Combining Code Reordering and Cache Configuration

ANN GORDON-ROSS, University of Florida
FRANK VAHID, University of California, Riverside
NIKIL DUTT, University of California, Irvine

The instruction cache is a popular optimization target due to the cache's high impact on system performance and power and because of the cache's predictable temporal and spatial locality. This article is an in depth study on the interaction of code reordering (a long-known technique) and cache configuration (a relatively new technique). Experimental results show that code reordering coupled with cache configuration reveals additional energy savings as high as 10–15% for several benchmarks with reduced cache area as high as 48%. To exploit these additional benefits, we architect and evaluate several design exploration heuristics for combining these two methods.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms: Design

Additional Key Words and Phrases: Configurable cache, code reordering, code reorganization, code layout, cache hierarchy, cache exploration, cache optimization, low power, low energy, architecture tuning

ACM Reference Format:
Gordon-Ross, A., Vahid, F., and Dutt, N. 2012. Combining code reordering and cache configuration. ACM Trans. Embedd. Comput. Syst. 11, 4, Article 88 (December 2012), 20 pages.
DOI = 10.1145/2362336.2399177 http://doi.acm.org/10.1145/2362336.2399177

1. INTRODUCTION AND MOTIVATION

Extensive past research for improving microprocessor performance and power has focused on the instruction cache, due to the cache's large impact on those design factors. Proposed optimization techniques typically exploit an instruction stream's spatial and temporal locality. Popular techniques include prefetching, victim buffers [Zhang and Vahid 2004a], filter caches [Hines et al. 2007; Kin et al. 1997], loop caches [Gordon-Ross et al. 2002; Gordon-Ross and Vahid 2002; Lee et al. 1999b], code compression [Benini et al. 1999], cache configuration [Balasubramonian et al. 2000; Gordon-Ross et al. 2008; Gordon-Ross et al. 2009; Zhang and Vahid 2003, 2004b], and code reordering [Kalmatianos and Kaeli 1999; Petis and Hanson 1990].

Most optimization techniques are proposed independently of other techniques, and, with so many techniques available, the interplay of those techniques is important to study as embedded system designers under tight time-to-market constraints cannot be expected to undergo this time-consuming examination themselves. Studying the

This research was supported in part by the National Science Foundation under Grant Nos. CNS-0953447, CCR-0203829, and CCR-9876006.
Authors' addresses: A. Gordon-Ross (corresponding author), Department of Electrical and Computer Engineering and CHREC, University of Florida 32611; email: ann@ece.ufl.edu; F. Vahid, Department of Computer Science and Engineering, and Center for Embedded Computing Systems, University of California, Riverside, CA 92521; N. Dutt, School of Information and Computer Science, University of California, Irvine, CA 92697.

DOI 10.1145/2362336.2399177 http://doi.acm.org/10.1145/2362336.2399177

interplay may demonstrate that one technique dominates over another technique, perhaps making the second technique unnecessary. If two techniques are complementary, then finding the most effective combination of the techniques is useful. In this article, we examine the interplay of a major software-based approach, code reordering, and a major hardware-based approach, cache configuration.

Code reordering/reorganization/layout at the basic block-level is a mature technique developed in the late 1980's to tune an instruction stream to the instruction cache in order to improve cache hit rates and improve the cache's utilization. This widely researched technique increases performance on average, but decreases performance for some benchmarks. Code reordering places an application's hot-path instructions (frequently executed instructions) contiguously in memory, thus moving infrequently executed instructions so that they are not inadvertently fetched into the cache (via cache prefeteching techniques or simply as apart of the same cache line as a hot-region instruction). Code reordering also reduces conflict misses through procedure placement. Initial code reordering techniques were non-cache-aware, meaning these techniques did not consider the specific cache configuration when applying code transformations. Enhanced code-reordering techniques are cache-aware and provide improved performance over non-cache-aware techniques, but cache-aware techniques require a priori knowledge of the target cache configuration. Typically, code reordering is a compile-time or link-time optimization requiring profile information to determine an applications hot path. Runtime/dynamic methods for code reordering also exist [Chen and Leupen 1997; Huang et al. 2006a, 2006b; Scales 1998], resulting in a simpler tool flow, but they incur some runtime overhead. Several previous works evaluate code reordering impacts, such as the impact of the instruction set [Chen and Zhang 2007] and instruction fetch architectures [Ramirez et al. 2001] on code reordering, the impact code reordering has on branch prediction [Ramirez et al. 2000], code placement for improving branch prediction accuracy [Ramirez et al. 2005], and the combined effects of code ordering and victim buffers [Bahar et al. 1998].

Due to new hardware technologies and core-based design methodologies, cache configuration is a more recently developed technique that tunes a cache's parameters, such as total size, associativity, and line size, to an application's instruction stream for decreased energy consumption and/or increased performance. Applications have distinct execution behaviors that warrant distinct cache requirements [Zhang et al. 2003]. Reducing a cache's size or associativity just enough, but not too much, can minimize an application's energy consumption. Tuning the line size, larger for localized applications, smaller for non-localized applications, can reduce energy and also improve performance. Caches may be configured in a core-based methodology in which a designer synthesizes a customized cache along with a microprocessor [Altera 2010; Arc 2010; Arm 2010]; Mips 2010; Tensilica 2010]. On the other hand, caches in predesigned chips may be hardware configurable, with configuration occurring by setting register bits during system reset or during runtime [Albonesi 2002; Malik et al. 2000; Zhang and Vahid 2004b]. To determine the lowest energy and/or best performance as defined by the system optimization goals, the cache configuration design space can be explored exhaustively or a search heuristic can be used.

Code reordering and cache configuration can be applied at different times during application design. Code reordering is typically carried out during design time as a designer guided step, while cache configuration can be applied at design time or easily applied during runtime. For code reordering, the designer must compile and profile the code and then generate an optimized executable by either recompiling the code or using a link-time code optimizer. Sanghai et al. [2007] presented a framework to provide code layout assistance for embedded processors with configurable memories. Dynamic procedure placement [Scales 1998] is possible but has received little attention

due in part to potentially significant runtime overhead. Cache configuration can also be applied as a designer-guided step, however, recent research focuses on cache configuration during runtime to eliminate the need for designer intervention [Zhang and Vahid 2004b] with little to no runtime overhead. Designer-guided optimization steps increase the complexity of the design task, whereas runtime optimization requires no special design efforts and also ensures optimizations use an application's real dataset.

The methods of code reordering and of cache configuration were largely researched independently in the past, due in part to the fact that the methods were developed by distinct research communities. However, the interaction between the two tuning approaches, that is, whether they complement, degrade, or obviate the need for each other, has not been considered before and needs to be addressed. In this article, we perform a compressive study of the interplay of both non-cache-aware and cache-aware code reordering and cache configuration. While results show that cache configuration largely dominates code reordering, particular benchmarks benefit from applying both techniques in terms of increased energy consumption reduction and cache area reduction as compared to cache configuration alone. Since highly constrained embedded systems would benefit from the combined savings, we developed and compared various design exploration heuristics for combining cache-aware code reordering and cache configuration.

2. INSTRUCTION CACHE OPTIMIZATIONS

In this section, we provide background on both non-cache-aware and cache-aware code reordering, cache configuration, and motivate the combined study of these interrelated techniques.

2.1. Non-Cache-Aware Code Reordering Background And Code Reordering Example

Much previous research exists in the area of code reordering (also referred to as code layout and code reorganization in the literature) at the basic-block-, loop-, and procedure-level reordering. Code reorganization at the basic-block-level dates back to early work in 1988 by Samples and Hilfinger [1988]. Early work by McFarling [1989] used basic block execution counts to reorder code at the basic-block-level and exclude infrequently used instructions from the cache. Pettis and Hansen [1990] and Hwu and Chang [1989] presented similar methods for both basic block and procedural reordering using edge profile information showing performance benefits of nearly 15%. The work by Pettis and Hanson [1990] serves as the basis for the majority of all modern code reordering methodologies.

Many modern tools implement the Pettis and Hanson [1990] code reordering methodology directly or in a manner [Cohn et al. 1997; Cohn and Lowney 2000; Gloy et al. 1997; Lee et al. 1999a; Muth et al. 2001; Ramirez et al. 2005]. Much research focuses on improving the Pettis and Hanson methodology to include cache line coloring to reduce conflict misses [Aydin and Kaeli 2000; Hashemi et al. 1997; Kalmatianos and Kaeli 1999, 2000] by placing popular procedures in the memory such that the number of overlapping cache lines in minimized. Also, various production tools offer Pettis and Hanson-based code reordering as an optimization option such as Compaq's Object Modification Tool (OM) [Srivastava and Wall 1992], its successor Spike [Cohn et al. 1997], and IBM's FDPR [Schmidt et al. 1998]. Other works provide methodologies for applying code reordering to commercial applications [Ramirez et al. 2002].

For several of the experiments presented in this article, we use the Pentium Link-Time Optimizer (PLTO) [Scharz et al. 2001], which performs code reordering using an improved Pettis and Hanson algorithm. In the Pettis and Hanson algorithm, basic blocks are reordered to reduce the number of taken branches and to reduce the number of misses in the instruction cache by increasing instruction locality. For example, loop

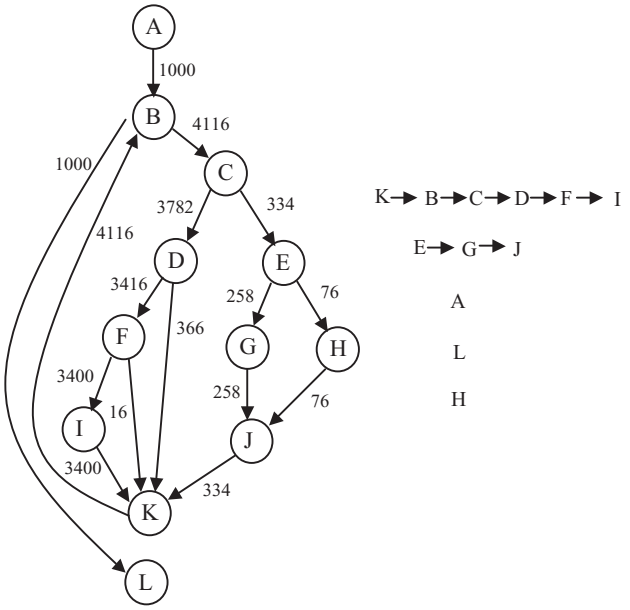

Fig. 1. Edge weighted control flow graph and resulting basic block chains.

bodies frequently contain an error condition that is checked in each loop iteration and the error code is infrequently executed. The error handling code is loaded into the instruction cache, polluting the instruction cache with code that may never be fetched. Code reordering moves infrequently executed code out of the loop body, replacing the code with a jump to the relocated code. Additionally, a jump is inserted at the end of the relocated code to transfer control back to the loop body.

Basic block reordering uses profile information to guide placement of basic blocks. The goal is to form chains of basic blocks that are to be placed as straight-line code. More successful code reordering methods use edge profiling as opposed to basic block profiling. Edge profiling counts the number of times each arc in a control flow graph is taken, while basic block profiling simply counts the number of times each basic block is executed. Edge profiling supplies more information on the flow of execution through the control flow graph so basic blocks that are frequently executed in sequence (have a high arc weight) can be grouped together in the final code layout.

The basic block reordering methodology used in this article is based on the bottom-up positioning algorithm as described by Pettis and Hanson [1990]. Initially, a control flow graph is created for the application with arcs annotated with their corresponding execution frequency. Figure 1 shows a sample control flow graph for a loop. The bottom-up algorithm begins with each basic block as the head and tail of a basic block chain. Next, each arc in the graph is processed from largest to smallest. The basic blocks at the source and destination of the arc are merged to form a new chain if either of the basic blocks is the tail of one chain and the other basic block is the head of different chain. If either the source or the destination basic block is not a head or tail of a chain, the basic blocks may not be connected. In this case, a new chain is formed.

Figure 1 shows the basic block chains that are formed after applying the bottom-up algorithm to the control flow graph. After the set of chains are determined, the chains are ordered in the executable based on the heaviness of the interconnecting edges. Additionally, unconditional branches are added to the code to maintain correctness.

PLTO implements a variation of the Pettis and Hanson bottom-up algorithm. PLTO improves upon the original Pettis and Hanson algorithm in two ways. The first improvement addresses minor modifications needed to address problems identified by Calder et al. [1994]. The improvements deal with branch alignment to benefit the underlying fetch architecture and branch predictor. The second improvement deals with the formation of the basic block chains. The basic blocks are grouped into three different sets: the hot set, the zero set, and the cold set. The hot set contains basic blocks that account for a threshold percentage of the execution time of the application. The zero set contains all basic blocks that are never executed, and all remaining basic blocks are placed in the cold set. Basic block chains from each set are determined using the Pettis and Hanson bottom-up algorithm. The chains from each set are then concatenated to form the final layout.

2.2. Cache-Aware Code Reordering

The Pettis and Hanson bottom-up algorithm was designed to exploit a single-level direct-mapped cache. Basic block reordering is performed without any attention paid to how the ordering may cause contention in a set associative cache or how code reordering effects conflicts in the other levels of the cache hierarchy. More complex algorithms extend code reordering to include cache-aware code placement [Bartolini and Prete 2005] and multiple levels of cache [Gloy et al. 1997]. For our study, we use one of the most advanced cache-aware code reordering tools, developed by [Kalmatianos and Kaeli 1999, 2000], which performs procedure reordering and cache line coloring. Using an instruction trace file of the application, the tool constructs a conflict miss graph, which is an undirected graph where each node represents a procedure. Every edge between two procedures is weighted with an estimation of the worst-case number of conflict misses between those two procedures. Conflict misses can only occur if the two procedures are simultaneously live, that is, both procedures occupy the cache at the same time. Next the tool prunes the conflict miss graph to remove unpopular edges and removes procedures from the graph with no edges remaining after pruning. Then the tool applies cache line coloring to place the procedures into the cache such that the number of conflict misses between simultaneous live procedures is minimized. Pairs of nodes are processed from the graph by decreasing edge weights. During this step, the tool is conscious of the placement of the procedures in main memory to keep the memory footprint as small as possible.

2.3. Cache Configuration/Tuning

Su and Despain [1995] showed in early work that the memory hierarchy is very important in determining the power and performance of an application. Recently, Zhang and Vahid [2004b] showed the vastly different cache configurations required to achieve minimal energy consumption by the cache. If a cache does not reflect the requirements of an application, excess energy may be consumed. For example, if the cache size is too large for an application, excess energy will be consumed fetching from the large cache. If the cache size is too small, excess energy may be consumed due to thrashing—the working set of an application is constantly being swapped in and out of the cache. Tunable parameters normally include cache size, line size, and associativity. However, other parameters such as the use of a victim buffer, instruction/data encoding, bus width, etc. could also be included as tunable parameters

Recent advances in research and technology have made the configurability of cache parameters possible. The availability of a tunable cache enables designers to specify cache parameters in a tunable core-based design for custom system synthesis [Altera 2010; Arc 2010; Arm 2010; Mips 2010; Tensilica 2010]. Additionally, predesigned chips

may contain hardware that supports cache configuration during system reset or even during runtime [Albonesi 2002; Malik et al. 2000; Zhang and Vahid 2004b].

Platune [Givargis and Vahid 2002] used an exhaustive method to search the cache configuration design space for reduced energy consumption. Whereas an exhaustive method produces optimal results, the time needed to exhaustively search the design space may not be available. To decrease exploration time, heuristic methods exist to explore the design space. Palesi and Givargis [2002] presented an extension to Platune that explored the design space using a genetic algorithm and produced near-optimal results in a fraction of the time. Ghosh and Givargis [2003] presented a method for directly computing cache parameters given design constraints. Balasubramonian et al. [2000] presented a runtime method for redistributing the cache size between the various cache levels. Zhang and Vahid [2003] presented a prototyping methodology for level-one cache configuration exploration for Pareto-optimal points trading off energy and performance. Further work by Zhang and Vahid [2004b] showed a methodology for runtime configuration of the cache for reduced energy consumption resulting in energy savings of 45% to 55% on average. Gordon-Ross et al. [2009] developed a heuristic for quickly searching a two-level configurable cache hierarchy for separate instruction and data caches showing energy savings averaging 53%

In this article, we use the configurable cache tuning methodology described by Zhang et al. [2003] and Zhang and Vahid [2003] and extended by Gordon-Ross et al. [2009] for a single-level cache configuration. Zhang's method is designed with runtime application in mind but is also applicable to a design time simulation environment. We chose to use Zhang's method so as not to rule out future work on code reordering and cache configuration during runtime. Zhang's heuristic quickly explores the design space producing near-optimal results. Tuning methods exist for exploration of two levels of cache, but we chose to explore only a single level of cache because many embedded systems contain only a single level of cache. Additionally, cache configuration at design time is applicable to an embedded environment where many systems run a single application for the lifetime of the device.

The cache configuration heuristic utilized in this article efficiently explores the cache parameters based on their impact on total system energy and miss rate, and it is based on the heuristic originating with Zhang and Vahid [2004b]. The heuristic explores cache parameters having a larger impact on system energy and miss rate before parameters having a smaller impact. The heuristic is summarized in the following text. Each parameter is explored from smallest to largest.

—Holding the cache line size and associativity at their smallest values, determine the cache size yielding the lowest energy consumption.
—Fixing the cache size at the size determined in the previous step and the associativity at the smallest value, determine the cache line size yielding the lowest energy consumption.
—Fixing the cache size and the cache line size at the values determined in the previous steps, determine the cache associativity yielding the lowest energy consumption.

This heuristic searches 7 of the 18 possible configurations given the cache parameters we have chosen. We found that this heuristic determines the optimal cache configuration in most cases. From this point forward, the heuristically determined cache configuration will be referred to as the *best cache configuration*.

To enable the cache tuning heuristic to tune during runtime, a tunable cache is necessary. Zhang et al. [2003] describe tunable cache hardware and present verification of the hardware layout of this configurable cache. Figure 2(a) shows the configuration cache tuning architecture. Figure 2(b) shows the base cache for four separate banks that may be turned on, concatenated with another bank (Figure 2(c)), or turned off

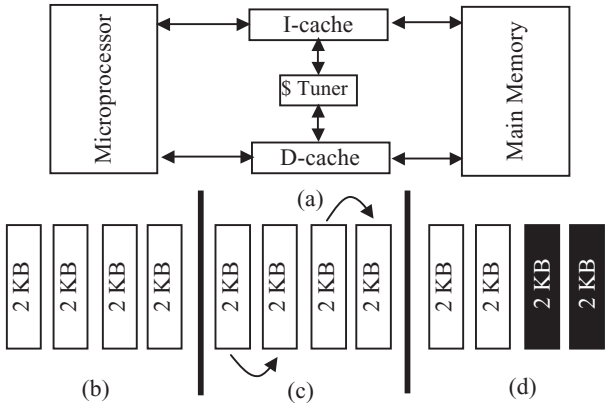

Fig. 2. (a) Configurable cache architecture; (b) 8KB 4-way base cache with for 2KB sub banks; (c) 8KB 2-way cache using way concatenation; (d) 4KB 2-way cache using way shutdown.

(Figure 2(d)) via a configuration register. Due to the use of configurable banks, certain cache configurations are not possible. For instance, if a base cache size of 8KB is desired, four banks of 2KB each will be utilized. An 8KB direct-mapped, 2-way, and 4-way set associative cache is available. To reduce the cache size to 2KB, three of the banks must be shut down leaving one remaining bank of 2KB. Since banks are used to increase associativity and there is only one bank available in a 2KB cache, only a direct-mapped cache is available for 2 KB. Further details are available in Zhang et al. [2003].

2.4. Code Reordering and Cache Configuration Tradeoffs

For completeness in our study, we explore both non-cache-aware and cache-aware code reordering techniques. Whereas cache-aware code reordering is state-of-the-art and typically produces increased performance benefits as compared to non-cache-aware code reordering, cache-aware code reordering can significantly increase system design time as system designers are required to perform cache configuration (cache configuration requires numerous simulations, while code reordering only requires one simulation). For non-cache-aware code reordering, the system designer may use a runtime cache configuration technique to eliminate design time cache configuration.

While we evaluate the two methods with respect to performance and energy, other trade-offs exist between the two methods. The hardware-based cache configuration has the advantage of being applicable to any binary, requiring no special compiler or linker, but it requires either a configurable cache architecture or a custom-synthesized cache design. The software method of code reordering has the advantage of being applicable to any off-the-shelf microprocessor cache. But the software method requires a special compiler and/or linker, which may disrupt standard tool flows. Both methods, in their commonly proposed forms, require software profiling, a common, but not universally-used, step in software development. Both methods have been proposed in dynamic and hence transparent forms, though dynamic code reordering involves the more complex task of dynamic binary modification.

3. NON-CACHE-AWARE CODE REORDERING AND CACHE CONFIGURATION

First, we explore the interplay of non-cache-aware code reordering and cache configuration. In this section, we describe our evaluation framework (which is similar to the framework used for our study using cache-aware code ordering in Section 4.1) and evaluate our results.

3.1. Evaluation Framework

To determine the combined effects of code reordering and cache configuration, we used 26 benchmarks: twelve benchmarks from the Powerstone benchmark suite [Malik et al. 2000], three benchmarks from the MediaBench benchmark suite [Lee et al. 1997], and eleven benchmarks from the EEMBC benchmark suite [EEMBC 2010]. For each benchmark suite, we report data for every benchmark that successfully ran through the compilation and simulation tools we utilized. Some benchmarks would not compile, would not run through the tools, or would not execute correctly after code reordering was applied. For all benchmarks, we used the provided input vectors.

We used PLTO [Scharz et al. 2001] to perform code reordering on the applications. PLTO is similar to the popular ALTO [Muth et al. 2001] tool but works with the x86 architecture instead of the Alpha architecture. We performed the following steps to produce code reordered executables.

(1) Compile the code with flags specifying the inclusion of the symbol table and relocation information and to not patch any of the instructions. Libraries are statically linked.
(2) Invoke PLTO to instrument the executable to gather edge profiles.
(3) Run the instrumented executable to produce a file containing the edge counts.
(4) Rerun PLTO with edge profiles and perform code reordering.

PLTO offers many other link-time optimizations. To ensure that we only explored code reordering, we turned off all other optimizations at the command line. Additionally, for comparison purposes, we created executables without code reordering using the same steps as just described except that in step 4, we turned off the code reordering optimization.

We used Perl scripts to drive the cache tuning heuristic along with an instruction cache simulator to determine cache statistics. Most x86 cache simulators are trace driven, requiring an instruction trace file for execution. Due to the long execution time of some of the benchmarks studied, trace-driven cache simulation would be cumbersome. To alleviate the need for instruction traces, we obtained a trap-based profiler from the University of Arizona to perform execution-driven cache simulation [Moseley 2003]. The trap-based profiler combines the trace cache simulator Dinero IV [Dinero 2010] and PLTO to create an execution-driven cache simulation. The trap-based profiler executes the application using PLTO, traps instruction addresses, and passes the instruction addresses to Dinero.

We determine energy consumption for a cache configuration for both static and dynamic energy using the following model.

```
total_energy=static_energy+dynamic_energy
dynamic_energy=cache_hits*hit_energy+cache_misses*miss_energy
miss_energy=offchip_access_energy+miss_cycles*CPU_stall_energy+cache_fill_energy
miss_cycles=cache_misses*miss_latency+cache_misses*memory_bandwidth
static_energy=total_cycles*static_energy_per_cycle
static_energy_per_cycle=energy_per_Kbyte*cache_size_in_Kbytes
energy_per_Kbyte=((dynamic_energy_of_base_cache*10%)/base_cache_size_in_Kbytes)
```

We used Cacti [Reinman and Jouppi 1999] to determine the dynamic energy consumed by each cache fetch for each cache configuration using 0.18-micron technology. The trap profiler provided us with the cache hits and cache misses for each cache configuration. Miss energy determination is quite difficult because it depends on the off-chip access energy and the CPU stall energy, which are highly dependent on the actually system configuration used. We could have chosen a particular system configuration and obtained hard values for the *CPU_stall_energy*, however, our results would only apply to one particular system configuration. Instead, we examine the stall energy for several microprocessors and estimate the *CPU_stall_energy* to be 20% of the active

energy of the microprocessor for this study. We obtain the *offchip_access_energy* from a standard low-power Samsung memory. To obtain miss cycles, the miss latency and bandwidth of the system is required. We estimate a cache miss to take 40 times longer than a cache hit to transfer the first block (16 bytes) and subsequent blocks (each additional 16 bytes) would transfer in 50% of the time it took to transfer the first block. Previous work [Gordon-Ross et al. 2009] showed that cache tuning heuristics remain valid across different configurations of miss latency and bandwidth. We determine the static energy per Kbyte as 10% of the dynamic energy of the base cache divided by the base cache size in Kbytes.

We chose cache parameters to reflect those available in typical embedded processors. We explore cache sizes of 2, 4, and 8 Kbytes, cache line sizes of 16, 32, and 64 bytes, and set associativities of direct-mapped, 2-way, and 4-way.

For comparison purposes, we generated cache statistics for every cache configuration for every benchmark, with and without code reordering, to determine the optimal cache configuration. We found that, in every case, the tuning heuristic determined the optimal cache configuration. From this point forward, we will refer to the heuristically determined cache configuration as the optimal cache configuration given that the two yield identical results for every benchmark.

To determine cache energy savings due to cache configuration, normally a large cache is used as a base cache for comparison purposes. The cache size reflects a common configuration likely to be found in a platform to accommodate a wide range of target applications. However, research shows that code reordering is most effective for small to medium cache sizes [McFarling 1989] because an application may entirely fit into too large of a cache; only in a small cache do we see large numbers of conflict misses. To best show the benefits of code reordering, we have chosen the smallest cache as our base cache configuration, a 2Kbyte direct-mapped cache with a line size of 16 bytes. This small cache size is not too small as to be dominated by capacity misses. Using the smallest cache possible is also a goal of many cost-constrained embedded systems.

3.2. Experiments

We explore the interplay of code reordering and cache configuration by producing four energy and performance results for each benchmark. The results include energy and performance values for the base cache configuration for each benchmark without code reordering and for each benchmark after code reordering has been performed. Additionally, we apply cache configuration to each benchmark without code reordering and for each benchmark after code reordering has been applied.

3.2.1. Energy and Performance Evaluation. Figure 3 shows the energy savings and Figure 4 shows the performance impact of cache configuration both with and without code reordering. All values have been normalized to the base cache configuration without code reordering for each benchmark. Overall, results show a similar trend for both energy and performance as expected, since code reordering simply reduces the number of cache misses. For code reordering alone, average energy savings and execution time reduction are approximately 3.5% over all benchmarks. However, the averages include two benchmarks, *CACHEB* and *CANRDR* where code reordering performs very poorly. Removing these two benchmarks from the average increases the average energy savings and execution time reduction to approximately 9%, closely reflecting results obtained in previous research [Muth et al. 2001]. (Ultimately, a designer would hopefully be able to detect the decreased performance of code reordering on a program and then choose to not apply reordering on that program).

When cache configuration is applied to the benchmarks, Figure 3 and Figure 4 show that on average both the energy savings and performance benefits are nearly

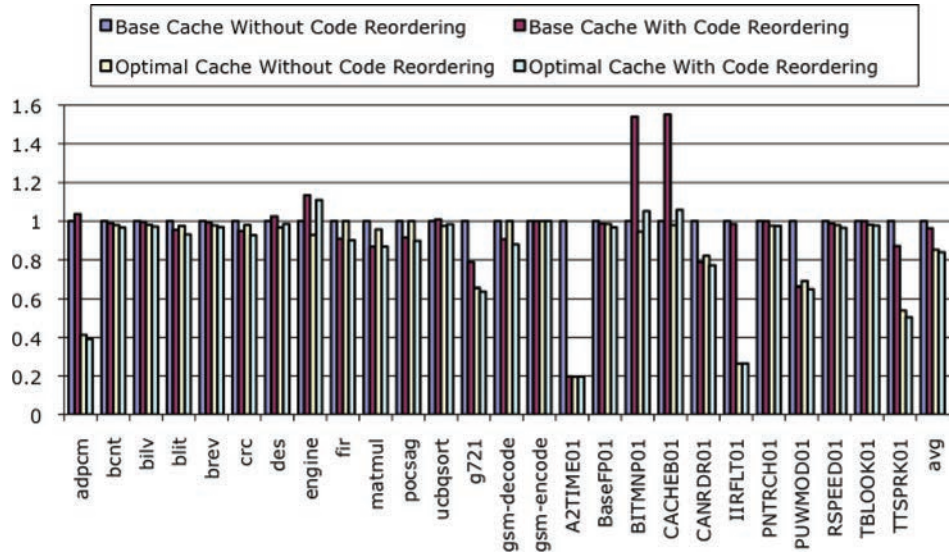

Fig. 3. Energy consumption with code reordering and cache configuration. Energy for each benchmark is normalized to the energy consumption of the base cache without code reordering.

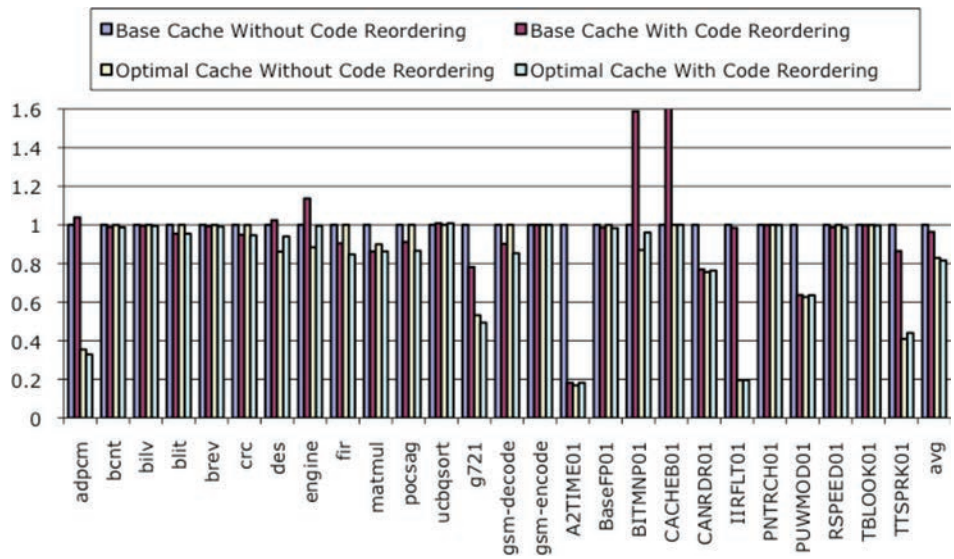

Fig. 4. Execution time with code reordering and cache configuration. Execution time for each benchmark is normalized to the execution time of the base cache without code reordering.

identical for cache configuration without code reordering and cache configuration with code reordering. Energy savings in the instruction cache obtained due to cache configuration is on average 15% without code reordering and 17% with code reordering over all benchmarks, a minor difference. Likewise, execution time reduction with cache configuration averages 17% without code reordering and 18.5% with code reordering over all benchmarks, again, a minor difference. From these results, we might conclude that the benefits due to code reordering are nearly negated when cache configuration

is used. A designer can thus eliminate the special tools, profiling setup, and time required to perform code reordering if runtime cache configuration is available. The additional savings due to adding code reordering to cache configuration are nominal and probably not worth the extra design effort required by the designer. Runtime cache configuration produces the benefits without designer effort.

Additionally, we observed a very interesting trend across all benchmarks. As Figure 3 and Figure 4 show, in a few benchmarks, both energy and execution time are increased when code reordering is applied. However, when cache configuration is applied along with code reordering, there is no execution time degradation for any benchmark. Execution time for each application is either as good or better than the base cache configuration with no code reordering. Cache configuration thus alleviates some of the negative performance impacts some applications incur due to code reordering.

3.2.2. Change in Cache Requirements Due to Code Reordering. Table I[1] shows the optimal cache configuration for each benchmark without code reordering and with code reordering, and the change in cache area due to code reordering. Cache area numbers were obtained using Cacti [Reinman and Jouppi 1999]. This information shows the effectiveness of code reordering in increasing the spatial locality of an application. In 30% of the benchmarks, code reordering results in an optimal cache having a larger line size. These cases are marked in bold in the *With Code Reordering* column. A larger line size in the optimal cache means that the configurable cache tuning algorithm found that a larger line size improved the cache hit rate, which in turn means that code reordering successfully placed linearly-executed blocks next to one another spatially.

Table I also shows that code reordering successfully increases the overall effectiveness of the optimal cache in 22% of the benchmarks, resulting in a smaller cache size. These cases are underlined in the *With Code Reordering* column. In only one case, *engine*, did code reordering actually increase the size of the optimal cache. The *Change in Area* column in Table I shows the overall change in optimal instruction cache area due to code reordering. Positive change denotes an increase in cache size, while a negative change denotes a decrease in cache size. Overall, we observed a 4% decrease in optimal instruction cache area due to code reordering. The decrease in cache size reveals an optimization available for small custom synthesized embedded systems with very tight area constraints. From this data, we can conclude that code reordering and cache configuration can be used to reduce the area devoted to the instruction cache by as much as 48%.

4. CACHE-AWARE CODE REORDERING AND CACHE CONFIGURATION

Next, we explore the interplay of cache-aware code reordering and cache configuration. We present our evaluation framework, associated design exploration heuristics, and evaluate our results.

4.1. Evaluation Framework

Our evaluation framework is similar to that described in Section 3.1, thus we limit our discussion here to the differences. Due to the use of a different tool suite, our benchmark suite is slightly different and included 30 embedded system benchmarks: eleven benchmarks are from the Powerstone benchmark suite [Malik et al. 2000], three benchmarks are from the MediaBench benchmark suite [Lee et al. 1997], and sixteen benchmarks are from the EEMBC benchmark suite [EEMBC 2010]. We could only

[1]The results in Table I differ from those published in "A First Look at the Interplay of Code Reordering and Configurable Caches" at the ACM Great Lakes Symposium on Very Large Scale Integration (GLSVLSI), 2005. In the previous article, the *Change in Area* column presented the increase in cache area requirements if cache configuration was not used, and that calculation was inadvertently labeled.

Table I. Optimal Cache Configuration For All Benchmarks with Code Reordering and without Code Reordering Configurations are noted as cache size followed by associativity followed by line size. Bold configurations denote cases where code reordering resulted in a larger line size. Underlined configurations denote cases where code reordering resulted in a smaller cache size.

	Without Code Reordering	With Code Reordering	Change in Area
adpcm*	4k1w32	4k1w64	10%
bcnt*	2k1w64	2k1w64	0%
bilv*	2k1w32	2k1w64	17%
blit*	2k1w64	2k1w64	0%
brev*	2k1w64	2k1w64	0%
crc*	2k1w64	2k1w64	0%
des*	8k1w16	4k1w16	−44%
engine*	4k1w16	8k1w16	78%
fir*	2k1w16	2k1w32	10%
matmul*	4k1w16	2k1w16	−48%
pocsag*	2k1w16	2k1w32	10%
ucbqsort*	2k1w64	2k1w64	0%
g721**	8k1w16	8k1w32	8%
gsm-decode**	2k1w16	2k1w64	27%
gsm-encode**	2k1w16	2k1w16	0%
A2TIME***	4k1w16	2k1w32	−42%
BaseFP***	2k1w32	2k1w64	17%
BITMNP***	4k1w64	4k1w32	−9%
CACHEB***	2k1w64	2k1w64	0%
CANRDR***	4k1w64	2k1w32	−41%
IIRFLT***	8k1w64	8k1w64	0%
PNTRCH***	2k1w64	2k1w64	0%
PUWMOD***	4k1w64	2k1w32	−41%
RSPEED***	2k1w64	2k1w64	0%
TBLOOK***	2k1w64	2k1w64	0%
TTSPRK***	8k1w64	4k1w32	−45%
		avg	−4%

*Powerstone **Mediabench ***EEMBC.

explore three benchmarks from the MediaBench benchmark suite because the other benchmarks produced trace files too large to process in a reasonable amount of time by the code reordering tool.

The code reordering tool we used [Kalmatianos and Kaeli 1999, 2000] required as input an instruction trace of the application to gather profile information, the location of all procedures in the application, a list of all active procedures in the application, and the cache configuration. To obtain the instruction trace file, we modified the sim-cache portion of SimpleScalar [Burger et al. 2000] to output each instruction address during execution. We obtained location information of all active and inactive procedures using Looan [Villarreal et al. 2001], a loop analysis tool that takes as input an application binary and the instruction trace and outputs the location of all loops and procedures in the application along with execution frequencies. A header file provides the cache configuration information to the code reordering tool.

The code reordering tool first performs code reordering for the cache configuration and then simulates the instruction cache to obtain cache statistics. For tests that involved reordering the code for a configuration other than the current cache configuration being explored, we modified the tool so that both the cache configuration to reorder for and the actual cache configuration to simulate could be different.

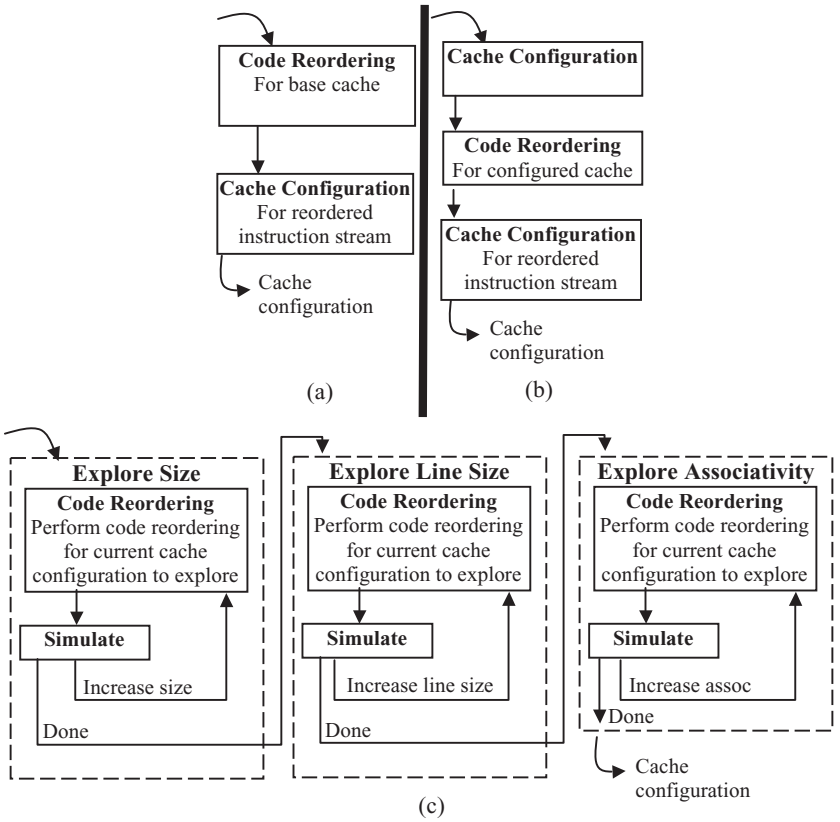

Fig. 5. Flow charts depicting optimization methods for the (a) reorder-configure heuristic, (b) the configure-reorder-configure heuristic, and (c) the reorder-during configuration heuristic.

4.2. Design Exploration Heuristics to Combine Cache-Aware Code Reordering and Cache Configuration

We also sought to develop the best heuristic for combining the two methods and to compare that heuristic to each method applied alone. We considered three possible exploration heuristics—reorder-configure, configure-reorder-configure, and reorder-during-configuration—which we now describe.

4.2.1. Reorder-Configure. Figure 5(a) shows the reorder-configure heuristic. This heuristic first performs code reordering and then applies cache configuration to the reordered code. However, since cache-aware code reordering must have the cache configuration as input, difficulty arises in choosing which cache configuration to reorder for before the best cache configuration is known. We chose to perform code reordering for the base cache configuration since our results will be compared to the base cache configuration. After code reordering, the cache configuration heuristic configures the cache for optimal energy consumption given the reordered instruction stream. With one code reordering round and one cache configuration round, this heuristic is the fastest to apply.

4.2.2. Configure-Reorder-Configure. The drawback of the reorder-configure heuristic is that the cache configuration must be arbitrarily chosen for cache-aware code reordering. The second heuristic we developed addresses this issue. Figure 5(b) shows the configure-reorder-configure heuristic. Instead of applying code reordering first,

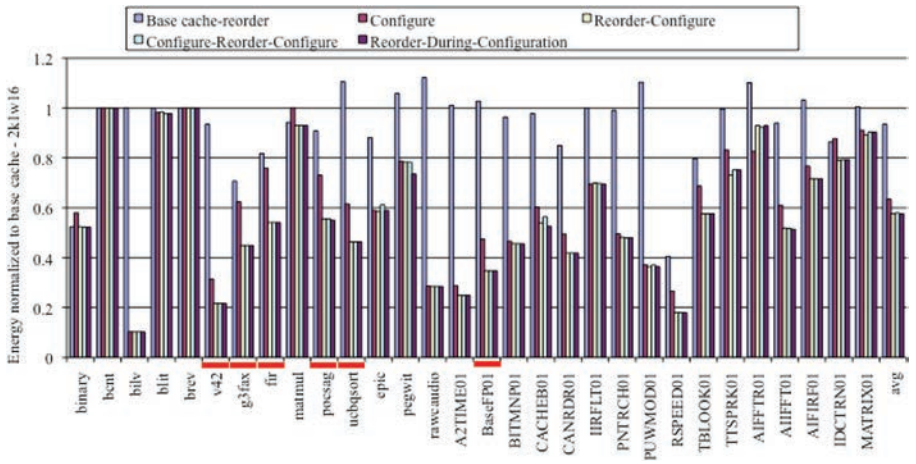

Fig. 6. Results for each benchmark showing energy consumption normalized to the base cache configuration. Underlined benchmarks highlight those that benefit exceptionally well from cache configuration combined with code reordering.

the configure-reorder-configure heuristic configures the cache given the original instruction stream. This cache configuration is then used as input to the code reordering phase, and the code is reordered for the best cache. However, after the code reordering phase, the cache may be in need of further tuning to the new optimized instruction stream. To account for this, the configure-reorder-configure heuristic applies another round of cache configuration to tune the cache to the optimized instruction stream. With one round of code reordering and two rounds of cache configuration, the configure-reorder-configure heuristic takes nearly twice as long to perform as the reorder-configure heuristic.

4.2.3. Reorder-During-Configuration. The final heuristic we develop represents the near-optimal search where code reordering is performed during the cache configuration loop. This method does not perform an exhaustive search of all possible code reordering situations for all possible cache configurations but searches only the most interesting cases. Figure 5(c) shows the reorder-during-configuration heuristic. This heuristic incorporates code reordering into the cache configuration loop by performing code reordering for each cache configuration before the configuration is simulated. Since the reorder-during-configuration heuristic performs code reordering for each cache configuration, this heuristic may take as much as seven times longer to simulate than the reorder-configure heuristic (seven cache configurations are explored and code reordering is applied to all seven configurations).

4.3. Experiments

4.3.1. Cache-Aware Code Reordering Verses Cache Configuration. Figure 6 shows the energy consumed by each benchmark for the code reordering method and the cache configuration method. The x-axis shows each benchmark studied and the y-axis represents the energy consumption of the instruction cache for each heuristic normalized to the energy consumption of the base cache configuration without any code reordering or cache configuration (shown as 1.0). The first bar shows the energy consumption of the base cache configuration with code reordering. The second bar shows the energy consumption of the best cache configuration with no code reordering applied.

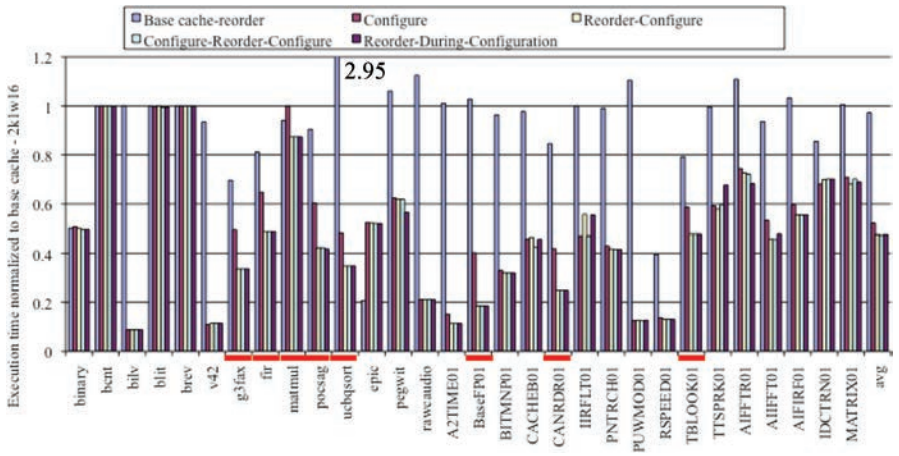

Fig. 7. Results for each benchmark showing execution time each normalized to the base cache configuration. Underlined benchmarks highlight those that benefit exceptionally well from cache configuration combined with code reordering.

The results show that code reordering yields weak improvements of only 6.5% on average. Furthermore, code reordering actually worsens energy in some examples, as much as by 12% in *rawcaudio*. Previous work [Muth et al. 2001] also observed such worsening. Of course, one could eliminate code reordering for worsened benchmarks, but that would improve the average only by a few percent. On the other hand, for some benchmarks code reordering obtains good energy savings, for instance, up to 60% savings for *RSPEED01*.

Cache configuration, on the other hand, yields energy savings of 36.5% on average. This number closes matches results obtained by Zhang [2004b], even though their base cache was a high-performance configuration rather than a small size configuration. The results show that cache configuration yields energy savings as much as 90% for *bliv*. Cache configuration also yields nearly equal or better savings than code reordering for every benchmark. Stated another way, for applications for which code reordering obtained substantial energy savings, cache configuration could achieve nearly equal or better, savings. Thus, one may conclude that, among the two methods, the hardware approach of cache configuration is superior, assuming either method is possible. Figure 7 shows similar data comparing benchmark performance results, yielding similar conclusions.

4.3.2. Combined-Methods Heuristics. Although cache configuration is clearly superior to code reordering alone, combining the two methods may still yield improvement for some benchmarks. We therefore compared the energy obtained by each of three heuristics described in Section 4.2 on the 30 benchmarks, with the results shown in Figure 6. The third, fourth, and fifth bars show the energy resulting from the reorder-configure, configure-reorder-configure, and reorder-during-configuration heuristics, respectively.

The results show that the three combined-method heuristics perform about the same on average. More significantly, the results show that combining code reordering and cache configuration only slightly improves the average energy compared to cache configuration alone, by about 6%. Thus, if a designer is concerned with reducing average energy savings across a variety of benchmarks, the hardware solution of a configurable cache may be sufficient.

The results also show that, for particular benchmarks, the combined-method heuristics do obtain a respectable additional 10% to 15% energy savings compared to cache

configuration alone. Thus, combining the methods may still be useful for applications with tight energy constraints. The performance data in Figure 7 shows similar results, yielding similar conclusions.

With regards to comparing the three heuristics, we see they perform about the same on average and that all three heuristics perform equally well for nearly every benchmark. These results were rather surprising to us, as we expected the "optimal" search performed by the reorder-during-configuration heuristic to achieve better results. Instead, the results show that the far more computationally efficient reorder-configure heuristic obtains near-optimal results.

We explain these results by first pointing out that, in most cases, the majority of energy and performance benefits come from cache configuration alone. Even if code reordering is applied to a suboptimal configuration, cache configuration will still determine the lowest energy cache for the instruction stream. Second, for both the reorder-configure and the configure-reorder-configure heuristics, code reordering is applied to wisely chosen cache configurations. For the reorder-configure method, code reordering is applied to the base cache configuration, which is the smallest available cache configuration. Research shows that code reordering is most effective at small cache sizes [McFarling 1989]. For the configure-reorder-configure method, code reordering is applied to the cache determined by applying cache configuration to the original instruction stream, again, a wise cache for which to perform reordering because that cache closely reflects the needs of the application. The goal of code reordering is twofold: to increase the spatial locality of code by placing hot paths through the application in contiguous memory, thus increasing the benefits of a larger line size, and to reduce conflict misses through cache line coloring. We manually observed that in the cases where code reordering was beneficial, the only change from the configured cache without code reordering to the configured cache with code reordering was a larger line size.

4.3.3. Non-Cache-Aware Code Reordering Verses Cache-Aware Code Reordering. In order to highlight the benefits gained by cache-aware code reordering, we compare the results presented in Sections 3.2 and 4.3.2. We note that the instruction set used for the results in Section 3.2 was the x86 instruction set, whereas the instruction set used in Section 4.3.2 was the PISA instruction set. Thus, direct normalization between the results is not possible (as even the instruction counts for the base cache's differ due to the different instruction set architectures), however, savings magnitudes can be compared. In addition, due to different tool chains and thus different benchmark sets, we can only compare benchmarks that successfully completed both tool chains.

Figure 8 compares (a) energy savings and (b) performance impacts for non-cache-aware code reordering with cache configuration and cache-aware code reordering with cache configuration for the best average heuristic as determined in Section 4.3.2. Results reveal the importance of considering the actual cache configuration while performing code reordering. Non-cache-aware code reordering with cache configuration reveals a 17% reduction in average energy consumption and a 19% reduction in average execution time. Cache-aware code reordering with cache configuration reveals a 47% reduction in average energy consumption and a 56% reduction in average execution time. Cache-aware code reordering with cache configuration increases the energy savings and reduces the execution time 2.8x and 2.9x, respectively. However, when looking at individual benchmarks, results reveal that cache-aware code reordering increases energy consumption and execution time as compared to non-cache-aware code reordering with cache configuration for six benchmarks and five benchmarks, respectively, out of eighteen benchmarks. Out of these benchmarks, only two benchmarks, TTPPRK01 and IIRFLT01, show a large increase in energy consumption and execution time (all other benchmarks show a negligible difference).

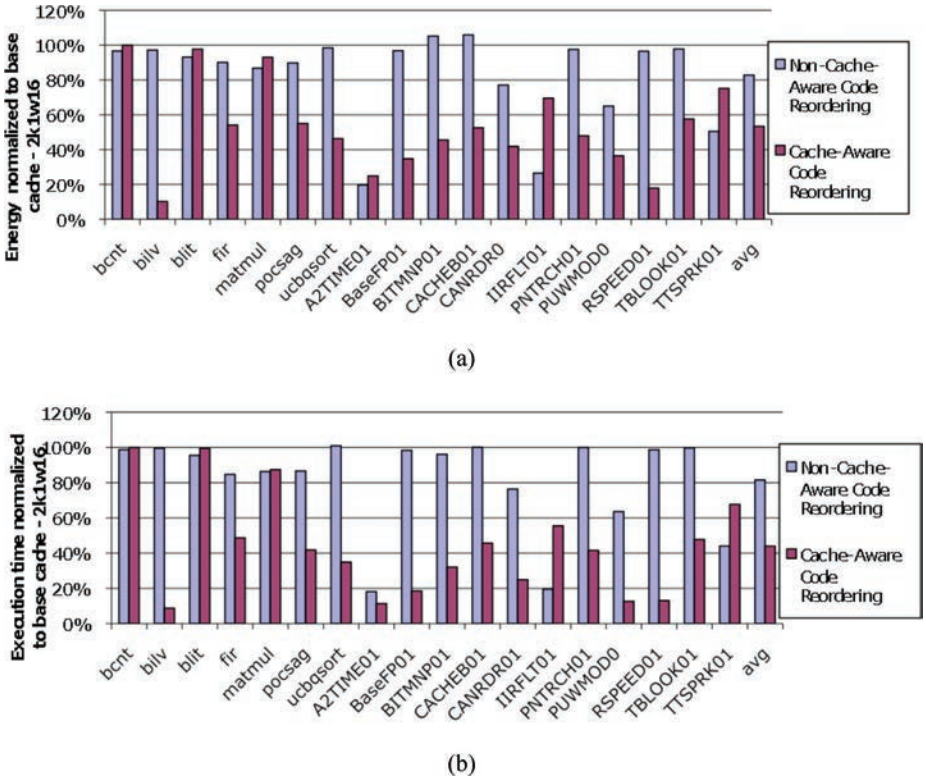

Fig. 8. Comparing non-cache-aware code reordering and cache-aware code reordering: (a) Energy consumption and (b) execution time normalized to the base cache configuration without code reordering.

We attribute these differences to the general uncertainty of code reordering benefits with respect to application particulars, however, overall, cache-aware code reordering with cache configuration help to alleviate most of this uncertainty.

4.3.4. Exploration Speedup. The reorder-configure heuristic we present provides near-optimal results and provides significant design space exploration speedup over an exhaustive method. Since code reordering is typically a link-time optimization, a designer applying both code reordering and cache configuration would likely perform cache configuration exploration in a simulation-based environment. In a simulation environment, the application is executed for each cache configuration to gather cache statistics. Simulation of complex systems can be quite time-consuming, easily requiring many hours or perhaps days to run just a single cache configuration, making development of an efficient heuristic essential to reducing design space exploration time.

The exhaustive reorder-during-configuration approach explores eighteen possible cache configurations, applying code reordering to each of the eighteen configurations before simulation. The reorder-configure heuristic explores only seven cache configurations and only performs code reordering once, reducing the number of cache configurations explored by 62% and reducing the number of code reorderings by 95%.

5. CONCLUSIONS AND FUTURE WORK

We provided a detailed study of the interplay between two optimization methods for instruction cache, namely, the software method of code reordering, and the hardware

method of a configurable cache. We found that the configurable cache method yields much better energy savings and performance on average and is always better or nearly equal to code reordering on every benchmark examined. We did observe particular applications where combining the two methods yielded additional savings—as much as 15%. We found that a simple heuristic that applies reordering once, followed by cache configuration, yielded near-optimal results. Furthermore, we show the importance of cache-aware code reordering. Cache-aware code reordering with cache configuration reveals an addition 2.8x and 2.9x increase in energy savings and reduction in execution time, respectively, as compared to non-cache-aware code reordering with cache configuration.

We plan to further investigate the interplay of the two methods for two-level cache hierarchies, which provide an exploration space orders of magnitude greater than a single level of cache (with tens of thousands of possible cache configurations rather than dozens). In addition, these results indicate that examining the interplay of previously introduced stand-alone optimization methods is an important step towards developing an effective system-level hardware/software design framework for embedded systems, thus we are currently exploring the interplay of code compression and cache configuration. Finally, since the magnitude of energy and performance benefits varies for different benchmarks, we plan to further study these benchmarks and classify characteristics that suggest the level of additional savings possible.

ACKNOWLEDGMENTS

We would like to thank Professor Saumya Debray and Patrick Moseley from the University of Arizona for providing PLTO and the trap profiler.

REFERENCES

ALBONESI, D. H. 2002. Selective cache ways: on demand cache resource allocation. *J. Instruction Level Parallel*.

ALTERA. 2010. Nios embedded processor system development. http://www.altera.com/corporate/news_room/releases/products/nr-nios_delivers_goods.html.

ARC INTERNATIONAL 2010. www.arccores.com.

ARM. 2010. www.arm.com.

AYDIN, H. AND KAELI, D. 2000. Using cache line coloring to perform aggressive procedure inlining. *ACM SIGARCH News 28*, 1, 62–71.

BAHAR, I. CALDER, B., AND GRUNWALD, D. A. 1998. Comparison of software code reordering and victim buffers. In *Proceedings of the 3rd Workshop of Interaction Between Compilers and Computer Architecture*.

BALASUBRAMONIAN, R., ALBONESI, D., BUYUKTOSUNOGLU, A., AND DWARKADAS, S. 2000. Memory heirarchy reconfiguration for energy and performance in general-purpose processor architecture. In *Proceedings of the 33rd International Symposium on Microarchitecture*.

BARTOLINI, S. AND PRETE, C. A. 2005. Optimizing instruction cache performance of embedded systems. *ACM Trans. Embedd. Comput. Syst. 4*, 4, 934–965.

BENINI, L., MACII, A., MACII, E., AND PONCINO, M. 1999. Selective instruction compression for memory energy reduction in embedded systems. In *Proceedings of the International Symposium on Low Power Emedded Systems*.

BURGER, D., AUSTIN, T., AND BENNET, S. 2000. Evaluating future microprocessors: The simplescalar toolset. Tech. rep. CS-TR-1308. Computer Science Department, University of Wisconsin-Madison.

CALDER, B. AND GRUNWALD, D. 1994. Reducing branch costs via branch alignment. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*.

CHEN, J. AND LEUPEN, B. 1997. Improving instruction locality with just-in-time code layout. In *Proceedings of the USENIX Windows NT Workshop*.

CHEN, Y. AND ZHANG, F. 2007. Code reordering on limited branch offset. *ACM Trans. Architec. Code Optimz. 4*, 2.

COHN, R., GOODWIN, P., LOWNEY, G., AND RUBIN, N. 1997. Spike: An optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows NT Workshop*.

COHN. R. AND LOWNEY, P. G. 2000. Design and analysis of profile-based optimization in Compaq's compilation tools for Alpha. *J. Instruction Level Parallelism 2*.

DINERO I. 2010. http://www.cs.wisc.edu/~markhill/DineroIV/.

EEMBC. 2010. The Embedded Microprocessor Benchmark Consortium. www.eembc.org.

GHOSH, A. AND GIVARGIS, T. 2003. Cache optimization for embedded processor cores: an analytical approach. In *Proceedings of the International Conference on Computer Aided Design*.

GIVARGIS, T. AND VAHID, F. 2002. Platune: a tuning framework for system-on-a-chip platforms. *IEEE Trans. Comput. Aid. Design*.

GLOY, N., BLACKWELL, T., SMITH, M. D., AND CALDER, B. 1997. Procedure placement using temporal ordering information. In *Proceedings of the 30th Anual ACM/IEEE International Symposium on Microarchitecture*. 303–313.

GORDON-ROSS, A., COTTERELL, AND VAHID, F. 2002. Exploiting fixed programs in embedded systems: A Loop cache example. *Comput. Architec. Letters* 1.

GORDON-ROSS, A., LAU, J., AND CALDER, B. 2008. Phase-based cache reconfiguration for a highly-configurable two-level cache hierarchy. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI (GLSVLSI)*.

GORDON-ROSS, A. AND VAHID, F. 2002. Dynamic loop caching meets preloaded loop caching—a hybrid approach. In *Proceedings of the International Conference on Computer Design*.

GORDON-ROSS, A., VAHID, F., AND DUTT, N. 2009. Fast Configurable-Cache Tuning with a Unified Second-Level Cache. *IEEE Trans. VLSI*.

HASHEMI, A., KAELI, D., AND CALDER, B. 1997. Efficient procedure mapping using cache line coloring. In *Proceedings of the International Conference on Programming Language Design and Implementation*.

HINES, S., WHALLEY, D., AND TYSON, G. 2007. Guaranteeing hits to improve the efficiency of a small instruction cache. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*.

HUANG, X., BLACKBURN, S., GROVE, D., AND MCKINLEY, K. 2006a. Fast and efficient partial code reordering: taking advantage of a dynamic recompiler. In *Proceedings of the International Symposium on Memory Managment*.

HUANG, X., LEWIS, T., AND MCKINLEY, K. 2006b. Dyanmic code management: improving whole program code locality in managed runtimes. In *Proceedings of the ACM International Conference on Virtual Execution Environments*.

HWU, W. W. AND CHANG, P. 1989. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual Intl. Symposium on Computer Architecture*.

KALMATIANOS, J. AND KAELI, D. 1999. Code reordering for multi-level cache hierarchies. Northeeastern University Computer Architecture Research Group. http://www.ece.neu.edu/info/architecture/publications.html.

KALMATIANOS AND J., KAELI, D. 2000. Accurate simulation and evaluation of code reordering. In *Proceedings of the IEEE International Symposium on the Performance Analysis of Systems and Software*.

KIN, J., GUPTA, M., AND MANGIONE-SMITH, W. The filter cache: an energy efficient memory structure. In *Proceedings of the IEEE Micro*.

LEE, D., BAER, J., BERSHAD, B., AND ANDERSON, T. 1999a. Reducing startup latency in web and desktop applications. In *Proceedings of the Windows NT Symposium*.

LEE, L. H., MOYER, W., AND ARENDS, J. 1999b. Low cost Embedded Program Loop Caching – Revisited. Tech. rep. N CSE-TR-411-99, University of Michigan.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*.

MALIK, A., MOYER, W., AND CERMAK, D. 2000. A low power unified cache architecture providing power and performance flexibility. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

MCFARLING. S. 1989. Program optimization for instruction caches. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*.

MIPS TECHNOLOGIES. 2010. www.mips.com.

MOSELEY, P., DEBRAY, S., AND ANDREWS, G. Checking program profiles. In *Proceedings of the 3rd IEEE International Workshop of Source Code Analysis and Manipulation*.

MUTH, R., DEBRAY, S., WATTERSON, S., AND DE BOSSCHERE, K. 2001. Alto: a link-time optimizer for the Compaq Alpha. *Softw. Pract. Exper. 31*, 6, 67–101.

PALESI, M. AND GIVARGIS, T. 2002. Multi-objective design space exploration using genetic algorithms. In *Proceedings of the International Workshop on Hardware/Software Codesign*.

PETTIS, K. AND HANSEN, R. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

RAMIREZ, A. 2005. Code placement for improving dynamic branch prediction accuracy. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

RAMIREZ, A., LARRIBA-PAY, J. NAVARRO, C., VALERO, M., AND TORRELLAS, J. 2002. Software trace caches for commerial applications. *Int. J. Parallel Program. 30*, 5.

RAMIREZ, A., LARRIBA-PEY, J., AND VALERO, M. 2000. The effect of code reordering on branch predition. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT).*

RAMIREZ, A., LARRIBA-PEY, J., AND VALERO, M. 2001. Instruction fetch architectures and code layout optimizations. *Proc. IEEE 89*, 11.

RAMIREZ, A., LARRIBA-PAY, J., AND VALERO, M. 2005. Software trace caches. *IEEE Trans. Comput. 54*, 1.

REINMAN, G. AND JOUPPI, N. P. 1999. Cacti2.0: An integraded cache timing and power model. Tech rep., COMPAQ Western Research Lab.

SAMPLES, A. D., AND HILFINGER, P. N. 1988. Code reorganization for instruction caches. Techn. rep. UCB/CSD 88/447, University of California, Berkeley.

SANGHAI, K., KAELI, D., RAIKMAN, A., AND BUTLER, K. 2007. A code layout framework for embedded processors with configurable memory hierarchy. In *Proceedings of the Workshop on Optimizations for DSP and Embedded Systems (ODES)*.

SCALES, D. 1998. Efficient dynamic procedure placement. Tech. rep. WRL-98/5, Compaq WRL Research Lab.

SCHARZ, B., DEBRAY, S., ANDREWS, G., AND LEGENDRE, M. 2001. PLTO: a link-time optimizer for the Intel IA-32 architecture. In *Proceedings of the Workshop on Binary Translation (WBT)*.

SCHMIDT, W. J., ROEDIGER, R. R., MESTAD, C. S., MENDELSON, B., SHAVIT-LOTTEM, I., AND BORTNIKOV-AND SITNITSKY, V. 1998. Profile-directed restructuring of operation system code. *IBM Syst. J. 37*, 2.

SRIVASTAVA, A., AND WALL, D. W. 1992. A practical system of intermodule code optimization at link-time. *J. Program. Lang. 11*, 1, 1–18.

SU, C. AND DESPAIN, A. M. 1995. Cache design trade-offs for power and performance optimization: a case study. *Proceedings of the International Symposium on Low Power Electronics and Design*.

TENSILICA. 2010. Xtensa processor generator. http://www.tensilica.com/.

VILLARREAL, J., LYSECKY, R., COTTERELL, S., AND VAHID, F. 2001. Loop analysis of embedded applications. *Tech. rep.* UCR-CSR-01-03, University of California Riverside.

ZHANG, C. AND VAHID, F. 2003. Cache configuration exploration on prototyping platforms. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP- 03)*.

ZHANG, C., VAHID, F., AND NAJJAR, W. 2003. A highly-configurable cache architecture for embedded eystems. In *Proceedings of the 30th Annual International Symposium on Computer Architecture.*

ZHANG, C. AND VAHID, F. 2004a. Using a victim buffer in an application-specific memory hierarchy. In *Proceedings of the Design, Automation and Test (DATE) Conference in Europe*.

ZHANG, C. AND VAHID, F. 2004b. A self-tuning cache architecture for embedded systems. In *Proceedings of the Design, Automation and Test (DATE) Conference in Europe.*

Received September 2009; revised February 2010; accepted June 2010