Tuning of Loop Cache Architectures to Programs in Embedded System Design

Susan Cotterell and Frank Vahid*
Department of Computer Science and Engineering
University of California, Riverside
{susanc, vahid}@cs.ucr.edu
*Also with the Center for Embedded Computer Systems at UC Irvine

Abstract
Adding a small loop cache to a microprocessor has been shown to reduce average instruction fetch energy for various sets of embedded system applications. With the advent of core-based design, embedded system designers can now tune a loop cache architecture to best match a specific application. We developed an automated simulation environment to find the best loop cache architecture for a given application and technology. Using this environment, we show significant variation in the best architecture for different examples. The results support the need for future fast synthesis of tuned loop cache architectures.

Categories and Subject Descriptors
B.3.0 [**Memory Structures**]: General.

General Terms
Design.

Keywords
Low power, low energy, tuning, loop cache, embedded systems, instruction fetching, filter cache, customized architectures, memory hierarchy, synthesis, architecture tuning, cores.

1. Introduction
Reducing energy and power consumption of embedded systems translates to longer battery lives and reduced cooling requirements. For embedded microprocessor based systems, instruction fetching can contribute to a large percentage of system power (nearly 50% in [19]), since such fetching occurs on nearly every cycle, involves driving of long and possibly off-chip bus lines, and may involve reading numerous memories concurrently – such as in set-associative caches.

Several approaches to reducing instruction fetch energy have been proposed, including program compression to reduce the amount of bits fetched [4][16][20], bus encoding to reduce the number of switched wires [5][22][27][29], and efficient instruction cache design [2][14][17][28]. Another category of approaches, which capitalize on the common feature of embedded applications spending much time in small loops, integrate a tiny (perhaps 64 word) instruction cache with the microprocessor. Such tiny caches have extremely low power per access, perhaps 50 times less than regular instruction memory access [19].

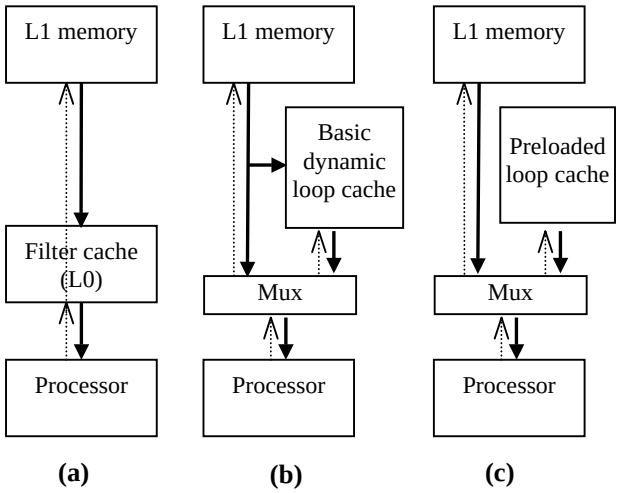
Several low-power tiny instruction cache architectures have been introduced in recent years, including the filter cache [15], dynamically-loaded tagless loop caches [18][19], and preloaded tagless loop caches [11]. Such tiny caches can be used in addition to an existing cache hierarchy. Not only can each type of cache vary in size, but also in certain features. A designer of a mass-produced microprocessor platform might select the cache architecture that performs best across a wide set of benchmarks.

However, an embedded system typically runs one fixed application for the system's lifetime. For example, a cell phone's software usually does not change. Furthermore, embedded system designers are increasingly utilizing microprocessor cores rather than off-the-shelf microprocessor chips. The combination of a fixed application and a flexible core opens the opportunity to tune the core's architecture to that fixed application. *Architecture tuning* is the customizing of an architecture to most efficiently execute a particular application (or set of applications) under given constraints on size, performance, power, energy, etc.[30], as discussed in the Y-chart methodology of [13]. A very aggressive form of tuning involves creating a customized instruction set [1][8][9][10], known as an application-specific instruction set.

Complementary to such application-specific instruction-set processor design is the design of customized memory architectures [7][12][23][24][25][26]. In this paper, we examine the need for customized design of the tiny instruction cache part of a memory architecture in order to minimize instruction fetch energy for a given program. We use an automated simulation environment to demonstrate the significant performance and energy variations for various tiny instruction cache architectures. We show that no one architecture is best across a particular set of benchmarks. For those benchmarks, tuning the cache architecture results in a 2% to 40% savings compared to the architecture that is best for the entire set of benchmarks. Variation would be even greater for more diverse benchmarks. The results illustrate the need for fast exploration and synthesis

ISSS'02, October 2–4, 2002, Kyoto, Japan.

Figure 1: Loop cache roles: (a) a filter cache is L0 memory – misses cause a fill from L1, (b) a basic dynamically loaded loop cache sits to the side – accesses are either to it or L1 memory, (c) a preloaded loop cache also sits to the side, but its contents do not change during program execution.



of tiny instruction cache hierarchies in embedded system design.

2. Filter/Loop Cache Architectures

Several tiny cache architectures, shown in Figure 1, have been proposed in recent years for low energy or power, each with several variations.

2.1 Filter Cache

The filter cache proposed in [15] is an unusually small direct-mapped cache. This filter cache is placed between the CPU and the L1 cache and utilizes standard tag comparison and miss logic. Because the filter cache is much smaller than the L1 cache, it will have a faster access time and lower power per access, due mainly to having shorter, lower capacitance wires. However, because the cache is so small, it may suffer from a high miss rate and hence may decrease overall performance. Profile-guided compilation was proposed in [3] to reduce misses.

Architecture variation for the filter cache involves different cache sizes. Larger filter caches may have a lower miss rate but will have higher power per access.

2.2 Dynamically Loaded Loop Caches

To eliminate performance degradation and the need for tag comparisons, a loop cache was proposed in [18]. The proposed loop cache is a small instruction buffer that is tightly integrated with the processor and that has no tag address store or valid bit. Instead of placing the loop cache between the processor and an L1 cache and risk degrading performance, the loop cache is simply an alternative location from which to fetch instructions. A loop cache controller is responsible for filling the loop cache when detecting a simple loop – defined as any short backwards branch instruction. At the end of the first iteration of a simple loop, the short backwards branch is detected. Then, during the second iteration, the loop cache is filled. Finally, starting with the third iteration, the loop cache controller fetches instructions from the loop cache instead of regular instruction memory.

The location from which to fetch an instruction is determined using a simple counter. The controller continues to fetch from the loop cache, resetting the counter each time it reaches zero (indicating the loop is iterating again). This behavior will continue until a control of flow change is encountered or until the triggering short backwards branch is not taken. We refer to this type of dynamically loaded loop cache as the *original dynamic* loop cache.

One drawback of the original dynamic loop cache is the cache's inability to handle loops that are larger than the cache itself. The original dynamic loop cache controller would only fill the loop cache if the loop completely fit within the cache. To alleviate the problem, the original dynamic cache was later extended in [19] to what is referred to as a *flexible dynamic* loop cache. In this design, if a loop is larger than the loop cache, the loop cache will be filled with the instructions located at the beginning of the loop until the loop cache is full.

Architecture variation for the dynamically loaded loop cache thus includes loop cache size as well as original versus flexible loop support.

2.3 Preloaded Loop Caches

In a dynamically loaded loop cache, internal branches within loops, multiple backwards branches to the same starting point in a loop, nested loops, and subroutines all pose problems. In each of the situations, the control of flow change would cause the filling of or execution from the loop cache to be aborted. A preloaded loop cache was proposed in [11] to overcome these limitations. Using profiling information gathered for a particular application, the loops that comprised the largest percentage of execution time are selected and preloaded into the loop cache during system reset, along with extra bits indicating whether control of flow changes within the loop cause an exit from the loop. In addition, loop address registers are preloaded to indicate which loops were preloaded into the loop cache. After this initialization, the contents of the loop cache do not change for the duration of program execution. By preloading, all of the above-mentioned situations with control of flow changes could be handled.

The loop cache controller can check for a loop address whenever a short backwards branch is executed. Since loops are preloaded, loop cache fetching can begin on a loop's second rather than third iteration. This approach is referred to as the *preloaded loop cache (sbb)*. Alternatively, loop addresses can be looked for on every instruction, allowing loop cache fetching to begin on a loop's first iteration. This approach is referred to as the *preloaded loop cache (sa)* (starting address).

While the preloaded loop cache is also a tagless loop caching scheme, it only allows a limited number of loops to be cached and requires more complex logic to index into the preloaded loop cache. Unlike the previously mentioned caches, a preloaded loop cache is not transparent to the designer or tool flow, requiring profiling and preloading, but with potentially greater energy savings.

Architecture variation for a preloaded loop cache includes cache size, number of supported loops, and loop address checking strategy.

3. Evaluation Framework

Which tiny instruction cache architecture and variation is best? The answer depends on the application being executed. We evaluated a number of cache architecture variations on a set of Powerstone benchmarks [21]. For each benchmark, we considered 106 different cache configurations:

- filter cache – cache sizes ranging from 8 to 1024 bytes, with lines sizes of 4 to 64 (configurations where lines sizes are greater than cache size were omitted)
- original dynamic loop cache – cache sizes ranging from 8 to 1024 entries
- flexible dynamic loop cache – cache sizes ranging from 8 to 1024 entries
- preloaded loop cache using short backwards branch address – cache sizes ranging from 8 to 1024 entries, with 2 to 6 loop address registers
- preloaded loop cache using start address – cache sizes ranging from 8 to 1024 entries, with either 2 or 3 loop address registers

For the dynamic and preloaded loop caches, each entry within the cache corresponds to a 32-bit instruction. In addition, for the preloaded loop caches, the number of loop address registers available indicates the maximum number of loops that can be preloaded into the loop cache.

We developed a suite of tools to evaluate each cache configuration for a given benchmark. Starting from C code for each benchmark, an lcc compiler ported to the MIPS instruction set is used to compile each benchmark. We then use a MIPS instruction-set simulator to obtain instruction traces for each benchmark. These instruction traces are then fed to the appropriate loop cache simulator that calculates the number of operations corresponding to the selected cache configuration. For example, in dynamically loaded loop caches, we are interested in the number of fetches from the loop cache, fills to the loop cache, and fetches from the instruction memory. Then, using this data, we calculate the instruction-fetch related power consumed by each configuration, the execution time for each benchmark, and the instruction-fetch related energy consumed.

The loop cache simulators used back-annotated power data obtained from synthesis. We calculated power and energy based on the switching activity of each operation and the relative capacitance of various components. The switching activity of each operation was measured by implementing the various cache designs in VHDL. Each design was synthesized using Synopsys Design Compiler and simulated at the gate-level to determine the average switching activity. The relative capacitance values (e.g., the capacitance of an internal net compared to a bus) associated with the different components of each design were factored out such that we could set these values to correspond to different technologies. Using this approach, we can compare the various cache configurations without limiting the results to a given technology. However, a designer interested in determining how each cache configuration performs for a specific technology can simply set the values to correspond with the technology of interest.

Table 1: Corresponding code for various cache configurations

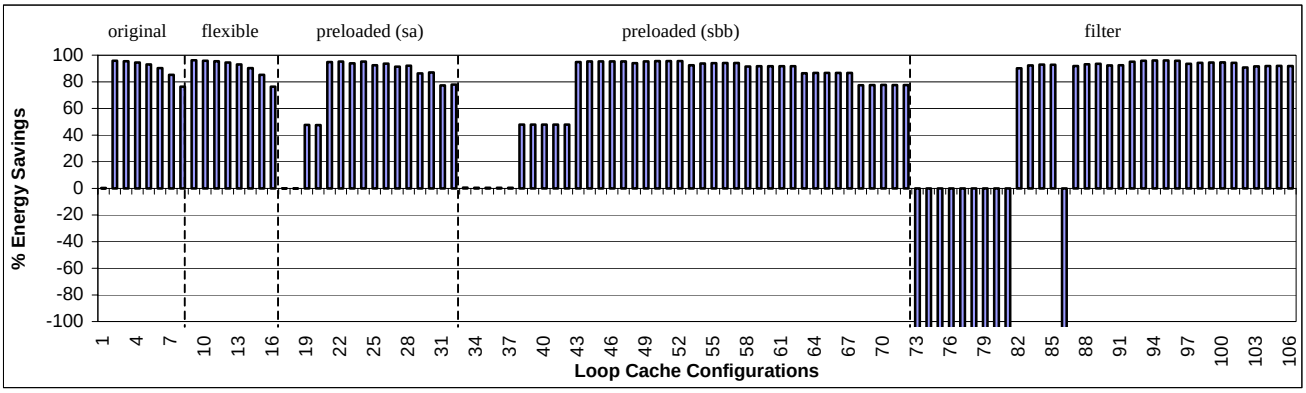
Cache Type	Size	Num Loops/ Line Size	Code
Original Dynamic	8-1024	N/A	1-8
Flexible Dynamic	8-1024	N/A	9-16
Preloaded Loop Cache (SA)	8-1024	2-3 loops	17-32
Preloaded Loop Cache (SBB)	8-1024	2-6 loops	33-72
Filter Cache	8-1024	8-64 line size	73-106

4. Results

To facilitate plotting of so many configurations, we map each configuration to a number, with Table 1 providing a key to show the mapping. For example, *1* represents the original dynamic cache with 8 entries, *2* represents the original dynamic cache with 16 entries, and so on. For the preloaded loop cache using start address, an 8 entry cache with 2 loop address registers is referred to with *17*, an 8 entry cache with 3 loop address register is referred to with *18*. For the filter cache, an 8 byte cache with line size of 8 is referred to with *73*. And, since an 8 byte cache cannot have a line size of 16, the next cache configuration (referred to with *74*) is a 16 byte cache with line size of 8.

We measured the average instruction access energy savings for each cache configuration for each benchmark compared to

Figure 2: Instruction fetch energy savings for *blit* for various cache configurations.



the instruction access energy consumed by a configuration that contains no filter/loop cache. Figure 2 shows the savings for each cache configuration for the *blit* benchmark (a graphics application) in Powerstone. For this benchmark, we see that the dynamic loop caches, preloaded loop caches, and most of the filter caches do well, achieving roughly 96% energy savings. Since the dynamic loop cache is transparent to the designer and has no performance overhead, a designer would likely choose a dynamic cache for this benchmark.

Figure 3 shows the savings for each cache configuration for the *v42* benchmark (a modem encoding/decoding application). Although the dynamic caches performed well for the *blit* benchmark, they are not competitive for *v42*. Instead, the preloaded loop caches yield a much larger energy savings compared to the dynamic and filter caches. The best preloaded loop cache using the short backwards branch address is a 512 entry cache with 5 loop address registers, resulting in instruction fetch energy savings of over 60%.

The remaining benchmarks showed similar variation with respect to which loop cache architecture was best. Each class of tiny instruction cache architecture was best for at least one benchmark.

Figure 4 shows the average savings of each cache configuration over all benchmarks. Cache configuration *30*, a pre-loaded loop cache using start address with 512 entries and 3 loop address registers, had an average savings of 73%. Cache configuration *105*, a filter cache of size 1024 bytes and line size of 32, did equally well with a savings of 73%. These two configurations had the highest average savings over all cache configurations. However, the filter cache does result in some performance degradation. Figure 5 shows the average increase in execution time given the various filter cache configurations compared with the execution time when using a loop cache.

If the memory architecture could not be customized to a particular application, as is the case for pre-fabricated microprocessors or even core's without support for customization, then the microprocessor designer would typically include the cache configuration that is best on the average over a set benchmarks. We thus compared the difference in energy savings of the best average configuration over all benchmarks to the best customized configuration for each benchmark, to see what additional savings we get through customization. We previously concluded that configurations *30* and *105* had the best average savings. The first bar in Figure 6 shows the savings for the best cache configuration for the given benchmark, the second bar shows the savings for configuration *30*, and the third bar shows the savings for configuration *105*. We see that the best customized cache configuration for *compress* and *jpeg* each have an increased savings of 23% over configuration 30. In addition, for *adpcm*, *ucbqsort*, and *v42*, the best customized cache configuration has an increased savings of 43%, 25%, and 45% respectively, over configuration *105*. Thus, although on average the difference was only 11% for both cache configurations, there exist certain benchmarks where using the

Figure 3: Instruction fetch energy savings for *v42* for various cache configurations

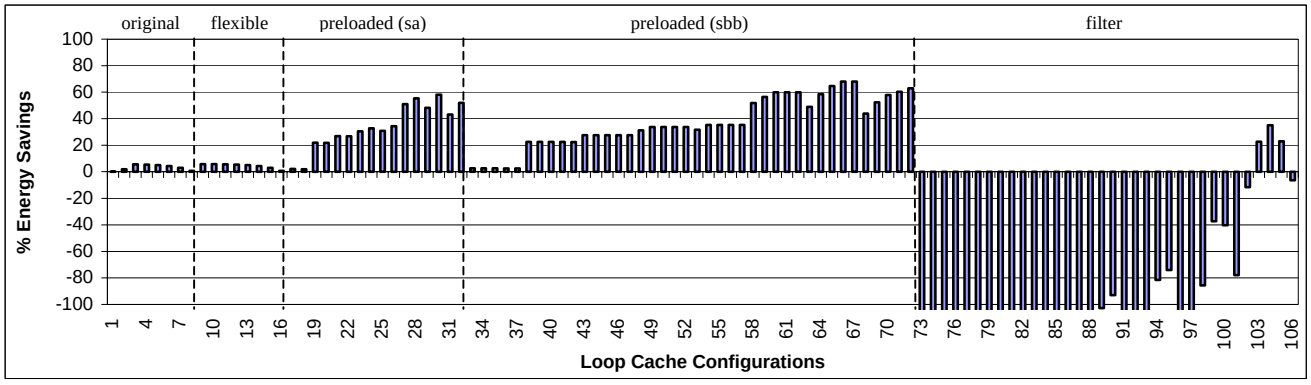


Figure 4: Average instruction fetch energy savings for Powerstone benchmarks for various cache configurations

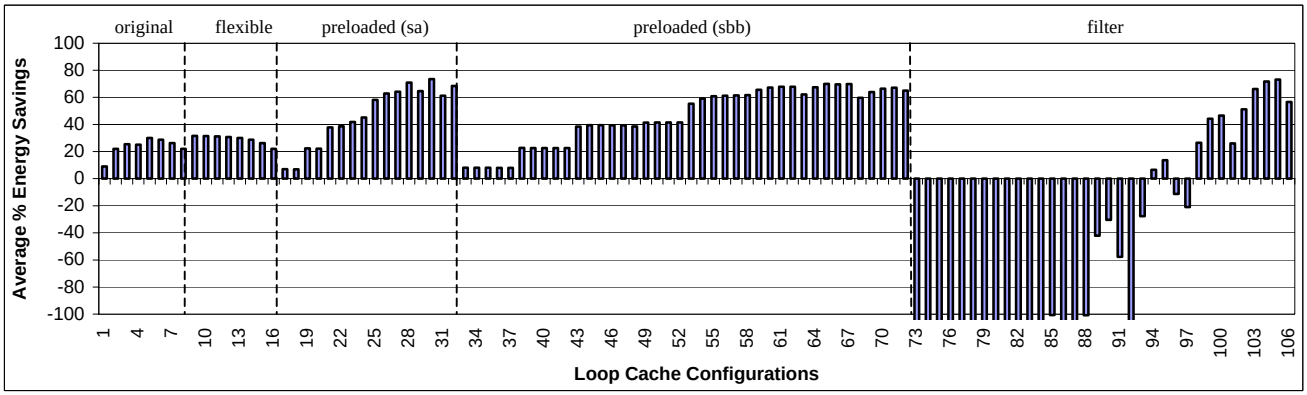
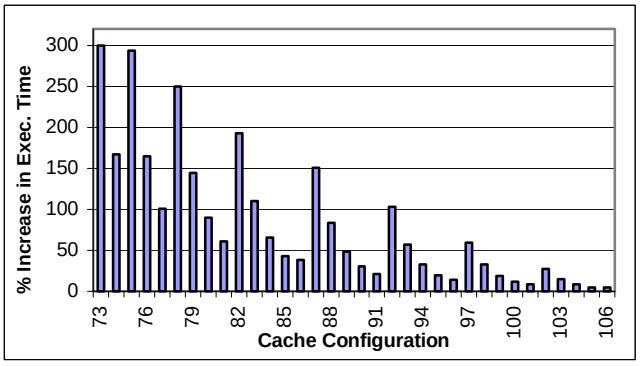


Figure 5: Average performance penalty for filter caches.



best average overall cache configuration will yield significantly less savings than using the best cache configuration for a given benchmark.

5. Conclusions

Incorporating a tiny instruction cache can result in instruction fetch energy savings, but many variations of such caches exists. By customizing such a cache to a particular program, we obtained an average additional energy savings of 11% compared to a non-customized cache, with savings over 40% in some cases examined.

Although our current environment is automated, the environment requires several hours to find the best configuration, because it reruns a cache simulator for every configuration. We have also developed faster exploration of the configurations [6]. Additionally, we plan to investigate a wider variety of loop cache architectures, such as warm-fill dynamically-loaded loop caches and hybrid dynamic/preloaded loop caches, as well as examining the impact that a good loop cache can have on the design of the first level of regular instruction cache.

6. Acknowledgments

This work was supported by a Department of Education GAANN fellowship and by the National Science Foundation (grant CCR-9876006).

7. References

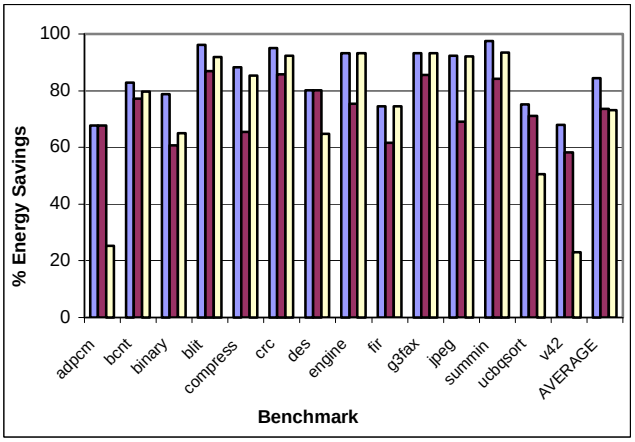
[1] Aditya, S., B. Rau, V. Kathail. Automatic Architectural Synthesis of VLIW and EPIC Processors. International Symposium on System Synthesis (ISSS), 1999.

[2] Bahar, R., G. Albera, S. Manne. Power and Performance Tradeoffs Using Various Caching Strategies. International Symposium on Low Power Electronics and Design (ISLPED), 1998.

[3] Bellas, N., I. Hajj, C. Polychronopoulos, G. Stamoulis. Energy and Performance Improvements in Microprocessor Desing Using a Loop Cache. International Conference on Computer Design, 1999.

[4] Benini, L., A. Macii, E. Macii, M. Poncino. Selective Instruction Compression for Memory Energy Reduction in Embedded Systems. International Symposium on Low Power Electronics and Design (ISLPED), 1999.

Figure 6: Best configuration for a benchmark (left bar) vs. best average configurations: configuration *30* (middle bar) and configuration *105* (right bar).



[5] Benini, L., G. Micheli, E. Macii, D. Sciuto, C. Silvano. Asymptotic Zero-Transition Activity Encoding for Address Busses in Low-Power Microprocessor-Based Systems. IEEE GLS-VLSI-97, 1997.

[6] Cotterell, S., F. Vahid. Synthesis of Customized Loop Caches For Core-Based Embedded Systems. International Conference on Computer Aided Design, 2002.

[7] Dutt, N. Memory Organization and Exploration for Embedded Systems-on-Silicon. International Conference on VLSI and CAD, 1997.

[8] Fisher, J. Customized Instruction-Sets For Embedded Processors. Design Automation Conference (DAC), 1999.

[9] Fisher, J., P. Faraboschi, G. Desoli. Custom-Fit Processors: Letting Applications Define Architectures. International Symposium on Microarchitecture (MICRO), 1996.

[10] Gonzales, R. Xtensa: A Configurable and Extensible Processor. International Symposium on Microarchitecture (MICRO), 2000.

[11] Gordon-Ross, A., S. Cotterell, F. Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. Computer Architecture Letters, Vol 1, 2002.

[12] Kavvadias, N., A. Chatzigeorgiou, N. Zervas, S. Nikolaidis. Memory Hierarchy Exploration For Low Power Architectures in Embedded Multimedia Applications. International Conference on Image Processing (ICIP), 2001.

[13] Kienhuis, B., E. Deprettere, K. Vissers, P. van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. Application-Specific Systems, Architectures, and Processors (ASAP), 1997.

[14] Kim, S., N. Vijaykrishnan, M. Kandemir, A. Sivasubramaniam, M. Irwin, E. Geethanjali. Power-aware Paritioned Cache Architectures. International Symposium on Low Power Electronics and Design, 2001.

[15] Kin, J., M. Gupta, W. Magione-Smith. The Filter Cache: An Energy Efficient Memory Structure. International Symposium on Microarchitecture (MICRO), 1997.

[16] Kirovski, D., J. Kin, W. Mangione-Smith. Procedure Based Program Compression. International Symposium on Microachitecture (MICRO), 1997.

[17] Ko, U., P. Balsara. Characterization and Design of A Low-Power, High-Performance Cache Architecture. International Symposium on VLSI Technology, Systems, and Applications, 1995.

[18] Lee, L., B. Moyer, J. Arends. Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops. International Symposium on Low Power Electronics and Design (ISLPED), 1999.

[19] Lee, L., B. Moyer, J. Arends. Low-Cost Embedded Program Loop Caching – Revisited. University of Michigan Technical Report CSE-TR-411-99, 1999.

[20] Lekatsas, H., J. Henkel, W. Wolf. Code Compression for Low Power Embedded System Design. Design Automation Conference (DAC), 2000.

[21] Malik, A., B. Moyer, D. Cermak. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. International Symposium on Low Power Electronics and Design. 2000.

[22] Mehta, H., R. Owens, M. Irwin. Some Issues in Gray Code Addressing. IEEE GLS-VLSI-96, March 1996.

[23] Nachtergaele, L., F. Catthoor, F. Balasa, F. Franssen, E. DeGreef, H. Samsom, and H. De Man., Optimization of Memory Organization and Hierarchy for Decreased Size and Power in Video and Image Processing Systems. International Workshop on Memory Technology, 1995.

[24] Panda, P., N. Dutt, A. Nicolau. Architectural Exploration and Optimization of Local Memory in Embedded Systems. International Symposium on System Synthesis (ISSS), 1997.

[25] Shiue, W., C. Chakrabarti. Memory Exploration for Low Power, Embedded Systems. Design Automation Conference (DAC), 1999.

[26] Slock, P., S. Wuytack, F. Catthoor, and G. Jong. Fast and Extensive System-Level Memory Exploration for ATM Applications. International Symposium on System-Level Synthesis, pp. 74-81, 1997.

[27] Stan, M., W. Burleson. Bus Invert for Low Power I/O. IEEE Transactions on VLSI, 1995.

[28] Su, C., C. Tsui, A. Despain. Cache Design Trade-offs for Power and Performance Optimization: A Case Study. International Symposium Low Power Design, 1995.

[29] Su, C., C. Tsui, A. Despain. Saving Power in the Control Path of Embedded Processors. IEEE Test and Design of Computers, Vol. 11, No. 4, 1994.

[30] Vahid, F., T. Givargis, Platform Tuning for Embedded Systems Design. IEEE Computer, Vol. 34, No 3, 2001.