Synthesis of Customized Loop Caches for Core-Based Embedded Systems

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 Computing at UC Irvine

Abstract

Embedded system programs tend to spend much time in small loops. Introducing a very small loop cache into the instruction memory hierarchy has thus been shown to substantially reduce instruction fetch energy. However, loop caches come in many sizes and variations – using the configuration best on the average may actually result in worsened energy for a specific program. We therefore introduce a loop cache exploration tool that analyzes a particular program's profile, rapidly explores the possible configurations, and generates the configuration with the greatest power savings. We introduce a simulation-based approach and show the good energy savings that a customized loop cache yields. We also introduce a fast estimation-based approach that obtains nearly the same results in seconds rather than tens of minutes or hours.

Keywords

Low power, low energy, tuning, loop cache, embedded systems, instruction fetching, customized architectures, memory hierarchy, estimation, synthesis.

1. Introduction

Reducing power and energy consumption of embedded systems translates to longer battery life and reduced cooling requirements. For embedded microprocessor based systems, instruction fetching can contribute to a large percentage of system power (50% in [18][10] and 43% in [23]), since such fetching occurs on nearly every cycle, involves driving of long and possibly off-chip bus lines, and may involve reading of numerous memories – 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 [3][15][20], bus encoding to reduce the number of switched wires [4][22][27][29] and efficient instruction cache design [2][13][16][28].

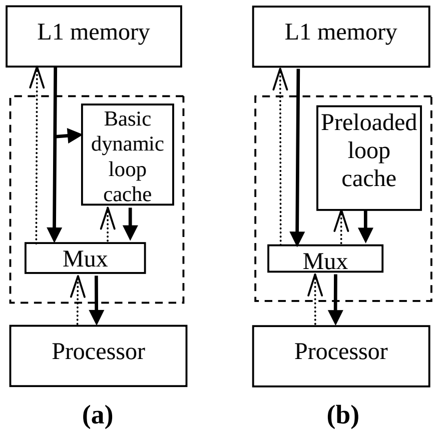
Another category of approaches, which capitalize on the common feature of embedded applications spending much time in small loops [19][32], integrate a tiny (perhaps 64 word) instruction cache with the microprocessor. Such tiny caches have extremely low power per access, perhaps 50 times less than regular instruction memory access [19], thus reducing total instruction fetch energy substantially. A filter cache [14] is one such tiny cache, implemented as a direct-mapped cache, but a filter cache may result in many misses and hence performance degradation. Tagless, missless low-power tiny instruction cache architectures have recently been introduced, including the dynamically-loaded tagless loop cache [18][19], which shows average power savings of 30-40%, followed by the preloaded tagless loop cache [9], which shows average power savings of 60-70%.

A designer of a mass-produced microprocessor platform might use 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 typically 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. [31], as discussed in the Y-chart methodology of [12]. A very aggressive form of tuning involves creating a customized instruction set [1][6][7][8].

Complementary to such application-specific instruction-set processor design is the design of customized memory architectures. Traditionally, these architectures have focused on the data memory organization and fast exploration. One such approach attempts to reduce power consumption by reducing memory traffic through memory optimizing transformations, storing frequently accessed variables in register files and on-chip cache, reducing misses by configuring the cache size correctly and data placement [26]. Another approach uses an exploration strategy for determining the on chip memory architecture [25]. In this approach, they focused on a memory architecture comprised of a Scratch-pad memory and cache parameters to decrease off chip memory traffic. To reduce system power, a hardware based approach is presented in [11], where they create a custom memory hierarchy consisting of additional layers of smaller memories in which the more frequently used data is stored. Furthermore, in [24] they present an exploration environment that utilizes a two phase memory exploration scheme along with system level transformations to reduce memory size and power.

Loop cache designs vary greatly as well. Choosing the right style and size of a loop cache can mean the difference between an 80% instruction fetch savings and no savings or even a loss. In this paper, we describe our automated environment for synthesizing the best loop cache architecture for a given

Figure 1: Basic architecture of dynamic loop cache (a) and preloaded loop cache (b).



program. We show excellent energy savings using a simulation based environment. Furthermore, we describe an estimation method that results in nearly two orders of magnitude speedup, enabling best cache selection in just seconds with almost no loss in result quality.

2. Loop Cache Architectures

The first type of loop cache we consider was proposed in [18], which was using in Motorola's M*CORE embedded processor. The loop cache, illustrated in Figure 1(a), is a small instruction buffer tightly integrated with the processor, having no tag comparison or valid bit. To eliminate performance degradation, the loop cache is not a genuine first level cache, but rather an alternative location from which to fetch instructions, selected through a multiplexor (controlled by the loop cache controller) as shown in the figure. The 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 loop, a 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. This type of loop cache is dynamically loaded, and requires no special compilers or instructions and is thus transparent to the designer. We refer to this type of 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. To alleviate this problem, the *flexible dynamic loop cache* was proposed in [19]. 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.

A problem with the dynamically loaded loop caches is that a control of flow change in a loop will cause filling or fetching from the loop cache to stop. Thus, internal branches within loops, multiple backwards branches to the same starting point in a loop, nested loops, and subroutines pose problems. A preloaded loop cache was proposed in [9] to overcome these limitations, illustrated in Figure 1(b). Using profiling information gathered for a particular application, the loops that comprise the largest percentage of execution time are selected and preloaded into the loop cache during system reset. After this initialization, the contents of the loop cache do not change for the duration of the program execution. 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 the second rather than third loop iteration. This approach is referred to as the *preloaded loop cache (sbb)*, which stands for short backwards branch. Alternatively, loop addresses can be looked for on every instruction, allowing loop cache fetching to begin on the first iteration. This approach is referred to as the *preloaded loop cache (sa)*, where *sa* stands for starting address. Both preloaded caching schemes can handle control of flow, but limit the number of loops that can be stored in the loop cache. In addition, preloaded loop caches are not transparent to the designer, but instead requires that programs be profiled beforehand and the loop cache be preloaded.

We thus see that a variety of loop cache configurations are possible. We described four basic styles above. Furthermore, for each style, different sizes are possible – larger loop caches can hold bigger loops, but at the expense of more power per access. Additionally, preloaded loop caches can support different numbers of loops. The best configuration depends directly on what program we are considering. Generally, a dynamically-loaded loop cache will work best for programs with large numbers of small, straight-line loops. A preloaded loop cache will work best for programs with a few key loops that possess control of flow changes. Furthermore, the best cache size will depend on the loop sizes.

3. Exploration Framework

We evaluated a number of cache architecture configurations on a set of Powerstone benchmarks [21]. For each benchmark, we considered 72 different cache configurations:

- original dynamic loop cache – cache sizes ranging from 8 to 1024 entries (by powers of 2).
- flexible dynamic loop cache – cache sizes ranging from 8 to 1024 entries (by powers of 2).
- preloaded loop cache using start address – cache sizes ranging from 8 to 1024 entries (by powers of 2), with either 2 or 3 loop address registers
- preloaded loop cache using short backwards branch address – cache sizes ranging from 8 to 1024 entries (by powers of 2), with 2-6 loop address registers

Each entry within a loop cache corresponds to a 32-bit instruction. 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.

3.1 Loop Cache Simulation Method

We developed a suite of tools to evaluate each cache configuration for a given benchmark. The tool chain is shown in Figure 2. Starting from C code for each benchmark, an lcc compiler ported to the MIPS instruction set compiles each application. We then use a MIPS instruction-set simulator to obtain a *program instruction trace* for each benchmark. The program instruction trace is then fed to a loop analysis tool called LOOAN [33]. LOOAN analyzes the original assembly program and the trace, from which it generates *loop statistics* describing the hierarchy of subroutines and loops, and detailed profiling statistics such as the number of executions of a particular loop, the number of dynamic instructions per loop iteration, and the total contribution of each loop and subroutine to the entire program execution. The loop statistics are fed to a *loop packer & script generator* tool that selects the best loops to store in a preloaded loop cache, and generates a script that will explore all 72 possible loop cache configurations with associated loop packings for the preloaded caches. We developed a loop cache simulator, called *lcsim*, that the script calls for each configuration. The simulator reads the program instruction trace, and keeps an accurate count of important loop cache operations, including the number of fill operations (for a dynamically loaded cache), the number of instruction-memory fetches, the number of loop cache fetches, the number of address comparisons (for a preloaded cache), etc. *lcsim* generates these *loop cache statistics* as a file. Finally, an *lc power calc* tool reads these statistics, as well as technology parameters, and the loop cache power information, to generate *loop cache power* data.

We calculated power and energy based on the switching activity of each operation and the relative capacitance of various components. The switching activity for each operation was measured by pre-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 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. A designer interested in determining how each cache configuration performs for a specific technology can simply set the performance value to the technology of interest.

The simulation-based approach mimics each loop cache controller exactly (*lcsim* contains exactly the same state machines as our VHDL loop cache controller models), and thus yields completely accurate counts of all loop cache related events (fills, fetches, compares, etc.). However, such an approach took anywhere from a couple minutes for small examples, to half an hour for medium sized examples. We also ran several larger examples through *lcsim* (from MediaBench [17]), which took tens of hours to complete. Most of the time came from having to read the very large program trace files for each configuration being examined.

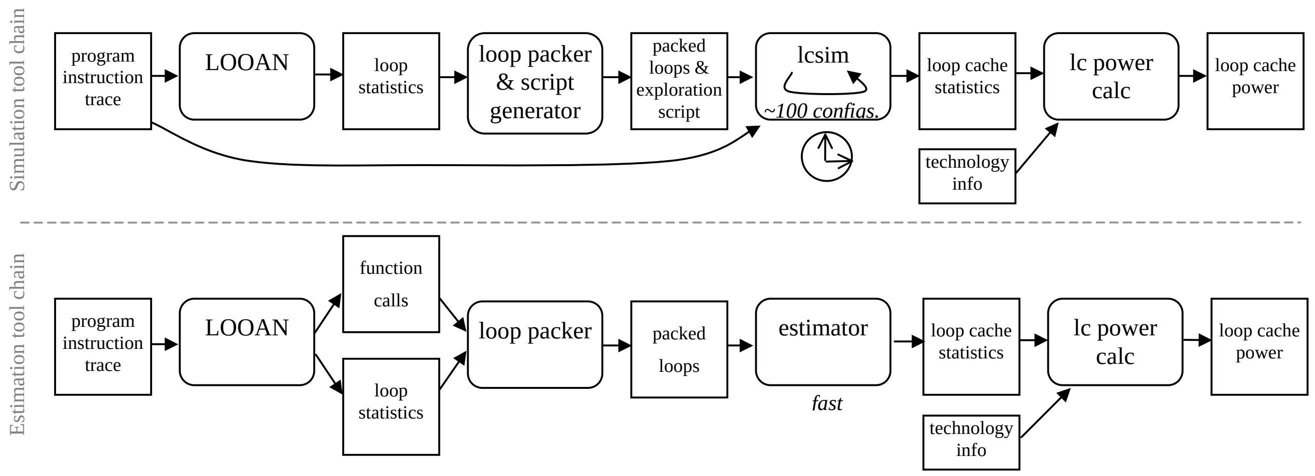
3.2 Loop Cache Estimation Method

We sought to develop a faster method than loop cache simulation for exploring the loop cache configuration space. Note that *lcsim* follows the paradigm of traditional cache simulators, like Dinero [5]. Thus, we could look into traditional cache simulation speedup methods, like examining multiple configurations per pass [30] or compacting the trace size using statistical methods [34]. However, we found that due to the nature of loop caches, a far faster and simpler estimation method was possible. In short, we could apply simple equations and algorithms to the loop statistics in order to generate very accurate loop cache statistics.

Our estimation method is illustrated in Figure 2. LOOAN is again used to generate loop statistics for a program. For our estimations, the loop statistics we are interested in are the start and end address of each loop, the loop size, the number of times a loop is called, the average times a loop iterates once it is called, and the total number of instructions executed by this loop. LOOAN was further augmented to output the addresses at which function calls were made to create a more complete picture of the executing program. Once this data is generated, we then use the various estimation techniques described below to statically analyze each benchmark. The goal of each estimation technique is to determine the loop cache statistics – the number of instruction memory fetches, detection operations (i.e. checking to see if we should execute from the loop cache or not), number of instructions filled into loop cache, and the number of instructions fetched from the loop cache – without running the time-consuming *lcsim* at all.

To determine the number of various operations we take the loop hierarchy provided by LOOAN and iterate through each loop. Then, for each loop we accumulate the estimated number of fills, fetches, detects corresponding to only the loop we are currently investigating. The estimation method varies according to the cache type being considered. We now discuss the estimation strategy for each loop cache type.

Figure 2: Simulation and estimation based loop cache configuration synthesis methods.



3.2.1 Original Dynamically Loaded Loop Cache

In the original dynamically loaded loop cache, we are interested in the number of times we fill the loop cache with an instruction, the number of times we fetch an instruction from the loop cache, and the number of instruction memory fetches. Since the original dynamically loaded loop cache contains no preloaded loops, there are no loop address registers we must compare addresses with, thus no detect operations.

On the first iteration of each loop, the loop cache controller sees a short backwards branch (sbb) that triggers filling the loop cache on the second iteration. It will continue to fill the loop cache until a control of flow is detected. Thus, to estimate the number of fill instructions, we first see if the loop size is less than or equal to the size of the loop cache we are interested in. Next, we check whether this loop would iterate at least two times, since otherwise the loop cache would never be filled with this loop. We then want to see how many instructions from this loop would be filled into the loop cache. We determine where the first control of flow will occur. This control of flow can originate from the sbb that trigged the fill, an sbb from a subloop, or a function call. The control of flow may also correspond to a jump, but this information is not provided in the static analysis. If the current loop contains subloops, the loop cache controller will fill to the end of the first subloop. Similarly, if the loop contains a function call, the function call map previously generated from LOOAN is used to determine the exact instruction from which the function call originates. The smallest of the three aforementioned addresses is determined and from it we subtract the start address of the loop we are interested in. This calculation is the number of instructions that will be filled into the loop cache. Therefore, each time this particular loop is called, it will fill that many instructions, so we then multiply this number by the number of times the loop is executed.

The original dynamic loop cache fetches instructions starting with the third iteration of the loop. Once again, the loop cache controller will stop fetching when a control of flow is detected. To calculate the number of fetches, we again check to see if the loop size is less than or equal to the size of the cache we are interested in. In addition, we check to see that the average number of iterations is greater than or equal to three. If not, this loop will never be fetched from the loop cache. The location of the first control of flow change within the loop is determined using the same method as mentioned above. If this control of flow change occurs at the end of the loop, the loop will be fetched from the loop cache starting with the third iteration. Thus, we multiply the number of instructions within the loop by the iteration average minus two. Additionally, this behavior occurs every time the loop is executed, hence, we multiply the fetches per execution by the number of times the loop is called.

Finally, we fetch an instruction from instruction memory when it is not fetched from the loop cache. Using the output from LOOAN indicating the total number of instructions executed, we obtain the number of instruction memory fetches by subtracting from the total number of instructions executed the number of fetch operations we previously determined.

3.2.2 Flexible Dynamically Loaded Loop Cache

The estimation method for the flexible dynamically loaded loop cache is similar to the estimation described for the original dynamically loaded loop cache. However, in the flexible dynamically loaded loop caches, loop size is not limited to less than or equal to the loop cache size. Thus, in determining the number of instructions fetched or filled we still determine the first control of flow, whether it be an sbb from a subloop, a function call, or the sbb corresponding to the end of this loop. We then check to see if the start address minus the end address is larger then the loop cache size. If it is, we set the number of instructions fetched or filled on a given iteration to the loop cache size. This value is still multiplied by number of times the loop is called if we are calculating the number of fills or by the number of times the loop is called and the number of iterations minus two if we are calculating the number of fetches. As before, the number of instruction memory fetches is equal to the number of total instructions executed minus the number of loop cache fetches.

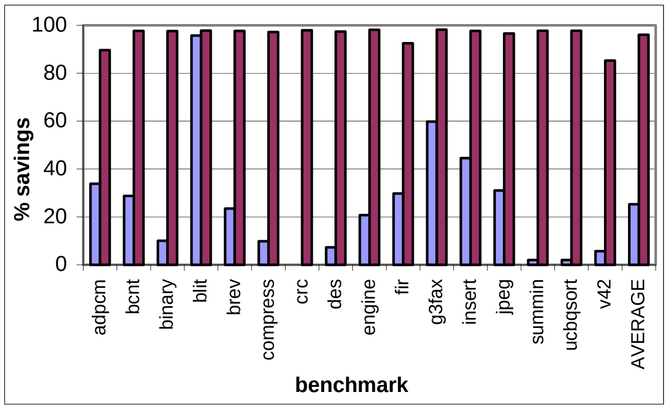
3.2.3 Preloaded Loop Cache (SA)

The preloaded loop caching scheme requires that we select loops beforehand via profiling. These loops are never replaced, thus the number of dynamic fills for this type of caching scheme is always zero (fills occur before regular program execution). Depending on the number of loops allowed in the loop cache, we have corresponding loop address registers that indicate which loops have been preloaded. With this caching scheme, we start fetching from the loop cache on the first iteration of the loop.

To determine the number of instructions fetched from the loop cache, we see if the current loop we are looking at was selected for preloading. If so, every instruction corresponding to that loop will always be fetched from the loop cache since the preloaded loop will always remain in the loop cache. The output from LOOAN indicates the number of instructions executed by the loop, thus the number of fetches is equal to the number of instructions executed by the current loop.

The start addresses of the preloaded loops are kept in the loop address registers. This means we are not able to wait for an sbb to detect if the current instruction is within the loop cache. Instead, for every instruction not fetched from the loop cache, we must compare its address with each of the loop address registers to see if we should indeed fetch from the loop cache. Thus, the number of detects is equal to the number of total instructions executed minus the number of instructions fetched from the loop cache. For each detect operation, we must compare it with each of the loop address registers so we multiply the aforementioned value by the number of loop address registers. In addition, in order to start fetching from the preloaded loop cache, we must initially check to determine if the current instruction address is located within the loop cache. To accommodate this behavior, we add to the number of detects the number of times the current loop is called multiplied by the number of loop address registers. Furthermore, each time the loop makes a function call we must jump to the function then jump back to the caller. When jumping to the function call we must see if the function is preloaded, and when returning from the function we must determine if the returning loop is preloaded. Thus, each function call results in two detect

Figure 3: Percent savings using a 32 entry flexible loop cache (configuration 11) (left) versus using optimal cache configuration for each benchmark (right).



operations. Therefore, we search the loop hierarchy to see if any of the instructions within the current loop make a function call. If so, we add to the number of detect operations the number of loop address registers times two for each function call.

To determine the number of instruction memory fetches, we once again subtract from the total number of instructions executed, as reported by LOOAN, the number of fetches from the loop cache.

3.2.4 *Preloaded Loop Caches (SBB)*

The preloaded loop cache (sbb) scheme is almost identical to the preloaded loop cache (sa) scheme. In this loop cache design, the loops are again selected and loaded before hand. Additionally, once the loops are loaded they do not change during the course of the program. However, unlike the preloaded loop cache (sa) scheme, the address of the sbb at the end of each loop is stored in the loop address registers. The loop cache controller will wait until a control of flow change to determine if the address is in the loop cache. Therefore, we do not fetch from the loop cache until the second iteration of the loop.

Given that the loops are preloaded the number of instructions filled into the loop cache is zero for the execution of the program.

Since loops are preloaded, only those loops will contribute to the number of fetches. In addition, we must wait for a control of flow to trigger the controller to compare the address with the loop address registers to see if the loop is preloaded. There are two cases which we can determine statically when a control of flow occurs. The first case is when the loop executes it's first iteration, when the loop reaches it's sbb this will trigger a detect and on the second iteration we fetch the corresponding instructions from the loop cache. Thus, the number of fetches contributed by this loop is equal to the number of times the loop is executed, multiplied by the number of average iterations minus 1. The second potential control of flow occurs when a function is called from within the loop. There is a control of flow to call the function and a control of flow to return from the function. Upon returning to the loop from the function call, the control of flow change will trigger fetching starting from the location following the function call. Thus, in this situation the number of fetches contributed by this loop is equal to the total number of instructions executed. However, to account for the above situation, the number of executions multiplied by the difference between the function call address and the starting address of the loop is subtracted from the total for this loop.

The number of detect operations corresponds to the number of control of flows in the given program. There exists a control of flow at the end of a loop, when calling a function, and when returning from a function. Thus for every loop we find, we add a detect operation to represent the sbb at the end of the loop. For every function call we add two detect operations.

Once we have gathered the various statistics we are interested in for each of the cache configurations considered, we feed this information into another program, which calculates the power of each loop cache design. The relative capacitance values can be varied in this stage so that the power values outputted correspond to the desired technology.

4. Results

Many times designers find a configuration that does relatively well and use this particular configuration for all applications. Figure 3 presents the percent power savings using a flexible 32 entry loop cache (configuration 11) versus the optimal loop cache configuration for each benchmark. Although the flexible loop cache performs fairly well for most examples, however, if we had selected the optimal cache configuration on a per application basis, an additional 70% power savings can be achieved on average. This clearly demonstrates that simply selecting a cache configuration that performs well and using it for all applications leaves substantial room for improvement. Thus, a tool for automatically determining the optimal loop cache configuration provides a great advantage.

In evaluating our simulation- and estimation-based approaches, we need to analyze the results for each with respect to accuracy and fidelity between the two approaches. Due to the number of loop cache configurations evaluated, in order to facilitate plotting of so many configurations, we associate each configuration with a numerical code, 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 pre-loaded loop cache using start address, an 8 entry cache with 2 loop address registers is referred to with 17 and an 8 entry cache with 3 loop address register is configuration 18.

To determine the accuracy of the estimation method, we first ran each of the benchmarks through the loop cache simulator to obtain the power savings for each cache configuration over a configuration without a cache. Next, each benchmark was run through the loop cache estimator to obtain the power savings of each cache configuration over a configuration without a cache. Figure 4 compares the reported

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
Pre-loaded Loop Cache (SA)	8-1024	2-3 loops	17-32
Pre-loaded Loop Cache (SBB)	8-1024	2-6 loops	33-72

Figure 4: Percent savings for various cache configurations, savings using simulation (left) versus savings using estimation (right) for the *jpeg* benchmark.

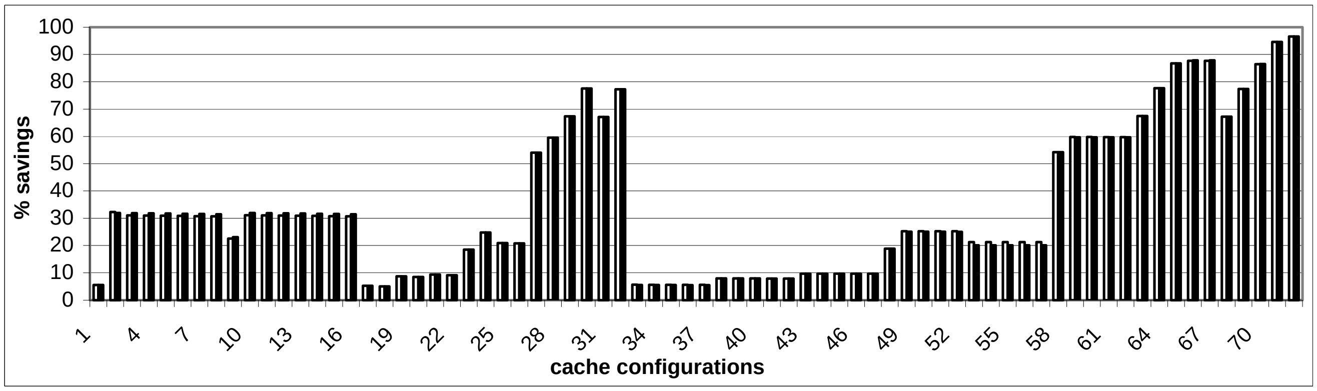


Figure 5: Percent savings for various cache configurations, savings using simulation (left) versus savings using estimation (right) for averages over all benchmarks

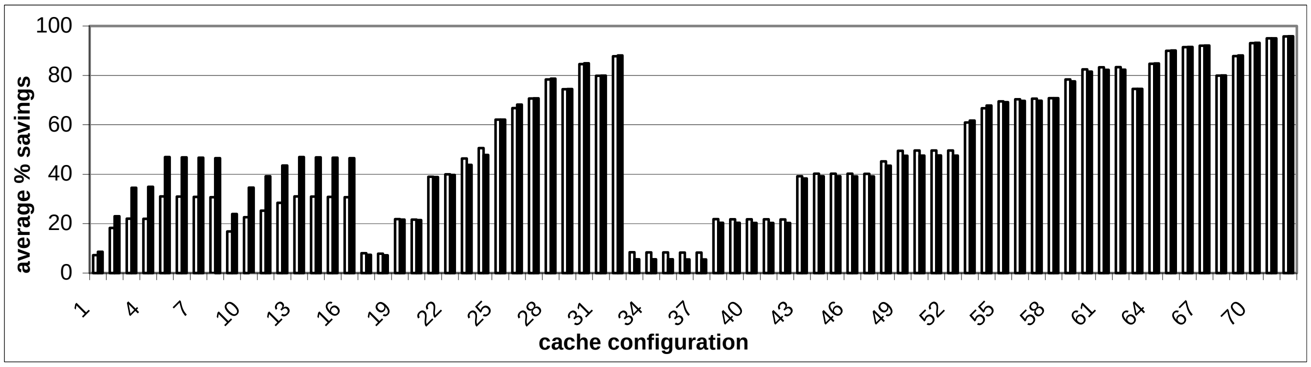
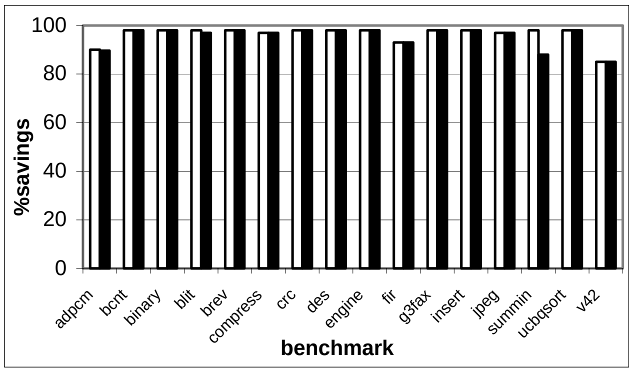


Figure 6: Power savings using cache configurations from simulation approach (left) versus estimation approach (right)



power savings obtained through simulation versus the reported power savings through estimation for the *jpeg* benchmark. By simply looking at the graph, it is easy to determine that the estimated results are very close to the simulation-based results. Specifically, on average, the power savings reported by the estimation method and the simulation method differ by less than 1%.

We then compared the average power savings reported for each cache configuration over all benchmarks using the simulation based method versus the average power savings reported for each cache configuration over all benchmarks using the estimation based method. This comparison is shown in Figure 5. For the dynamic loop caches (original and flexible), the estimation method reported approximately 15% more power savings than reported by the simulation-based results. For the preloaded loop caches (sa and sbb), the estimator reported –1% to 3% difference in power savings. However, on average the estimation methodology reported 2% more power savings over the simulation based methodology.

While the relative accuracy of the estimated power savings is important, in order for this approach to be viable, there must be fidelity between the choices selected under each approach as the best loop cache configuration. Therefore, to ensure any inaccuracies from estimation do not compromise the fidelity, for each benchmark we selected the loop cache configuration chosen as the best by both the simulation-based approach and the estimation-based approach. Figure 6 shows the power savings for the cache configuration selected by the simulation-based approach versus the power savings using the cache configuration selected by the estimation-based approach across all benchmarks. In most cases, the cache configuration selected by the estimation method saves as much power as the cache configuration selected by the simulation methodology. The worst difference in performance of the loop cache obtain from estimation versus simulation is for the *summin* benchmark, where the estimation approach selects a cache configuration that

Table 2: Speedup of simulation verses estimation (in seconds)

		Simulation Tool Chain					Estimation Tool Chain				
Benchmark	Number Instr Executed	LOOAN	Script Gen	lcsim	lc Power Calc	Total Simulation Time	LOOAN	Estimator	lc Power Calc	Total Estimation Time	**Speedup**
adpcm	63891	0.31	0.01	32.15	0.01	32.48	0.31	0.16	0.01	0.48	**68**
bcnt	1938	0.01	0.01	1.17	0.01	1.20	0.02	0.06	0.01	0.09	**13**
binary	816	0.01	0.01	0.87	0.01	0.90	0.01	0.08	0.01	0.10	**9**
blit	22845	0.07	0.01	7.26	0.01	7.35	0.07	0.06	0.01	0.14	**53**
brev	2377	0.01	0.01	1.20	0.01	1.23	0.01	0.06	0.01	0.08	**15**
compress	138573	0.85	0.01	82.50	0.01	83.37	0.85	0.14	0.01	1.00	**83**
crc	37650	0.15	0.01	16.03	0.01	16.20	0.15	0.07	0.01	0.23	**70**
des	122214	0.44	0.02	45.28	0.01	45.75	0.44	0.07	0.01	0.52	**88**
engine	410607	2.12	0.02	214.99	0.01	217.14	2.12	0.08	0.01	2.21	**98**
fir	16211	0.07	0.02	7.60	0.01	7.70	0.07	0.09	0.01	0.17	**45**
g3fax	1128023	3.54	0.02	385.44	0.01	389.01	3.54	0.09	0.01	3.64	**107**
insert	1942	0.01	0.01	1.18	0.01	1.21	0.01	0.06	0.01	0.08	**15**
jpeg	4594721	17.57	0.01	1837.28	0.01	1854.87	17.57	0.12	0.01	17.7	**105**
summin	1909787	11.42	0.01	903.73	0.01	915.17	8.25	0.09	0.01	8.35	**110**
ucbqsort	219978	0.93	0.01	82.62	0.01	83.57	0.89	0.09	0.01	0.99	**84**
v42	2442551	12.07	0.01	1252.48	0.01	1264.57	12.27	0.12	0.01	12.4	**102**
										AVERAGE	**67**

is 10% less than the optimal configuration. However, on average the cache configuration obtained through estimation is less than 1% away from the optimal reported by the simulation method.

We have shown that through estimation we have good accuracy and preserve fidelity. Now we describe the speedup obtained by using estimation rather than simulation. Table 2 shows the breakdown of time spent in various areas of the simulation based approach for each benchmark. In addition, the breakdown of time spent in various areas of the estimation based approach for each benchmark is also shown. All simulations and estimations were executed on a 500 MHz Sun Ultra60 workstation. From Table 2, it can be seen that the majority of time for the simulation-based method is spent running the loop cache simulator (lcsim). Thus, by decreasing this time by using estimation, a significant speed up is achievable. For the larger examples, jpeg, summin, and v42, the simulation based approach required approximately 30 minutes, 15 minutes, and 21 minutes, respectively. However, by using the estimation based method the time required were reduced to approximately 17 seconds, 8 seconds, and 12 seconds, respectively. While, many of the other benchmarks did not require a very long time for simulation due to their small size, the estimation approach still resulted in significant speed up. Overall, the speedup using estimation ranges from 9 to 109 across various benchmarks, with an average speed up of 66. We have begun to investigate even larger examples from MediaBench, and are finding that the simulation based method takes tens of hours, while estimation still requires only seconds to minutes.

After we explore the design space, we know exactly which loop cache configuration would yield the greatest savings. We create a simple tool that generates synthesizeable VHDL code corresponding to the desired cache configuration.

5. Conclusions

We have shown that synthesizing a custom loop cache yields good energy savings for embedded applications and thus is an important part of a core-based embedded system design flow. We have implemented a simulation based tool that finds the loop cache configuration yielding the best energy savings for a given program. We have also implemented a fast estimation based method that obtains nearly the same savings but in seconds rather than hours.

Our future work includes investigating additional loop cache structures and investigating a wider variety of benchmarks.

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. Int. Symp. on System Synthesis, 1999.

[2] Bahar, R., G. Albera, S. Manne. Power and Performance Tradeoffs using Various Caching Strategies. Int. Symp.on Low Power Electronics and Design, 1998.

[3] Benini, L., A. Macii, E. Macii, M. Poncino. Selective Instruction Compression for Memory Energy Reduction in Embedded Systems. Int. Symp. on Low Power Electronics and Design, 1999.

[4] 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.

[5] Elder, J., M.D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. http://www.cs.wisc.edu/~markhill/DineroIV.

[6] Fisher, J. Customized Instruction-Sets For Embedded Processors. Design Automation Conference, 1999.

[7] Fisher, J., P. Faraboschi, G. Desoli. Custom-Fit Processors: Letting Applications Define Architectures. Int. Symp. on Microarchitecture, 1996.

[8] Gonzales, R. Xtensa: A Configurable and Extensible Processor. Int. Symp. on Microarchitecture, 2000.

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

[10] Kalambur, A., M. J. Irwin. An Extended Addressing Mode for Low Power. Int. Symp. on Low Power Electronics and Design, 1997.

[11] Kavvadias, N., A. Chatzigeorgiou, N. Zervas, S. Nikolaidis. Memory Hierarchy Exploration For Low Power Architectures in Embedded Multimedia Applications. Int. Conf. on Image Processing, 2001.

[12] 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, 1997.

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

[14] Kin, J., M. Gupta, W. Magione-Smith. The Filter Cache: An Energy Efficient Memory Structure. Int. Symp. on Microarchitecture, 1997.

[15] Kirovski, D., J. Kin, W. Mangione-Smith. Procedure Based Program Compression. Int. Symp. on Microachitecture, 1997.

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

[17] Lee, C., M. Potkonjak, W. Magione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. International Symposium on Microarchitecture, 1997.

[18] Lee, L., B. Moyer, J. Arends. Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops. Int. Symp. on Low Power Electronics and Design, 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, 2000.

[21] Malik, A., B. Moyer, D. Cermak. A Low Power Unified Cache Architecture Providing Power and Performance Flexibility. Int. Symp. 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] Montanaro, J., et. al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. IEEE Journal of Solid State Circuits, 1996.

[24] 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. Int. Workshop on Memory Technology, 1995.

[25] Panda, P., N. Dutt, A. Nicolau. Architectural Exploration and Optimization of Local Memory in Embedded Systems. Int. Symp. on System Synthesis, 1997.

[26] Shiue, W., C. Chakrabarti. Memory Design and Exploration for Low Power, Embedded Systems. Journal of VLSI Signal Processing – Systems for Signal, Image, and Video Technology, Vol. 29, No. 3, pp. 167-178, 2001.

[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. Int. Symp. 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] Sugumar, R., and S. Abraham. Efficient Simulation of Multiple Cache Configurations using Binomial Trees. Technical Report CSE-TR-111-91, CSE Division, University of Michigan, 1991.

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

[32] Villarreal, J., D. Suresh, G. Stitt, F. Vahid, and W. Najjar. Improving Software Performance with Configurable Logic. Design Automation of Embedded System, 2002.

[33] Villarreal, J., R. Lysecky, S. Cotterell, and F. Vahid. A Study on the Loop Behavior of Embedded Programs. Technical Report UCR-CSE-01-03, University of California, Riverside, 2002.

[34] Wu, Z, and W. Wolf. Iterative Cache Simulation of Embedded CPUs with Trace Stripping. International Conference on Hardware/Software Co-Design, 1999.