A First Look at the Interplay of Code Reordering and Configurable Caches

Ann Gordon-Ross, Frank Vahid*
Department of Computer Science and Engineering
University of California, Riverside
{ann/vahid}@cs.ucr.edu
http://www.cs.ucr.edu/~vahid
*Also with the Center for Embedded Computer Systems at UC Irvine

Nikil Dutt
Center for Embedded Computer Systems
School of Information and Computer Science
University of California, Irvine
dutt@cecs.uci.edu
http://www.ics.uci.edu/~dutt

ABSTRACT

The instruction cache is a popular target for optimizations of microprocessor-based systems because of the cache's high impact on system performance and power, and because of the cache's predictable temporal and spatial locality. Optimization techniques can be designed based on this predictability. We explore for the first time the interplay of two popular instruction cache optimization techniques: the long-known technique of code reordering and the relatively-new technique of cache configuration. We address the question of whether those two optimizations complement each other or if one optimization dominates the other. Through experiments using embedded system benchmarks, we show that cache configuration dominates a particular category of code reordering techniques with respect to optimizing performance and energy, obviating the need for reordering. We also examine the modern scenario of synthesized custom caches, and show that combining cache configuration with code reordering results in cache size reductions of 13% on average, and up to 89% in some benchmarks, beyond just cache configuration alone.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures: Design Styles – *cache memories*.

General Terms

Design.

Keywords

Configurable cache, code reordering, code reorganization, code layout, cache hierarchy, cache exploration, cache optimization, low power, low energy, architecture tuning.

1. INTRODUCTION AND MOTIVATION

Optimization is an important part of the design of an application or system. Many methods exist for optimizing performance, energy consumption, power consumption, area, etc. The instruction cache in a microprocessor-based system is a popular target for optimizations because the cache plays a major role in the microprocessor's performance and power consumption, and because the instruction cache is amenable to optimizations due to having predictable temporal and spatial locality.

Two popular approaches exist for tuning the instruction cache: code reordering and cache configuration. Code reordering/reorganization/layout at the basic block level is a mature method developed in the late 1980's to tune an instruction stream to the instruction cache to improve cache hit rates and improve the cache's utilization. This widely-researched method increases performance on average, though for some examples may decrease performance if applied. Due to new hardware technologies and core-based design methodologies, much recent research focuses on a second approach for instruction cache tuning: cache configuration. Instruction cache configuration tunes a cache to an application's instruction stream to find the lowest energy or best performing cache configuration for an application. Configurable parameters normally include cache size, line size, and associativity. Caches may be configured in a core-based methodology in which a designer synthesizes a customized cache along with a microprocessor [2][3][4][18][31]. Caches in pre-designed chips may instead be hardware configurable, with configuration occurring by setting register bits during system reset or during runtime [1][16][34].

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 tuning can be applied at design time or easily applied during runtime. For code reordering, the designer must compile the code, profile the code, and then generated an optimized executable by either recompiling the code or using a link-time code optimizer. Dynamic procedure placement [25] 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 tuning during runtime to eliminate the need for designer intervention [34] 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 data set.

Code reordering tunes the instruction stream for a cache whereas cache configuration tunes the cache to the resulting instruction stream. However, the interaction between the two tuning approaches – i.e., whether they complement, degrade, or obviate the need for each other – has not been considered before, and needs to be addressed. We present in this paper a study that investigates the interaction so that system designers can appropriately integrate these two approaches. Section 2 describes background work on code reordering and cache configuration. Section 3 describes our evaluation framework. Section 4 describes our experiments and results examining the interplay of the two techniques. Section 5 provides our conclusions.

GLSVLSI'05, April 17–19, 2005, Chicago, Illinois, USA.

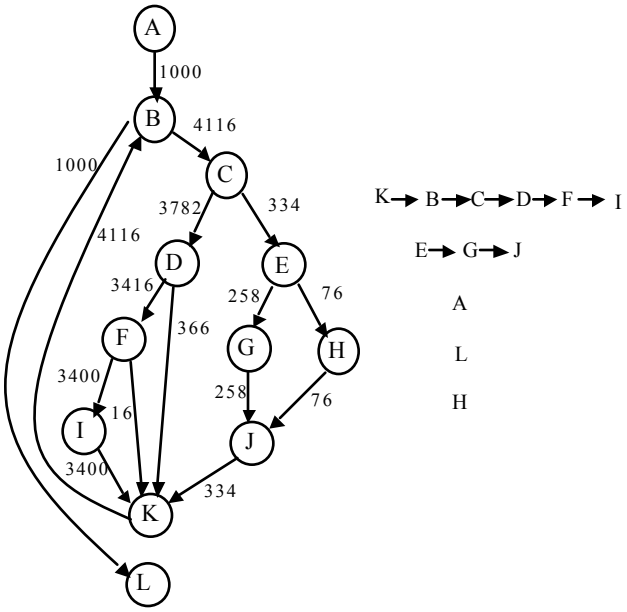

Figure 1: Edge weighted control flow graph and resulting basic block chains.

2. INSTRUCTION CACHE OPTIMIZATIONS

2.1 Code Reordering

Much research exists exploring the benefits of code reordering. Code reorganization at the basic block level dates back to early work in 1988 by Samples et. al [24]. McFarling [17] uses basic block execution counts for code ordering and instruction exclusion to maximize instruction cache hit ratios and significantly improves the effectiveness of the cache. Hwu et. al [13] and Pettis and Hansen [21] (PH) use similar techniques to reorder basic blocks using edge profile information. The PH code reordering methodology serves as the basis for many of the modern code reordering tools and techniques and in many cases is applied directly with no modification [6][7][11][14][20][23][26][27][28][29].

For the experiments presented in this paper, we use the Pentium Link-Time Optimizer (PLTO) [27], which performs code reordering using an improved PH algorithm. In the PH 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 bodies frequently contain an error condition that is checked in each loop iteration but 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 where 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 together in the final code layout.

The basic block reordering methodology used in this paper is based on the bottom-up positioning algorithm as described by PH. 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 PH bottom-up algorithm. PLTO improves upon the original PH algorithm in two ways. The first improvement addresses minor modifications needed to address problems identified by Calder et. al [5]. 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 PH bottom-up algorithm. The chains from each set are then concatenated to form the final layout.

The PH bottom-up algorithm is designed to exploit a single level direct mapped cache. Basic block reordering is performed without any attention 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 temporal aware code placement and multiple levels of cache [10]. The effects of code reordering and cache configuration with these considerations are left for future work.

2.2 Configurable Cache Tuning

Su et. al [30] show in early work that the memory hierarchy is very important in determining the power and performance of an application. Recently, Zhang et. al [34] 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

Recently, much research has focused on cache tuning. Motorola's M*CORE processor [16] offers way configuration, which allows the ways of a unified cache to be specified as data, instruction, or unified way or the way may be shutdown altogether. Albonesi [1] presented a method for using way shutdown to reduce dynamic energy consumption by an average of 40%. Zhang et. al [33] introduced a methodology called way concatenation where a cache may be configured as direct-mapped, 2-way, or 4-way set associative. Zhang shows an average energy savings of 40% compared to a 4-way set associative cache. In additional work by Zhang [32], a tuning heuristic is presented to tune the level one cache, producing a set of Pareto optimal points trading off energy consumption and performance. In [12], Gordon-Ross et. al extend Zhang's work to include a second level of cache. A cache tuning heuristic is designed to tune separate level one and level two caches showing average cache energy savings of 53% over a base cache configuration

In this paper, we will apply the cache tuning heuristic developed by Zhang et. al [34] to configure the level one cache for reduced energy consumption. Zhang describes a runtime tuning heuristic to efficiently and effectively search the level one cache configuration for optimal size, line size, and associativity showing average cache energy savings of 45% to 55%. The heuristic searches parameters based on their impact on the energy consumption of the cache. Zhang determined cache size to have the greatest impact on the energy consumption of the cache, followed by line size and finally associativity. The heuristic also takes care to explore each parameter in a way that minimizes the need for any cache flushing. The cache tuning heuristic explores the cache as follows:

1. Begin with the smallest cache size, line size, and associativity. Increase the cache size (keeping the line size and associativity constant) while there is a decrease in energy. If an increase in energy occurs or the cache is at the maximum size, the cache size is fixed at the last size explored.
2. Using the cache size determined in step one, increase the line size while there is a decrease in energy. If an increase in energy occurs or the maximum line size is reached, the cache line size is fixed at the last size explored.

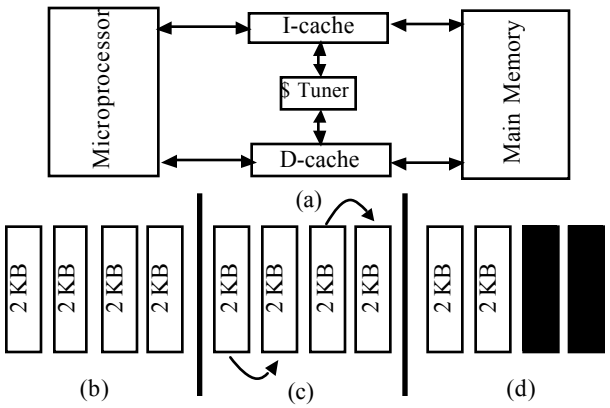

Figure 2: (a) Configurable cache architecture; (b) 8 KByte 4-way base cache with for 2 KByte sub banks; (c) 8 KByte 2-way cache using way concatenation; (d) 4 KByte 2-way cache using way shutdown.

3. Using the cache size determined in step one and the line size determined in step two, increase the associativity while there is a decrease in energy. If an increase in energy occurs or the maximum associativity is reached, the associativity is fixed at the last associativity explored.

This configuration heuristic may be implemented as software running on a co-processor or as specialized hardware.

To enable the cache tuning heuristic to tune during runtime, a tunable cache is necessary. Zhang [33] describes tunable cache hardware and presents 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 consisting of four separate banks that may be turned on, concatenated with another bank (Figure 2(c)), or turned off (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 8 KB is desired, four banks of 2 KB each will be utilized. An 8 KB direct-mapped, 2-way, and 4-way set associative cache is available. To reduce the cache size to 2 KB, three of the banks must be shut down leaving one remaining bank of 2 KB. Since banks are used to increase associativity and there is only one bank available in a 2 KB cache, only a direct-mapped cache is available for 2 KB. Further details are available in [33].

3. EVALUATION FRAMEWORK

To determine the combined effects of code reordering and cache configuration, we used 27 benchmarks: 12 benchmarks from the Powerstone benchmark suite [16], 3 benchmarks from the MediaBench benchmark suite [15], and 11 benchmarks from the EEMBC benchmark suite [9]. 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.

We used PLTO [27] to perform code reordering on the applications. PLTO is similar to the popular ALTO [20] 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 described above 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 [19]. The trap-based profiler combines the trace cache simulator Dinero IV [8] 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 * (linesize/16) * 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 [22] 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 examined 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 require. 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 [12] 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 [17] 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 2 Kbyte 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.

4. EXPERIMENTS

We explore the interaction 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. We are also interested in applying code reordering again after cache configuration is performed, however, the code reordering method used does not take the cache configuration into account and consequently reapplying code reordering given the tuned cache configuration will not effect the code ordering. In the future, we hope to explore the iterative application of code reordering and cache tuning; however, setting up an experimental framework for this is non-trivial due to the lack of support for past tools developed for research on cache configuration aware code reordering. Creating the setup described in Section 3 took over two months to locate, obtain, and get all tools interacting properly. Nevertheless, the non-iterative interplay that we examine covers a wide range of potential applications of code reordering and cache configuration, especially those that apply reordering at compile time and cache configuration at runtime, since iteration would not be possible in such a case.

4.1 Code Reordering and Cache Configuration

Figure 3 shows the energy savings and 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 [20]. (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).

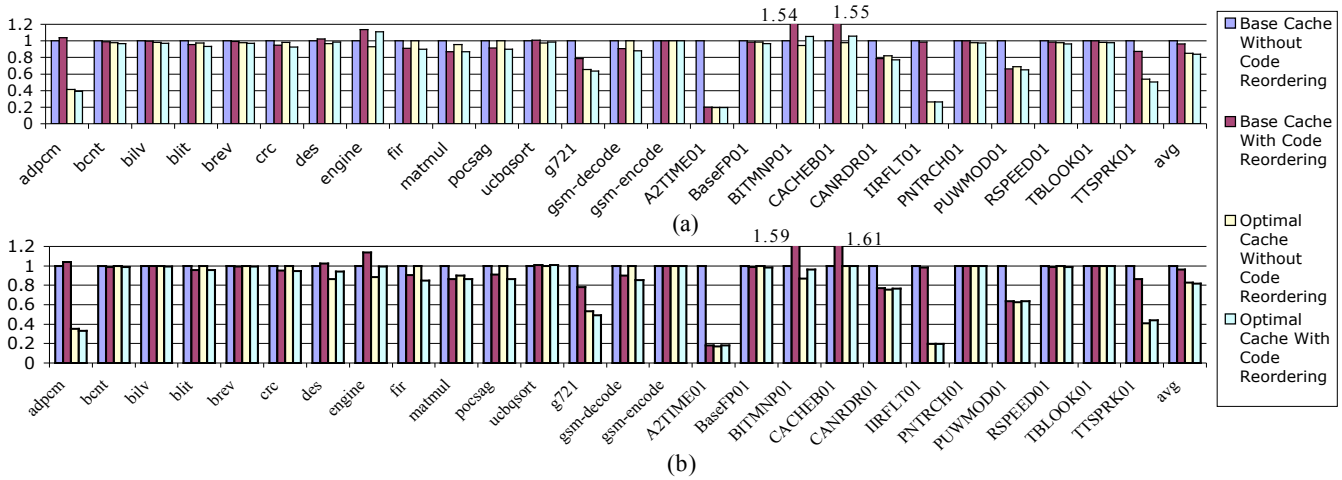

Figure 3: (a) Energy consumption with code reordering and cache configuration. Energy for each benchmark is normalized to the energy consumption of the base cache without code reordering. (b) Execution time with code reordering and cache configuration. Execution time for each benchmark is normalized to the execution time of the base cache without code reordering.

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

4.2 Change in Cache Requirements Due to Code Reordering

Table 1 shows the optimal cache configuration for each benchmark without code reordering and with code reordering. 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 1 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 1 shows the overall change in optimal instruction cache area due to code reordering. Positive change denotes an

Table 1: 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**	8.71%
bcnt*	2k1w64	2k1w64	0%
bilv*	2k1w32	**2k1w64**	14.28%
blit*	2k1w64	2k1w64	0%
brev*	2k1w64	2k1w64	0%
crc*	2k1w64	2k1w64	0%
des*	8k1w16	4k1w16	**-78.70%**
engine*	4k1w16	8k1w16	44.04%
fir*	2k1w16	**2k1w32**	8.72%
matmul*	4k1w16	2k1w16	**-89.08%**
pocsag*	2k1w16	**2k1w32**	8.72%
ucbqsort*	2k1w64	2k1w64	0%
g721**	8k1w16	**8k1w32**	7.53%
gsm-decode**	2k1w16	**2k1w64**	21.75%
gsm-encode**	2k1w16	2k1w16	0%
A2TIME***	4k1w16	**2k1w32**	**-72.60%**
BaseFP***	2k1w32	**2k1w64**	14.28%
BITMNP***	4k1w64	4k1w32	**-9.54%**
CACHEB***	2k1w64	2k1w64	0%
CANRDR***	4k1w64	2k1w32	**-67.96%**
IIRFLT***	8k1w64	8k1w64	0%
PNTRCH***	2k1w64	2k1w64	0%
PUWMOD***	4k1w64	2k1w32	**-67.96%**
RSPEED***	2k1w64	2k1w64	0%
TBLOOK***	2k1w64	2k1w64	0%
TTSPRK***	8k1w64	4k1w32	**-80.95%**
*Powerstone **Mediabench ***EEMBC		**avg**	**-13.03%**

increase in cache size while a negative change denotes a decrease in cache size. Overall, we observed a 13% 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 an average of 13% and by as much as 89%.

5. CONCLUSIONS AND FUTURE WORK

We explored for the first time the interplay of two instruction cache optimization techniques: code reordering and cache configuration. Our results show that cache configuration obviates the need for code reordering with respect to performance and energy, using a popular code reordering method. Thus, cache configuration applied dynamically during runtime would eliminate the need for designer applied code reordering. Even for cache configuration applied at design time, our results show that cache configuration performs better than code reordering, yielding better average improvement, and avoiding the undesirable energy and performance degradation obtained by code reordering for some benchmarks.

Additionally, we showed that in 52% of the benchmarks, code reordering was successful in improving cache utilization by resulting in an optimal cache configuration that was either smaller or had a larger line size (and better hit rate). The success of code reordering resulted in an average reduction in instruction cache size of 13% and as high as 90%. This reduction in size reveals a benefit for applying both optimization methods in core-based embedded microprocessor design flows.

Future work includes studying code reordering techniques that take temporal locality and multiple cache levels into account. Additionally, we plan to study the iterative interplay of code reordering and cache configuration using a code reordering technique that takes into consideration the target cache configuration. In such an approach, we might apply reordering and cache configuration multiple times and in different sequences in a search for the best tuning of instruction stream to cache and cache to instruction stream.

6. ACKNOWLEDGEMENTS

We would like to thank Professor Saumya Debray and Patrick Moseley from the University of Arizona for providing PLTO and the trap profiler. This research was supported in part by the National Science Foundation (CCR-0203829, CCR-9876006).

7. REFERENCES

[1] Albonesi, D.H. Selective cache ways: on demand cache resource allocation. Journal of Instruction Level Parallelism, May 2002.

[2] Altera, Nios Embedded Processor System Development, http://www.altera.com/corporate/news_room/releases/products/nr-nios_delivers_goods.html

[3] Arc International, www.arccores.com

[4] ARM, www.arm.com

[5] Calder, B, Grunwald, D. Reducing branch costs via branch alignment. 6th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1994.

[6] Cohn, R., Goodwin, P., Lowney, G., Rubin, N. Spike: an optimizer for Alpha/NT executables. in USENIX Windows NT Workshop, August 1997.

[7] Cohn. R., Lowney, P.G. Design and analysis of profile-based optimization in Compaq's compilation tools for Alpha. Journal of Instruction Level Parallelism, vol. 2, May 2000.

[8] Dinero IV, http://www.cs.wisc.edu/~markhill/DineroIV/

[9] EEMBC, the Embedded Microprocessor Benchmark Consortium, www.eembc.org.

[10] Gloy, N. Code Placement using Temporal Profile Information. PhD thesis, Harvard University, 1998.

[11] Gloy, N., Blackwell, T., Smith, M.D., Calder, B. Procedure placement using temporal ordering information. Proceedings of the 30th Anual ACM/IEEE Intl. Symposium on Microarchitecture, pages 303--313, Dec. 1997.

[12] Gordon-Ross, A., Vahid, F., Dutt, N. Automatic tuning of two-level caches to embedded applications. Design, Automation and Test Conference in Europe (DATE), 2004.

[13] Hwu, W.W., Chang, P. Achieving high instruction cache performance with an optimizing compiler. Proceedings of the 16th Annual Intl. Symposium on Computer Architecture, June 1989.

[14] Lee, D., Baer, J., Bershad, B., Anderson, T. Reducing startup latency in web and desktop applications. In Windows NT Symposium, July 1999

[15] Lee, C., Potkonjak, M., Mangione-Smith, W.H. MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. Proc 30th Annual International Symposium on Microarchitecture, December 1997

[16] Malik, A., Moyer, W., Cermak, D. A low power unified cache architecture providing power and performance flexibility. International Symposium on Low Power Electronics and Design, 2000.

[17] McFarling, S. Program optimization for instruction caches. ASPLOS-III: 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, April 1989.

[18] MIPS Technologies, www.mips.com

[19] Moseley, P., Debray, S., Andrews, G. Checking program profiles. Third IEEE International Workshop of Source Code Analysis and Manipulation, September 2003.

[20] Muth, R., Debray, S., Watterson, S., de Bosschere, K. Alto: a link-time optimizer for the Compaq Alpha. Software Practice and Experience, 31(6):67--101, Jan. 2001

[21] Pettis, K., Hansen, R. Profile guided code positioning. Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation, 1990.

[22] Reinman, G., Jouppi, N.P. Cacti2.0: an integraded cache timing and power model. COMPAQ Western Research Lab, 1999.

[23] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B., Chen, B. Instrumentation and optimization of Win32/Intel executables using ETCH. In USENIX Windows NT Workshop, August 1997,

[24] Samples, A.D., Hilfinger, P.N. Code reorganization for instruction caches. Technical Report UCB/CSD 88/447, University of California Berkly, October 1988.

[25] Scales, D.J. Efficient dynamic procedure placement. Technical Report WRL-98/5, Compaq WRL Research Lab, May 1998.

[26] Scales, D.J., Randall, K.H, Ghemawat, S., Dean, J. The swift java compiler: design and implementation. Technical Report 2000/2, Compaq Western Research Laboratory, Apr. 2000.

[27] Scharz, B., Debray, S., Andrews, G., Legendre, M. PLTO: a link-time optimizer for the Intel IA-32 architecture. Proc. 2001 Workshop on Binary Translation (WBT-2001), Sept. 2001.

[28] Silicon Graphics Inc, Cord manual page. IRIX 5.3

[29] Srivastava, A., Wall,D. A practical system for intermodule code optimization at link-time. Technical Report 92/6. Digital Western Rearch Labrartory. June 1992.

[30] Su, C., Despain, A.M. Cache design trade-offs for power and performance optimization: a case study," Int. Symp. on Low Power Electronics and Design , 1995.

[31] Tensilica, Xtensa Processor Generator, http://www.tensilica.com/.

[32] Zhang, C., Vahid, F. Cache configuration exploration on prototyping platforms. 14th IEEE International Workshop on Rapid System Prototyping (RSP- 03). San Diego, June 2003.

[33] Zhang, C., Vahid, F., Najjar, W. A highly-configurable cache architecture for embedded systems. 30th Annual International Symposium on Computer Architecture, June 2003.

[34] Zhang, C., Vahid, F. A self-tuning cache architecture for embedded systems. Design, Automation and Test Conference in Europe (DATE), 2004