programs is no longer the same. Thus in order to have the optimal cache configuration, more analysis should be done including hit/miss behavior of the compressed programs. In other words, cache reconfiguration needs to be aware of code compression to obtain best possible area, power and performance results. In this paper, we present an elaborate analysis of combining two optimization techniques: dynamic cache reconfiguration and code compression. In addition, we propose efficient heuristics to explore large design space of two-level cache hierarchy in order to find energy efficient cache configurations. Our experimental results demonstrate that the combination is synergistic and achieves more energy savings as well as overall performance improvement compared to DCR and code compression alone.

The rest of the paper is organized as follows. Section 2 provides an overview of related research activities. In Section 3, we describe our compression-aware cache reconfiguration methodology. Section 4 presents efficient heuristics to explore two-level cache hierarchy. Section 5 presents our experimental results. Finally, Section 6 concludes the paper.

2. Background and related work

2.1. Dynamic cache reconfiguration (DCR)

In power constrained embedded systems, nearly half of the overall power consumption is attributed to the cache subsystem [1]. Applications require vastly different cache requirements in terms of cache size, line size, and associativity. Research shows that specializing the cache to application's needs can significantly reduce energy consumption [2]. Fig. 1 illustrates how energy consumption can be reduced by using inter-task (application-based) cache reconfiguration in a simple system supporting three tasks. In application-based cache tuning, DCR happens when a task starts its execution or it resumes from an interrupt (either by preemption or when execution of another task completes) and the same cache for the application gets chosen no matter if it is starting from the beginning or resuming anywhere in between. Fig. 1(a) depicts a traditional system and Fig. 1(b) depicts a system with a reconfigurable cache. For the ease of illustration let's assume cache size is the only reconfigurable parameter of cache (associativity and line size are ignored). In this example, Task1 starts its execution at time P1. Task2 and Task3 start at P2 and P3 respectively. In a traditional approach, the system always executes using a 4096-byte cache. We call this cache as **base cache** throughout the paper. *Base cache is the best possible cache configuration optimized for all the tasks*. With the option of reconfigurable cache, Task1, Task2, and Task3 execute using 1024-byte cache starting at P1, 8192-byte cache starting at P2, and 4096-byte cache starting at P3 respectively. Through proper selection of cache size for each task the system can achieve significant amount of energy savings as well as performance gains compared to using only the *base cache*.

The inter-task DCR problem is defined as follows. Consider a set of n applications (tasks) $\mathbf{A} = \{a_1, a_2, a_3, \ldots, a_n\}$ intended to run on a configurable cache architecture capable of supporting m possible cache configurations $\mathbf{C} = \{c_1, c_2, c_3, \ldots, c_m\}$. We define $e(c_j, a_i)$ as the total energy consumed by running application a_i on the architecture with cache configuration c_j. We also define $c_o \in \mathbf{C}$ as the optimal cache configuration for application a_i, such that $e(c_o, a_i) \leq e(c_j, a_i), \forall c_j \in \mathbf{C}$. Through exhaustive exploration of all possible configurations of $\mathbf{C} = \{c_1, c_2, c_3, \ldots, c_m\}$, best energy optimal cache configuration for each application can be found.

Dynamic cache reconfiguration has been extensively studied in several works [5–8]. The reconfigurable cache architecture proposed by Zhang et al. [6] determines the best cache parameters by using Pareto-optimal points trading off energy consumption and performance. Their method imposes no overhead to the critical path, thus cache access time does not increase. Chen et al. [9] introduced a novel reconfiguration management algorithm to efficiently search the large design space of possible cache configurations for the optimal one. None of these approaches consider the effects of compressed code on cache reconfiguration.

DCR can be viewed as a technique that tries to squeeze cache size with other cache parameters to reduce energy consumption without (or with minor) performance degradation. Smaller caches contribute less static power but may increase cache misses which can lead to increased dynamic power and performance degradation (longer execution time thus higher energy consumption). Therefore, the smallest possible cache may not be a feasible solution in many cases. DCR techniques find the best cache that fits the application by exploring cache configurations using various schemes. In this paper, we show that code compression which significantly reduces the code size can also help the cache reconfiguration technique to choose relatively smaller cache sizes, smaller associativity, or smaller line size without performance degradation, therefore, reduces cache energy consumption significantly.

The configurable caches used in our work are based on the architecture described in [10]. The underlying cache architecture contains four separate banks that can operate as four separate ways. Special configuration registers are used to inform the cache tuner – a custom hardware or a lightweight process – to concatenate ways such that the associativity can be altered. The special registers may also be configured to shut down ways to vary the cache size. Similarly, by configuring the fetch unit to fetch cache lines in various lengths, we can adjust the line sizes. The area overhead for this architecture is 3%. In addition, searching an average of 5.4 configurations to find the best configuration has a very low energy consumption of 11.9 nJ on average. This energy is negligible compared to the energy consumption in benchmarks that is 2.34 J on average. In this paper the only part that we used is dynamic cache reconfiguration. It means our architecture is not self-tuning and has much less overhead compared to [10].

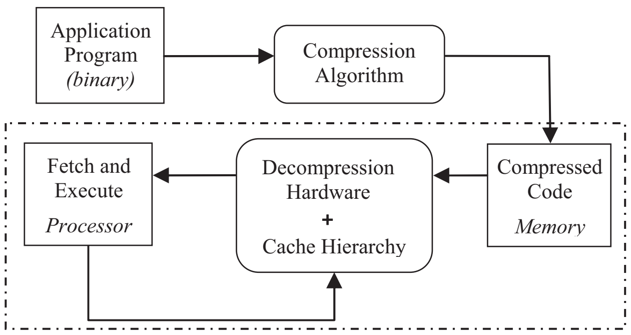

Application Program (binary)
Compression Algorithm
Fetch and Execute
Processor
Decompression Hardware + Cache Hierarchy
Compressed Code
Memory

Fig. 2. Traditional code compression methodology.

2.2. *Code compression in embedded systems*

Various code compression algorithms are suitable for embedded systems, i.e., provide good compression efficiency with minor (acceptable) or no decompression overhead. Wolfe and Chanin [11] were among the first to propose an embedded processor design that incorporates code compression. Xie et al. [12] introduced a compression technique capable of compressing flexible instruction formats in VLIW architectures. Seong and Mishra [13] modified dictionary-based compression (BMC) technique using bitmasks which improved compression efficiency without introducing any additional decompression overhead. Lin et al. [14] proposed LZW-based algorithms to compress branch blocks. Recently, Rawlins and Gordon-Ross [15] used compressed programs in their approach of combined loop caching with DCR. Their approach has several limitations. They primarily focus on loop caching which may not be applicable in many embedded systems due to intrusive addition of another level of cache. Furthermore, due to emphasis on loop caching, interactions between compression and DCR was not explored in detail. In this paper we provide comprehensive analysis of how compression and DCR synergistically interact with each other as well as energy-performance trade-offs available for system designer.

Traditional code compression and decompression flow is illustrated in Fig. 2 where the compression is done offline (prior to execution) and the compressed program is loaded into the memory. The decompression is done during the program execution (online) and as shown in Fig. 7 it can be placed before or after cache. It is possible to place the decompression unit between two levels of cache as well, if the system has multi-level cache hierarchy.

In this paper we explore three compression techniques: dictionary-based compression (DC), bitmask-based compression (BMC) [13], and Huffman coding. DC and Huffman coding represent two extremes. DC is a simple compression technique and therefore produces moderate compression but decompression is very fast. On the other hand, Huffman coding is considered to be one of the most efficient compression techniques but has higher decompression overhead/latency. DC and Huffman are widely used but BMC is a recent enhancement of DC that enables more matching patterns. Fig. 3 shows the generic encoding formats of bitmask-based compression technique for various numbers of bitmasks. Compressed data stores information regarding the bitmask type, bitmask location, and the mask pattern itself. The bitmask can be applied in different places in a vector and the number of bits required for indicating the position varies depending on the bitmask type. Bitmasks may be sliding or fixed. A fixed bitmask can be applied to fixed locations, such as byte boundaries. However, sliding bitmasks can be applied anywhere in the code vector.

The main advantage of bitmask-based compression over traditional dictionary-based compression is the increased matching patterns. In dictionary-based compression, each vector is compressed only if it completely matches with a dictionary entry. Fig. 4 illustrates an example of bitmask-based compression in which it can compress up to six data entries using bitmask-based compression, whereas using only dictionary-based compression would compress only four entries. The example in Fig. 4 uses only one bitmask. In this case, vectors that match exactly a dictionary entry are compressed with 3 bits. The first bit represents whether it is compressed (using 0) or not (using 1). The second bit indicates whether it is compressed using bitmask (using 0) or not (using 1). The last bit indicates the dictionary index. Data that are compressed using bitmask requires 8 bits. The first two bits, as before, represent if the data is compressed, and whether the data is compressed using bitmasks. The next three bits indicate the bitmask position and followed by two bits that indicate the bitmask pattern.

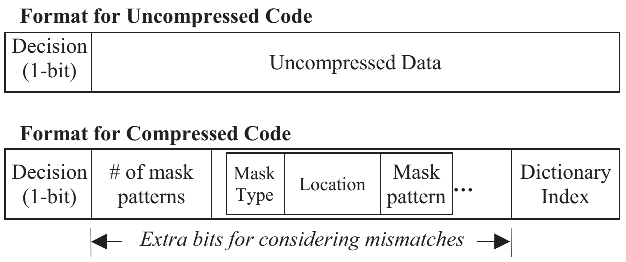


Fig. 3. Encoding format for incorporating mismatches.

In this example, the compression ratio is 80%. *Compression ratio* (CR), widely accepted as a primary metric for measuring the efficiency of code compression, is defined as:

$$CR = \frac{\text{Compressed program size}}{\text{Original program size}}$$

Bitmask selection and dictionary selection are two major challenges in bitmask-based code compression. Seong and Mishra [13] have shown that the profitable bitmasks to be selected for code compression are 1s, 2s, 2f, 4s, and 4f (**s** and **f** stand for sliding and fixed bitmasks respectively). Since the decompression engine must be able to start execution from any of jump targets, branch targets should be aligned in the compressed code. In addition, the mapping of old addresses (in the original uncompressed code) to new addresses (in the compressed code) is kept in a jump table.

3. Compression-aware DCR

It is a major challenge to optimize both performance and energy consumption simultaneously. In case of DCR, tradeoffs between performance and energy consumption should be considered in order to choose the most profitable cache configuration for each application. Fig. 5 shows an example of performance-energy

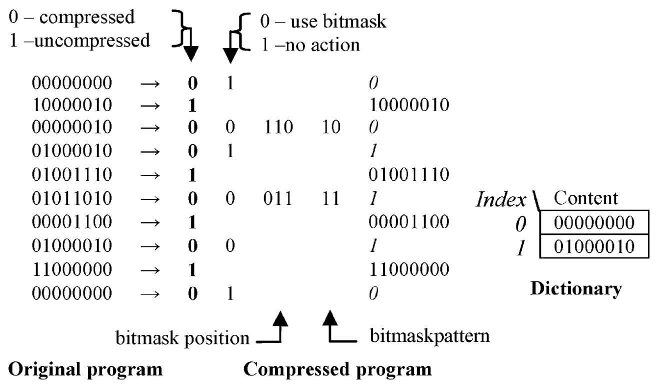


Fig. 4. An example of bitmask-based code compression.

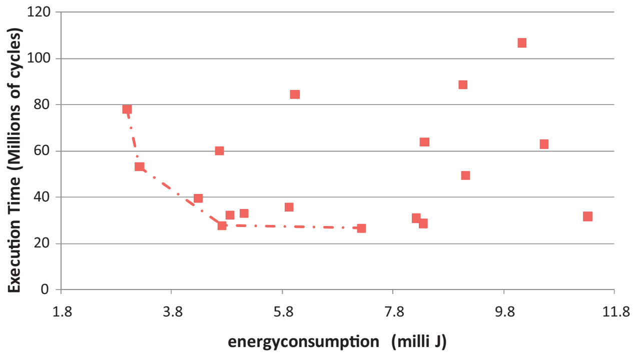


Fig. 5. An example of performance-energy consumption tradeoff using *Anagram* benchmark (Pareto optimal alternatives are connected using dashed lines).

consumption tradeoff using *Anagram* benchmark. Each dot represents a cache configuration showing its corresponding energy consumption and total execution time of the task. By plotting all cache configurations in performance-energy consumption graph (based on time and energy consumption from simulation results) we can determine Pareto optimal points representing feasible alternatives. For instance, increasing cache line or associativity can improve performance and may increase energy consumption as well. High performance alternatives will sacrifice some amounts of energy while selecting energy saving options would have lower performance. The remainder of this section describes how to combine the advantages of both compression and dynamic reconfiguration.

3.1. Motivation

A reconfigurable cache can be viewed as an elastic cache with flexible parameters such as cache size, line size, and associativity. The dynamic reconfiguration technique exploits the elasticity of such caches by selecting a profitable cache configuration which is capable of maintaining the critical portion of the application to reduce energy consumption. Choosing smaller caches that fail to store the critical portion of the program may lead to increased cache misses thus longer execution time and eventually escalation in energy consumption. However, it is possible that the cache reconfiguration method may find a cache configuration that increases the execution time of the application in spite of reduced energy consumption. This may not be an issue for systems without real-time constraints but timing constraints in real-time applications limit use of such cache reconfiguration techniques. Integrating code compression with cache reconfiguration resolves this problem by effectively shrinking the program size in order to fit the critical portion of the application into a smaller cache.

Fig. 6 illustrates different caches for a real-time embedded system with a set of applications. Associativity is ignored for the ease of illustration. The horizontal and vertical axis show different possibilities of cache size and line size, respectively. The *base cache* is a globally optimized cache is used for all applications and has the minimal aggregate energy consumption while ensuring that no deadlines are missed. As an illustrative example, Fig. 6 shows one application in this set in three scenarios: no reconfiguration or compression, reconfiguration without compression, and reconfiguration + compression. *Cache1* is used for this application when no compression or cache reconfiguration is available. *Cache2* is the cache selected by dynamic reconfiguration technique (with no compression) to reduce the energy consumption of this application. But to ensure real-time (deadline) constraints, low energy cache alternatives may get rejected because of longer execution times (critical portion of applications may not fit, for example).

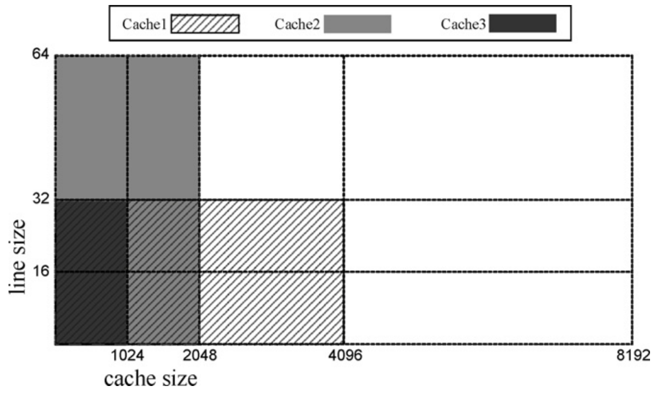


Fig. 6. Different caches used in different scenarios. Cache1: conventional system without reconfiguration, Cache2: only dynamic reconfiguration (no compression), Cache3: both dynamic reconfiguration and compression.

Incorporating compression into DCR would lead to selection of *Cache3*. Applying compression will help dynamic reconfiguration to perfectly fit the critical portion of the application into smaller cache thus gaining even more energy savings without increasing the execution time.

3.2. Compression-aware DCR

Here, we consider systems with one level cache. In Section 4 we extend our approach for systems with two-level cache. Algorithm 1 outlines the major steps in our cache configuration selection in the presence of compressed applications. The algorithm collects simulation results for all possible cache configurations (cache sizes of 1 KB, 2 KB, 4 KB, and 8 KB; associativity of 1, 2, 4-way; cache line sizes of 16, 32, 64). It finds the best energy optimal cache configuration for each application through exhaustive exploration of all possible cache configurations of $C = \{c_1, c_2, c_3, \ldots, c_m\}$. Number of simulation cycles for each run is collected based on the simulation results. The energy model of [6] is used to calculate the energy consumption using the cache hit and miss statistics. The algorithm finally constructs the Pareto optimal alternatives and returns it in a list. The most energy efficient cache configuration among all Pareto optimal alternatives which satisfies timing requirements of the application is chosen next. Suppose there are two cache configurations, C1 with execution time of 2 million cycles and energy consumption of 5 mJ and C2 with execution time of 1.8 million cycles and energy consumption of 6 mJ, available in the Pareto optimal list of alternatives. If the task has to be done in 1.9 million cycles, the faster alternative (C2) gets chosen. If the timing requirement of the task is not constrained by 2 million cycles, the more energy efficient cache alternative (C1) gets selected.

Algorithm 1.

```
Finding Pareto optimal cache configurations
Input: compressed code
Output: List of Pareto optimal cache alternatives
Begin
  li = an empty list to store cache alternatives
  for s = cache sizes of 1 KB, 2 KB, 4 KB, and 8 KB do
    for a = associativity of 1,2,4-way do
      for l = cache lines of 16,32,64 do
        do cycle accurate simulation for cache $C_{s,a,l}$;
        $t_{s,a,l}$ = simulation cycles;
        $e_{s,a,l}$ = energy consumption of the cache subsystem;
        add the triple ($C_{s,a,l}$, $t_{s,a,l}$, $e_{s,a,l}$) to li;
      end for
    end for
  end for
  return Pareto optimal points in li;
end
```

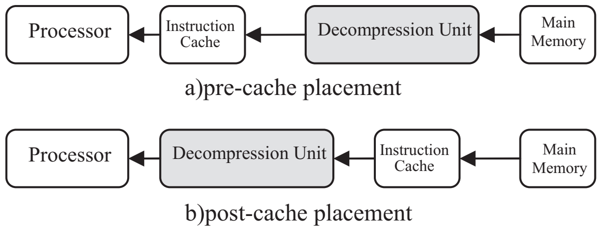


Fig. 7. Different placement of decompression unit.

The algorithm is similar to traditional DCR but uses compressed code. Therefore the simulation/profiling infrastructure needs to have decompression unit to provide the ability of decoding compressed instructions. For example, in our case, we implemented and placed the required decompression routines/functions for respective compression algorithms in *Simplescalar* simulator [16].

In this section, we consider systems with only one level of reconfigurable cache architecture; therefore number of cache configurations is small. So we can exhaustively explore all possible configurations in a reasonable time. Since the reconfiguration of associativity is achieved by way concatenation, 1 KB L1 cache can only be direct-mapped as other three banks are shut down. For the same reason, 2 KB cache can only be configured to direct-mapped or 2-way associativity. Therefore, there are 18 (=3 + 6 + 9) configuration candidates for L1.

3.3. Placement of decompression hardware

Fig. 7 shows two different placement of the decompression unit. In pre-cache placement the memory contains compressed code and instructions are stored in cache in original form. Whereas, in the post-cache placement the decompression unit is placed between cache and processor thus both memory and cache contain compressed instructions.

Our studies show that having the pre-cache placement has very little effect on energy and performance of cache. In this case uncompressed instructions are stored in the cache and when cache miss occurs, the cache controller asks the decompression unit to provide a block of instructions. In majority of the cases the decompression hardware requires one clock cycle in pipelined mode (as shown in Fig. 7), so one clock cycle will be added to the latency of entire block fetch. In rare cases, e.g., when the first instruction of the block is not compressed, it will introduce two cycle penalty since it will take two cycles to fetch and decompress the instruction [17]. As demonstrated in Fig. 8, the energy consumption of cache in the pre-cache placement is almost the same as the case when there is no compression involved. So the best choice is to use post-cache

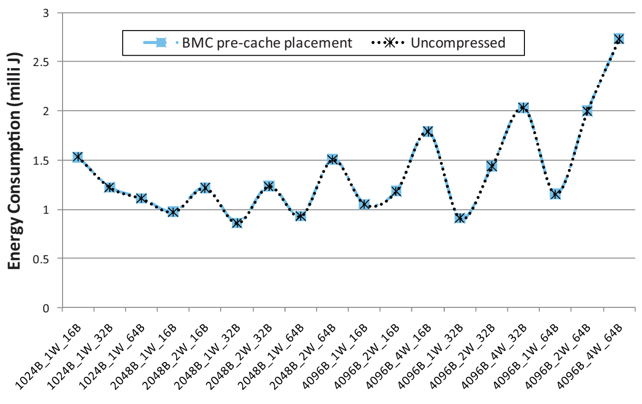


Fig. 8. The impact of pre-cache placement of decompression engine on cache energy – *djpeg* benchmark.

placement to achieve maximum performance as well as minimum energy consumption.

Incorporating compression, cache miss penalty caused by memory fetch latency is reduced because of improved bandwidth (since compressed code is smaller). In addition, off-chip access energy (the buses to main memory and memory access) is also reduced since the decompression engine reads compressed code from memory resulting in lower traffic to main memory. However, post-cache placement can introduce significant performance overhead to the system. Seong and Mishra [13] presented a bitmask-based compression technique that adds no penalty to the system performance using pipelined one-cycle decompression engine with negligible power requirement. Using this decompression engine makes it practical to place the decompression unit after cache (post-cache placement) and benefit from the compressed code stored in the cache.

In the context of embedded systems one of the main goals is maximizing energy savings while ensuring the system will meet applications requirements. Usually, choosing a cache configuration for energy savings may result in performance degradation. However, the synergistic combination of cache reconfiguration and code compression enables energy savings without loss of performance. Our proposed methodology provides an efficient and optimal strategy for cache tuning based on static profiling using compressed programs.

4. Tuning of two-level caches

In this section, we study the effect of a two-level cache hierarchy on compression and DCR. We consider a system with a unified (instruction/data) level two cache (L2). We compress only instructions. In other words, we do not consider data compression in this paper. However, selecting energy efficient cache configuration for L2 cache is dependent on both level one instruction and data caches (IL1 and DL1). Therefore we consider the energy consumption of the entire cache subsystem including IL1, DL1, and L2.

We present efficient heuristics to generate profile tables with profitable cache configurations. Tuning a two-level cache faces the difficulty of exploring an enormous configuration space. In this paper, we examine typical exploration parameters of a two-level cache in conventional embedded systems. As discussed in Section 3.2, there are 18 (=3 + 6 + 9) configuration candidates for L1 caches. Let S_{il1} and S_{dl1} denote the size of exploration space for IL1 cache and DL1 caches, respectively. So we have $S_{il1} = 18$ and $S_{dl1} = 18$. For L2 cache, we choose 8 KB, 16 KB and 32 KB as cache sizes; 32, 64 and 128 bytes as line sizes; 4-, 8- and 16-way set associativity with a 32 KB cache architecture composed of four separate banks. Similarly, there are 18 possible configurations ($S_{ul2} = 18$). For comparison, we have chosen a *base cache* hierarchy, which reflects a global optimal configuration for all the tasks, consisting of two 2 KB, 2-way set associative L1 caches with a 32 byte line size, and a 16 KB, 8-way set associative unified L2 cache with a 64 byte line size. The remainder of this section describes our proposed exploration techniques.

4.1. Exhaustive exploration

Intuitively, if the two levels of caches can be explored independently, one can easily profile one level at a time while holding the other level to a typical configuration, which will result in a much smaller exploration space. However, there is no certainty that the combination of three independently found energy-optimal configurations would be close to the global optimal one. The two cache levels affect each other's behavior in various ways. For instance, L2 cache's configuration determines the miss penalty of the L1 caches. Also, the number of L2 cache accesses directly depends on the number of L1 cache misses.

The obvious way to find the optimal configuration is to search the entire space exhaustively. Since the instruction and data caches could have different configurations, there are 324 (=$S_{il1}*S_{dl1}$) possible configurations for L1 cache. Addition of the L2 cache increases the design space size to 4752 (Not equal to $S_{il1}*S_{dl1}*S_{ul2}$ because the candidates in which L2 cache's line size is smaller than any of the L1 caches are eliminated). We use the exhaustive method for comparison with the heuristics presented in the following sections. Design of these heuristics is motivated by the exploration heuristics of Wang and Mishra [18]. However, our approach also considers the effect of compression during exploration.

4.2. Independent L1 cache tuning – ICT

While different cache levels are dependent on each other, our initial results demonstrate that instruction and data caches are relatively independent. In this study, we fix one's configuration while changing the other's to see whether varying one impacts the fixed one. We observe that the profiling statistics for the instruction cache almost remain identical with different data caches and vice versa. It is mainly due to the fact that access pattern of L1 cache is purely determined by the application's characteristics, and the instruction and data streams are relatively independent from each other. Furthermore, factors affecting the instruction cache's energy consumption as well as performance (such as hit energy, miss energy and miss penalty cycles) have very little dependency on the data cache and vice versa.

This observation offers an opportunity to reduce the exploration space. We propose ICT – Independent L1 Tuning heuristic – during which IL1 and DL1 caches always use the same configuration while exploring with all L2 cache configurations. This method results in a total of 288 configurations – a considerable cut down of the original quantity, though still not small. Throughout the static analysis, we make book keeping including the energy consumptions and miss cycles of each cache individually. The energy-optimal IL1 cache is the one with the lowest energy consumption of itself (and same for DL1 cache and L2 cache). We choose the cache configuration combination composed of the three locally energy-optimal caches as the energy-optimal cache hierarchy to be stored in the profile table.

4.3. Interlaced tuning – ILT

We adapt the strategy used in TCaT [2] and propose ILT – Interlaced Tuning heuristic – which finds energy-optimal parameters throughout the exploration. The basic idea is to tune cache parameters in the order of their importance to the overall energy consumption, which is cache size followed by line size and finally associativity. In order to increase the chances of finding optimal L2 cache size, which we believe has the highest importance, we combine the exploration of L2 cache's size and associativity together. ILT is described below:

1. First, tune by cache size. Hold the IL1's line size, associativity as well as DL1 to the smallest configuration. L2 is set to the base cache. Explore all three instruction cache sizes (1 KB, 2 KB and 4 KB) and find out the energy-optimal one(s). Perform same explorations for DL1 cache size. In L2 size exploration, we try all the associativities for each cache size. We set L1 sizes to the energy-optimal ones in the process of finding energy-optimal L2 size(s).
2. Next, tune by line size. We set cache sizes to the energy-optimal ones and L2's associativity found in the first step in exploring energy-optimal line sizes for each cache. These two tasks are repeated for both L1 caches and L2.
3. Finally, tune by associativity. We set the cache sizes and line sizes to the energy-optimal ones in exploring energy-optimal associativity. Note that we only explore associativities for L1 caches in this step. During the process of finding DL1's optimal associativities, we already have all the other parameters we needed to compute the total numbers of execution cycles that are required in the profile table.

In the worst case, ILT explores 30 configurations. The first step explores 6 for L1 caches and 9 for L2 cache. The second step explores 9 (=3*3) candidates. Final step explores 6 (=3*2) candidates. However, in most cases, there are a lot of repetitive configurations throughout the process that we only have to execute once. In practice, ILT has exploration space size of around 19 configurations.

4.4. Hierarchy level independent tuning – HIT

Although we stated that IL1 and DL1 can be selected independently, in some cases it is better to explore the two level one caches together. Suppose for a particular benchmark, there is a large variation in the require L2 size for data/instruction when changing IL1 or DL1. In this case, using a large portion of L2 for instruction for a specific IL1 configuration can affect DL1 indirectly (may increase data access miss ratio in L2). Since ICT finds energy-optimal caches for IL1 and DL1 independently without considering the effect of each on L2 cache behavior, it may produce suboptimal results. We propose HIT – Hierarchy Level Independent Tuning – in which we first find the optimal cache configurations for level one caches fixing L2 to the base cache. We explore all possible 324 (=18*18) combinations for IL1 and DL1 caches and select the energy optimal ones. Next we fix L1 caches to the found energy optimal caches in the first step and try all 18 candidates for L2 cache. In summary, ILT explores only 30 configurations, whereas ICT and HIT explore 288 and 342 (=324 + 18) configurations, respectively.

5. Experiments

In order to quantify compression-aware cache configuration tradeoffs, we have applied our methodology to select embedded system benchmarks. Following the same flow in Sections 3 and 4, we first investigate integration of code compression with DCR for systems with one level of cache. In subsection 0, we extend our experiments to evaluate our method with the presence of a two-level cache.

5.1. Experimental setup

We examined *cjpeg*, *djpeg*, *epic*, and *adpcm* (*rawcaudio*), *g.721* (*encode*, *decode*) benchmarks from the MediaBench [19] and *dijkstra*, *patricia* from MiBench [20] compiled for the Alpha target architecture. These benchmarks are all specially designed for embedded systems and suitable for the cache configuration parameters described in Section 3.2. All applications were executed with the default input sets provided with the benchmarks suites.

Three different code compression techniques including bitmask-based, dictionary-based and Huffman code compression were used. To achieve the best attainable compression ratios, in bitmask-based compression, for each application we examined dictionaries of 1 KB, 2 KB, 4 KB, and 8 KB. Similar to Seong and Mishra [13] we tried three mask sets including one 2-bit sliding, 1-bit sliding and 2-bit fixed, and 1-bit sliding and 2-bit fixed masks. Similarly for dictionary-based and Huffman compression we used 0.5 KB, 1 KB, 2 KB, 4 KB, and 8 KB dictionary sizes with 8 bits, 16 bits and 32 bits word sizes. We found out that dictionary size of

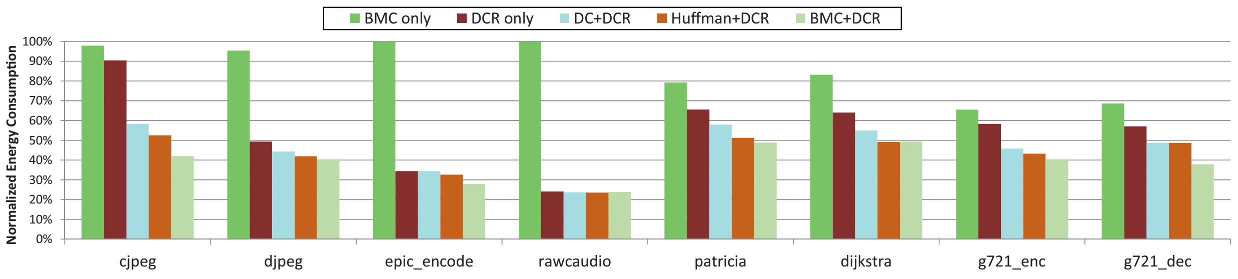


Fig. 9. Energy consumption of the selected "minimal-energy cache" normalized to the base cache.

2 KB and word size of 16 bits are the best choices for this set of benchmarks. The reason is that using 8 bits words increases the number of compression decision bits and using 32 bits word size decreases the words frequencies significantly. Hence, as simulation results showed, 16 bits word size is the best choice.

Code compression is performed offline. In order to extract the code (instruction) part from executable binaries, we used ECOFF (Extended Common Object File Format) header files provided in SimpleScalar toolset [16]. We placed the compressed code back into binary files so that they can be loaded into the simulator.

We utilized the configurable cache architecture developed by Zhang et al. [6] with a four-bank cache of base size 4 KB, which offers sizes of 1 KB, 2 KB, and 4 KB, line sizes ranging from 16 bytes to 64 bytes, and associativity of 1-way, 2-way, and 4-way. For comparison purposes, we used the *base cache* configuration for L1 set to be a 4 KB, 4-way set associative cache with a 32-byte line size, a reasonably common configuration that meets the average needs of the studied benchmarks.

To obtain cache hit and miss statistics, we modified the *SimpleScalar* toolset [16] to decode and simulate compressed applications. We implemented and placed the required decompression routines/functions for respective compression algorithms in *Simplescalar* simulator. We considered the latency of decompression unit carefully. Decompression unit can decompress the next instruction in one cycle (in pipelined mode) if it finds the entire needed bits in its buffer. Otherwise, it takes one cycle (or more cycles, if cache miss occurs) to fetch the needed bits into its buffer and on more cycle to decompress the next instruction. Correctness of the compression and decompression algorithms was verified by comparing the outputs of compressed applications with uncompressed versions. The performance overhead of decompression includes decompression unit buffer flush overhead due to jumps, and variable latency of memory reads in each block fetch (because of variable length compressed code). These overhead are negligible according to the experimental results.

We applied the same energy model used in [6], which calculates both dynamic and static energy consumption, memory latency, CPU stall energy, and main memory fetch energy. The energy model was modified to include decompression energy. We updated the dynamic energy consumption for each cache configuration using CACTI 4.2 [21].

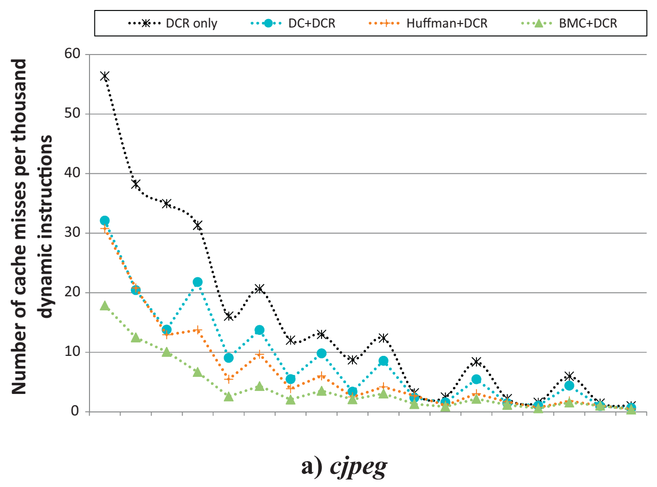
5.2. One-level cache tuning

Energy consumption for several benchmarks from the MediaBench and MiBench in different approaches are analyzed: a fixed *base cache* configuration, bitmask-based compression without utilizing DCR (BMC only), DCR without compression (DCR only), dictionary-based compression with DCR (DC + DCR), Huffman coding with DCR (Huffman + DCR), and bitmask-based compression with DCR (BMC + DCR). The most energy efficient cache configuration found by exploration in each technique is considered for comparison. Fig. 9 presents energy savings for the instruction cache subsystem. Energy consumption is normalized to the fixed *base cache* configuration such that value of 100% represents our baseline. Energy savings in the instruction cache subsystem ranges from 10% to 76% with an average of 45% for utilizing only DCR. As we expected, due to higher decompression overhead, Huffman (when combined with DCR) achieves lower energy savings compared to BMC virtually for all benchmarks. Energy savings in DC + DCR approach are even lower than Huffman + DCR as a result of moderate compression ratio by DC. Incorporating BMC in DCR increases energy savings up to 48% – on top of 10–76% energy savings obtained by DCR only – without any performance degradation. Our methodology achieves on average 61% energy savings of the cache subsystem.

Energy consumption of some benchmarks is reduced drastically when using BMC. For example, energy consumption of *cjpeg* benchmark is decreased by nearly 50% when applying BMC on DCR compared to using DCR alone. Fig. 10(a) shows the number of cache misses per thousand dynamic instructions for *cjpeg* benchmark. It shows that for smaller cache sizes, cache misses are drastically reduced when incorporating compression. In other words, by using compression, smaller cache sizes are capable of containing the critical portion of cjpeg benchmark and keep the number of misses low (maintaining performance) while reducing static energy consumption. Fig. 10(b) presents the same statistics for *rawcaudio* benchmark. It should be noticed that although integrating compression with DCR reduces the number of cache misses when using small cache sizes (similar to *cjpeg* behavior) it does not drastically decrease energy consumption. The extremely low range of cache misses, usually less than 0.05 (0.45 in the extreme case) misses per thousand dynamic instructions, leads to nominal contribution of cache misses to the overall energy consumption of the cache. For this reason dynamic energy consumption is nearly the same for all configurations and DCR chooses the smallest possible cache configuration to minimize the static energy. In this case, incorporating compression in DCR with the selection of small cache size can only reduce the cache misses and therefore dynamic energy and thus has a small impact on overall cache energy consumption for *rawcaudio* benchmark.

Fig. 11 illustrates an example of performance-energy consumption tradeoffs for both uncompressed and compressed (using BMC) cases for *rawcaudio* (*adpcm-enc*) benchmark. It can be observed that for every possible configuration for the uncompressed program there is an alternative which has a better performance and lower energy requirement if the program is compressed. This observation shows that compression-aware DCR leads to better design choices.

Another observation we have made is that without DCR, applying compression on an application (which executes using *base cache* configuration that already fits the critical portion of the application) will not gain noticeable energy savings. However, compression-aware DCR effectively uses the advantage of reduced program size achieved by compression to choose smaller cache size, associativity, or line size and yet fit critical portion of programs. Therefore, compression aware-DCR can achieve more energy savings compared to DCR alone. Fig. 12 illustrates comparison of energy profile



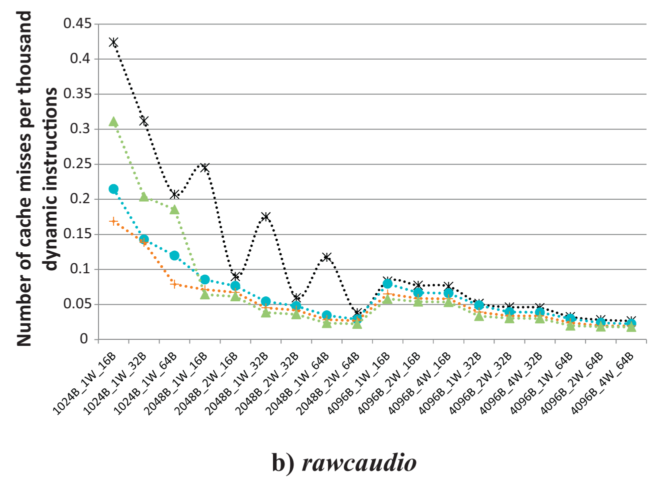


Fig. 10. Number of cache misses using DCR only and with various compression techniques.

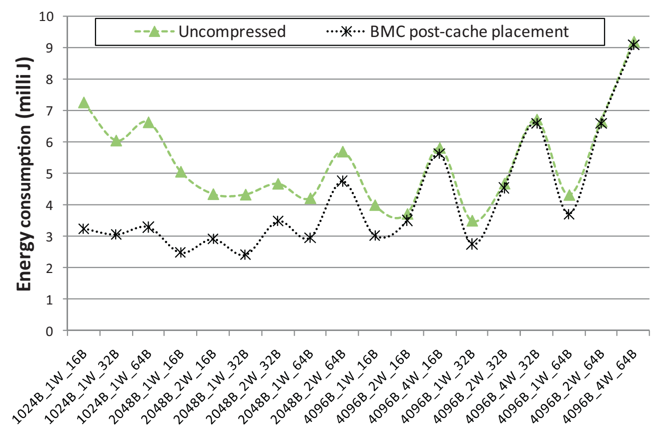


Fig. 12. The impact of cache/line size on energy profile of cache using *cjpeg* benchmark.

for different caches for compressed (using BMC) and uncompressed *cjpeg* benchmark. Using a 4 KB cache with associativity of 4 and 64-bit line size, energy consumption of *cjpeg* benchmark is nearly the same for compressed and uncompressed programs.

In the post-cache placement, compression has a significant effect when combined with small cache sizes. In this case compressed instructions are stored in the cache. Since the compressed code size is 30–45 percent less than uncompressed code it can fit in smaller cache sizes. However, when size of the selected cache increases, the critical portion of program (regardless of whether compressed or not) will fit into cache entirely. Therefore by utilizing large cache sizes energy consumption of the compressed code is very close to uncompressed one. It should be noted that the main objective of exploration is to find the most energy efficient cache configurations so we are not interested in large cache sizes since they require more energy.

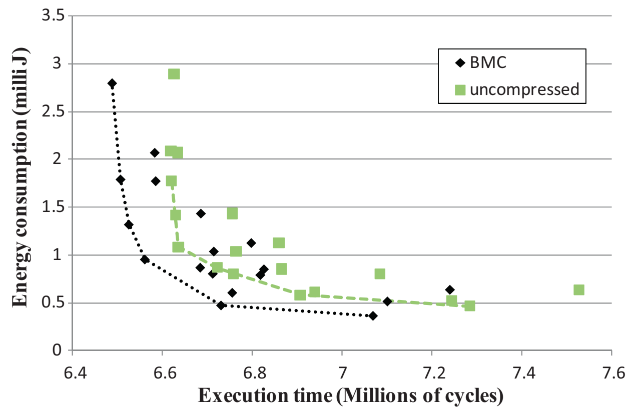


Fig. 11. Performance-energy consumption tradeoff for compressed and uncompressed codes using *rawcaudio* (*adpcm-enc*) benchmark.

Fig. 13 shows performance of applications for different schemes normalized to the *base cache*. Applying DCR alone for the purpose of energy saving, results in 12% performance loss on average. We observe that code compression can improve performance in many scenarios while achieving significant reduction in energy consumption. For instance, in the case of the application *patricia*, applying only DCR would result in 12% performance degradation with 34% energy savings. However, incorporating BMC boosts performance by 33% while gaining extra 17% energy savings on top of DCR achieving 51% energy savings compared to the *base cache*. Results show that synergistic integration of BMC with DCR achieves as much as 75% performance improvement for *g721_enc* (27% improvement on average) compared to DCR alone. Thus it is possible to have a cache architecture that is tuned for applications to have both increased performance as well as lower energy consumption. Fig. 14 shows performance and miss statistics for g721_enc benchmark. Further analysis of g721_enc benchmark reveals that having numerous small *if then else* and switch clauses leads to large number of misses due to overlapping addresses (conflict miss). In this case, compression reduces the number of misses by decreasing the amount of overlap the address of these small code sections. Fig. 14 confirms that compression drastically improves the performance of *g721_enc* benchmark for most of available cache configurations.

Fig. 15 shows performance trend of all cache configurations for both uncompressed and compressed codes for *cjpeg* benchmark. It is interesting to note that compression also improves performance. The compressed program can fit in smaller cache because of 30–45% reduction in code size. This decreases cache misses significantly for small caches. Reduced number of misses can lead to reduced stalls and improved performance. As it can be observed in Fig. 15, without compression, reducing the cache size may lead to major performance degradation so DCR is forced to discard many cache alternatives due to timing constraints. For instance, having timing constraint of 25 million cycles for *cjpeg* benchmark will force to discard all cache configurations of size 2048 KB or lower. However, compression improves the performance significantly when small cache sizes are used. Thus combination of cache reconfiguration and code compression enables energy savings while improving overall performance.

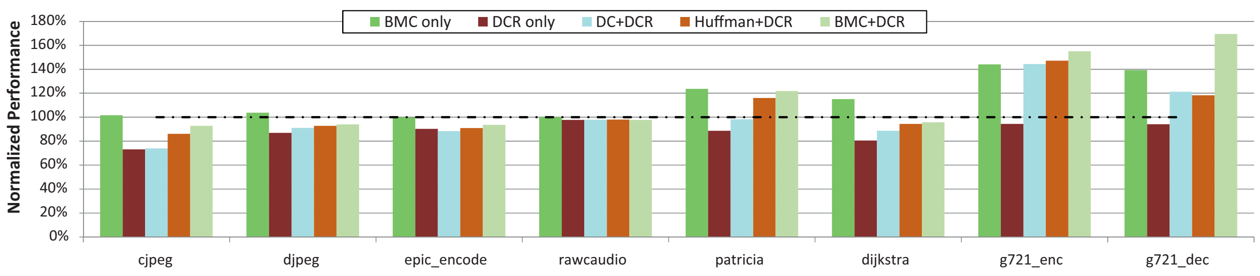


Fig. 13. Performance of the selected "minimal-energy cache" normalized to the base cache.

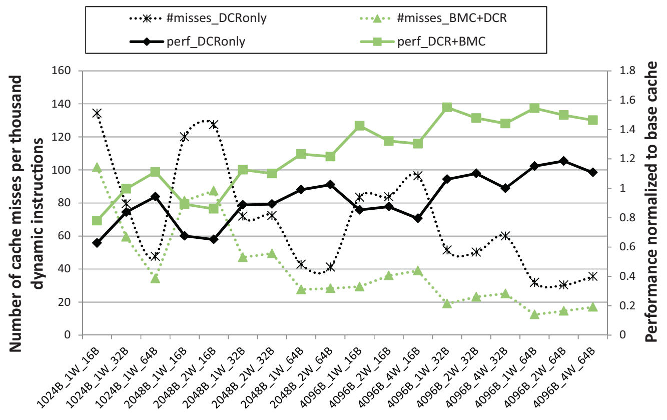


Fig. 14. Number of cache misses using DCR only and DCR + BMC for *g721_enc* benchmark.

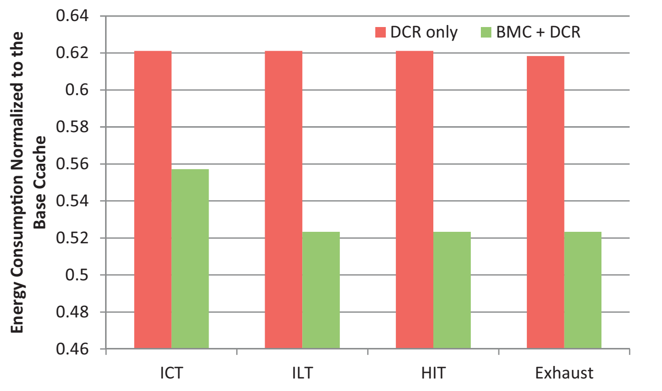


Fig. 16. Cache hierarchy energy consumption using heuristics for *cjpeg* benchmark.

5.3. Two-level cache tuning

To evaluate the effect of two-level cache hierarchy using our exploration heuristics, we selected *cjpeg, djpeg, epic* benchmarks from MediaBench [19] and *crc32* from MiBench [20] benchmark suites. For L2 cache, we choose 8 KB, 16 KB and 32 KB as possible cache sizes; 32, 64 and 128 bytes as line sizes; 4-, 8- and 16-way set associativity with a 32 KB cache architecture composed of four separate banks. L2 cache is unified; in other words, it contains both instructions and data. We define L2 *base cache* to be a 16 KB, 8-way set associative L2 cache with a 64 byte line size. We quantify the cache subsystem energy savings using our approach by comparing to the base cache scenario. We use four cache exploration methods – exhaustive, ICT, ILT, and HIT – to generate profile tables. Fig. 16 presents the total cache hierarchy energy consumption normalized to the base cache for *cjpeg* benchmark using each exploration technique. It can be observed that, for *cjpeg* benchmark, the best results obtained by heuristics are very close to the optimal value obtained by exhaustive search. As we explained in Section 4.1, exploring all possible configurations exhaustively results in 4752 simulations. Performing these set of simulations for *cjpeg* benchmark (which takes the lowest simulation time among others in the benchmark suites) on a system with a 4-core AMD Opteron (an ×86 server processor) running at 3.0 GHz takes more than three days. Clearly, this will take longer for other benchmarks. Although, these heuristics take significantly less time than exhaustive exploration, they provide very close to optimal energy savings. Table 1 presents the total number of cache configurations explored by each exploration heuristic. Our experience is that it may take several days to profile a task using exhaustive method while few minutes if ILT is employed. Designers can decide which heuristic to use based on the static profiling time and the overall energy savings. Therefore, we only perform heuristic space exploration for the remaining benchmarks.

Fig. 17 presents the total cache hierarchy energy consumption normalized to the base cache for *cjpeg*, *djpeg*, *epic*, *patricia* and *dijkstra* benchmarks using each exploration technique for uncompressed and compressed scenarios. ICT achieves best results obtaining 67% average energy saving when applying DCR only. It achieves up to 22% (*patricia* benchmark) more energy savings (11% on average) incorporating compression. ILT reduces the number of simulations significantly but presents results that are slightly inferior. It achieves 62% energy savings using only DCR and up to 20% extra savings adding compression to DCR. HIT outperforms ICT and ILT in *djpeg* benchmark but on average saves 61% of the energy consumption. Integrating compression boosts energy savings achieved by HIT by 7%. The reason for ICT not finding the optimal

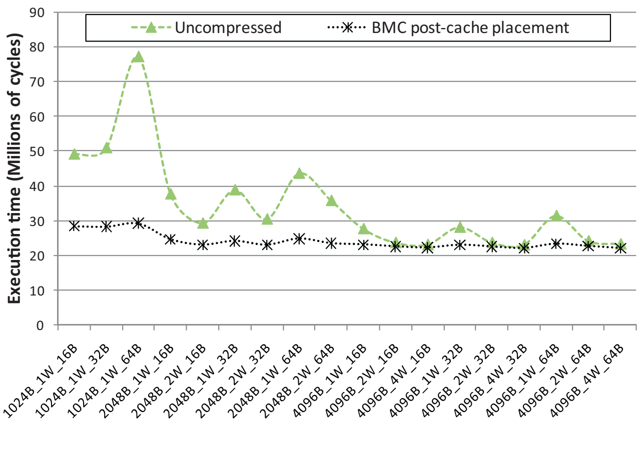


Fig. 15. Performance trend of different cache configurations using *cjpeg* benchmark.

Table 1
Cache hierarchy configuration explored using different exploration methods for *cjpeg* benchmark.

Exhaust	ICT	ILT	HIT
4752	288	18	342

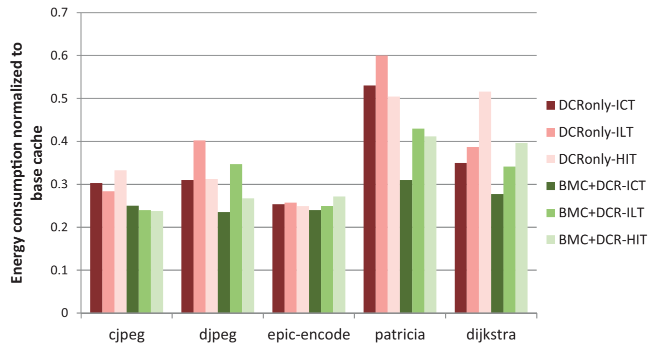


Fig. 17. Cache hierarchy energy consumption using three heuristics.

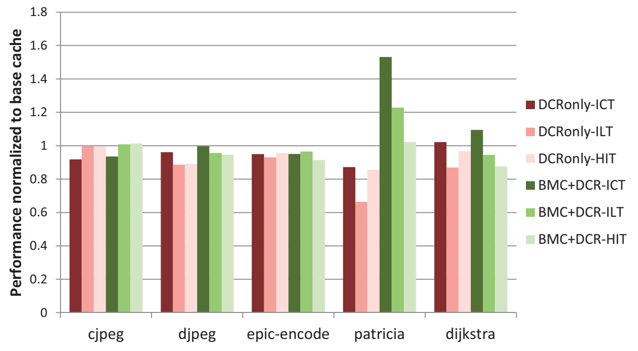


Fig. 18. Performance of the selected "minimal-energy cache" using different heuristics normalized to the base cache.

configurations is that though L1 caches are relatively independent, they both have impact on the L2 cache which has effect back on L1 caches. So they are essentially indirectly dependent on each other through the L2 cache. HIT only considers Pareto-optimal configurations at the cost of losing the chance of finding more efficient cache combinations which actually consists of non-beneficial ones. One of the reasons is that a less energy efficient (due to oversize) L1 cache may cause fewer accesses to L2 cache. Hence an appropriate L2 cache may make this non-beneficial L1 cache overall better. Since ILT is least expensive, it is expected to produce worst results. In reality, it produces comparable, sometimes even better, than some expensive heuristics (HIT).

Fig. 18 shows the performance of selected energy-optimal caches using each heuristic normalized to the *base cache* configuration. On ICT, ILT, and HIT gain up to 16%, 15% and 2% performance improvements when compression is added to DCR.

6. Conclusion

Optimization techniques are widely used in embedded systems to improve overall area, energy and performance requirements. Dynamic cache reconfiguration (DCR) is very effective to reduce energy consumption of cache subsystem. Code compression can significantly reduce memory requirements, and may improve performance in many scenarios. In this paper, we presented a synergistic integration of DCR and code compression for embedded systems. Our methodology employs an ideal combination of code compression and dynamic tuning of two-level cache parameters with minor or no impact on timing constraints. Our experimental results demonstrated 61% reduction on average in overall energy consumption of the cache subsystem as well as up to 75% performance improvement (compared to DCR only) in embedded systems.

References

[1] A. Malik, B. Moyer, D. Cermak, A low power unified cache architecture providing power and performance flexibility, ISLPED (2000).
[2] A. Gordon-Ross, F. Vahid, N. Dutt, Automatic tuning of two-level caches to embedded applications, DATE (2004).
[3] C. Lefurgy, Efficient execution of compressed programs, Ph.D. Thesis, University of Michigan, 2000.
[4] L. Benini, F. Menichelli, M. Olivieri, A class of code compression schemes for reducing power consumption in embedded microprocessor systems, IEEE Transactions on Computers (April) (2004) 467–482.
[5] A. Gordon-Ross, F. Vahid, N. Dutt, Fast configurable-cache tuning with a unified second level cache, in: International Symposium on Low Power Electronics and Design, 2005.
[6] C. Zhang, F. Vahid, W. Najjar, A highly-configurable cache architecture for embedded systems, in: 30th Annual International Symposium on Computer Architecture, June 2003.
[7] P. Vita, Configurable Cache Subsetting for Fast Cache Tuning, in: Design Automation Conference, DAC, 2006.
[8] D.H. Albonesi, Selective Cache Ways: On-Demand Cache Resource Allocation, 2000.
[9] L. Chen, X. Zou, J. Lei, Z. Liu, Dynamically reconfigurable cache for low-power embedded system, in: Third International Conference on Natural Computation, 2007.
[10] C. Zhang, F. Vahid, R. Lysecky, A self-tuning cache architecture for embedded systems, DATE (2004).
[11] A. Wolfe, A. Chanin, Executing compressed programs on an embedded RISC architecture, in: Proc. of the Intl. Symposium on Microarchitecture, 1992, pp. 81–91.
[12] Y. Xie, W. Wolf, H. Lekatsas, A code decompression architecture for VLIW processors, in: 34th ACM/IEEE International Symposium on Microarchitecture (MICRO), 2001.
[13] S. Seong, P. Mishra, Bitmask-based code compression for embedded systems, IEEE Trans. CAD (2008) 673–685.
[14] C. Lin, Y. Xie, W. Wolf, LZW-based code compression for VLIW embedded systems, Proc. DATE (2004) 76–81.
[15] M. Rawlins, A. Gordon-Ross, On the interplay of loop caching, code compression, and cache configuration, ASP-DAC (2011).
[16] D. Burger, T. Austin, S. Bennet, Evaluating future microprocessors:the simplescalar toolset, University of Wisconsin-Madison, Computer Science Department Technical Report CS-TR-1308, July 2000.
[17] C. Murthy, P. Mishra, Lossless Compression Using Efficient Encoding of Bitmasks, isvlsi, IEEE Computer Society Annual Symposium on VLSI, 2009, pp. 163–168.
[18] W. Wang, P. Mishra, Dynamic reconfiguration of two-level caches in soft real-time embedded systems, in: IEEE International Symposium on VLSI, 2009.
[19] C. Lee, M. Potkonjak, W.H. Mangione-smith., MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, in: International Symposium on Microarchitecture, 1997.
[20] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, MiBench: a free, commercially representative embedded benchmark suite, International Workshop on Workload Characterization (WWC), 2001.
[21] CACTI, HP Labs, CACTI 4.2, http://www.hpl.hp.com/.
[22] H. Hajimiri, K. Rahmani, P. Mishra, Synergistic Integration of Dynamic Cache Reconfiguration and Code Compression in Embedded Systems, International Green Computing Conference (IGCC), 2011.