

UNIVERSITY OF CALIFORNIA  
RIVERSIDE

Customization of Loop Caches For Embedded Systems Design

A Thesis submitted in partial satisfaction  
of the requirements for the degree of

Master of Science

in

Computer Science

by

Susan J. Cotterell

August 2003

Thesis Committee:

Dr. Frank Vahid, Chairperson

Dr. Walid Najjar

Dr. Jun Yang

Copyright by  
Susan J. Cotterell  
2003

The Thesis of Susan J. Cotterell is approved:

---

---

---

Committee Chairperson

University of California, Riverside

## ABSTRACT OF THE THESIS

Customization of Loop Caches for Embedded Systems Design

by

Susan J. Cotterell

Master of Science, Graduate Program in Computer Science  
University of California, Riverside, August, 2003  
Dr. Frank Vahid, Chairperson

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, 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. Furthermore, we also introduce a fast estimation-based approach that obtains nearly the same results in seconds rather than tens of minutes or hours.

# Contents

|   |    |
|---|----|
| <b>1. Introduction</b>                        | 1  |
| <b>2. Loop Cache Architectures</b>            | 4  |
| 2.1. Overview                                 | 4  |
| 2.2. Filter Cache                             | 4  |
| 2.3. Dynamically Loaded Loop Caches           | 5  |
| 2.4. Preloaded Loop Caches                    | 8  |
| 2.5. Loop Cache Selection                     | 10 |
| <b>3. Loop Cache Simulation</b>               | 11 |
| 3.1. Overview                                 | 11 |
| 3.2. Loop Cache Simulation Methodology        | 12 |
| 3.3. Simulation Results                       | 14 |
| <b>4. Loop Cache Estimation</b>               | 19 |
| 4.1. Overview                                 | 19 |
| 4.2. Loop Cache Estimation Methodology        | 20 |
| 4.2.1. Original Dynamically Loaded Loop Cache | 22 |
| 4.2.2. Flexible Dynamically Loaded Loop Cache | 24 |
| 4.2.3. Preloaded Loop Cache (SA)              | 24 |

|                                   |           |
|-----------------------------------|-----------|
| 4.2.4. Preloaded Loop Cache (SBB) | 26        |
| 4.3. Estimation Results           | 27        |
| <b>5. Conclusions</b>             | <b>33</b> |
| 5.1. Summary                      | 33        |
| 5.2. Future Work                  | 34        |
| <b>References</b>                 | <b>35</b> |

# List of Figures

|           |   |    |
|-----------|---|----|
| <b>1</b>  | Loop Cache Architectures.   | 5  |
| <b>2</b>  | Loop Cache Controller for Dynamically Loaded Loop Caches.   | 6  |
| <b>3</b>  | Loop Cache Controller for Basic Preloaded Loop Caches.  | 9  |
| <b>4</b>  | Loop Cache Simulation-Based Tool Chain.   | 13 |
| <b>5</b>  | Energy Savings for <i>blit</i> Given Various Cache Configurations.  | 15 |
| <b>6</b>  | Energy Savings for <i>v42</i> Given Various Cache Configurations.   | 15 |
| <b>7</b>  | Average Energy Savings for Various Cache Configurations.  | 16 |
| <b>8</b>  | Average Performance Penalty for Filter Cache.   | 17 |
| <b>9</b>  | Best Configuration for a Benchmark (left bar) vs. Best Average Configurations: Configuration 30 (middle bar) and Configuration 105 (right bar).             | 18 |
| <b>10</b> | Loop Cache Simulation- and Estimation-Based Tool Chain.   | 20 |
| <b>11</b> | Percent Savings For Various Cache Configurations, Savings Using Simulation (left) Versus Savings Using Estimation (right) For The <i>jpeg</i> Benchmark.    | 28 |
| <b>12</b> | Percent Savings For Various Cache Configurations, Savings Using Simulation (left) versus Savings Using Estimation (right) For Averages Over All Benchmarks. | 28 |

**13** Power Savings Using Cache Configurations from Simulation Approach (left)  
verses Estimation Approach (right).

29

# List of Tables

|          |  |    |
|----------|--|----|
| <b>1</b> | Benchmark Descriptions.                                | 12 |
| <b>2</b> | Corresponding Codes for Various Cache Configurations.  | 14 |
| <b>3</b> | Simulation Times verses Estimation Times (in seconds). | 30 |

# Chapter 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 [20]), 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 [4][17][22], bus encoding to reduce the number of switched wires [5][24][29][31] and efficient instruction cache design [2][15][18][30].

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. [33], as discussed in the Y-chart methodology of [14]. 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 [6][13][25][26][27][28]. Traditionally, these architectures have focused on 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 [27]. Another approach uses an exploration strategy for determining the on-chip memory architecture [26]. 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 [13], 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 [25] they present an exploration environment that utilizes a two phase memory exploration scheme along with system level transformations to reduce memory size and power.

Another category of approaches, which capitalize on the common feature of embedded applications spending much time in small loops [21][34], 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 [21], thus reducing total instruction fetch energy substantially. A filter cache [16] 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 [20][21], which shows average power savings of 30-40%, followed by the preloaded tagless loop cache [11][12], which shows average power savings of 60-70%.

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 2-40% energy savings compared to the architecture that is best for the entire set of benchmarks. Variation would be even greater for more diverse benchmarks. 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.

# Chapter 2

## Loop Cache Architectures

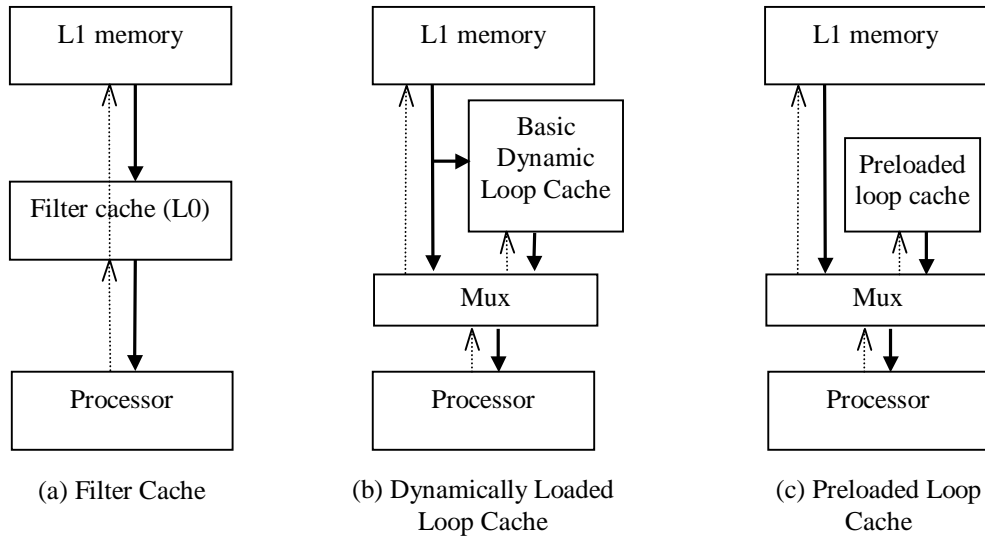
### 2.1 Overview

Loop cache designs vary greatly. Choosing the right style and size of a loop cache can mean the difference between an 80% instruction fetch savings, no savings, or even a loss. Not only can each type of cache vary in size but also in certain features. Below we describe four basic loop cache architectures we considered in our evaluation framework in more detail.

### 2.2 Filter Cache

The filter cache proposed in [16] is an unusually small direct-mapped cache. The filter cache, shown in Figure 1(a), is placed between the CPU and the L1 cache and utilizes standard tag comparison and miss logic. Because the filter cache is smaller than the L1 cache, it will have a faster access time and cost less to access. However, because the cache is so small, it will suffer from a lower hit rate and decreased performance. Profile-guided compilation was proposed in [3] to reduce misses.

Figure 1: Loop Cache Architectures.

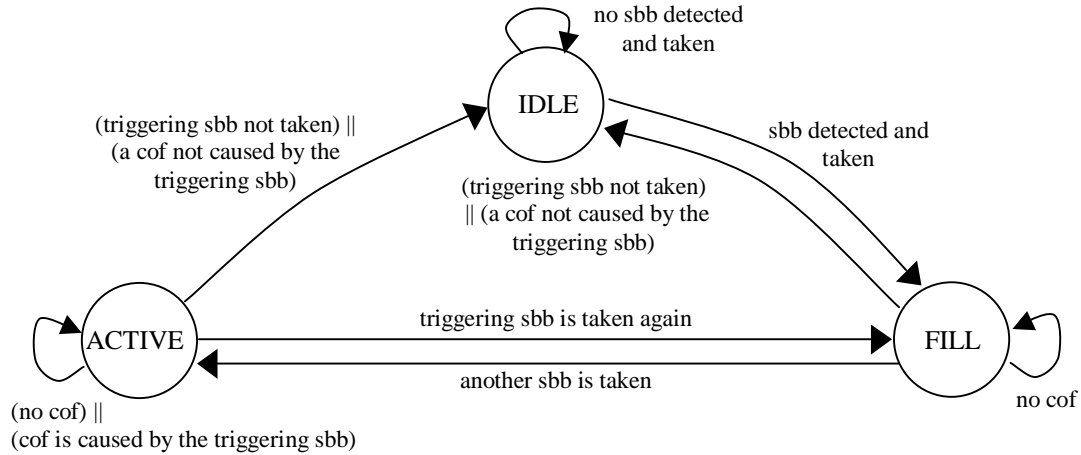


Architectural variation for the filter cache involve different cache sizes. Larger filter caches may have a lower miss rate but will have higher power per access. In this work we consider various sizes of the original filter cache design, which did not use the profile-guided compilation.

### 2.3 Dynamically Loaded Loop Caches

To eliminate performance degradation and the need for tag comparisons a loop cache was proposed in [20] which was used in Motorola's M\*CORE embedded processor. The proposed loop cache is a small instruction buffer that is tightly integrated with the processor and 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 as shown in Figure 1(b). A state diagram of the corresponding loop cache controller is shown in Figure 2. The loop cache controller is responsible for filling the loop cache when

Figure 2: Loop Cache Controller for Dynamically Loaded Loop Caches.



detecting a simple loop – defined as any short backwards branch instruction (sbb), activating fetches from the loop cache upon subsequent iterations of the loop, and exiting the loop cache upon a control of flow change (cof) which is not the original sbb detected at the end of the loop. Thus, at the end of the first iteration of a loop, a short backwards branch is detected transitioning the control from idle to fill. Then, during the second iteration, as instructions are fetched from memory, the loop cache is also filled. Finally, starting with the third iteration, the loop cache controller transitions from the fill to active state, thereby fetching 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*.

The aforementioned caching strategy is a cold-fill strategy. The loop cache is not filled until the second iteration and not active until the third iteration of the loop. [21] also discusses a warm-fill strategy in which the loop cache controller is reduced to two states – filling or active. Thus after the first iteration of the loop is taken, some or all of the loop is already located in the loop cache. The loop cache can then be accessed on the second iteration of the loop.

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 [21] to a flexible dynamically loaded loop cache design, referred as the *flexible dynamic loop cache*. In this design, if a loop is larger than the loop cache, depending on how the loop cache is filled, the instructions located in either the upper portion (cold-fill strategy) or lower portion (warm fill strategy) of the loop fills the loop cache until the cache is full.

Furthermore, in a dynamically loaded loop cache, cof such as internal branches within loops, multiple backwards branches to the same starting point in a loop, nested loops, and subroutines pose problems. In each of the situations, the cof would cause the filling of or execution from the loop cache to be aborted.

Architecture variation for the dynamically loaded loop cache thus not only included loop cache size but also original versus flexible loop support and warm-fill

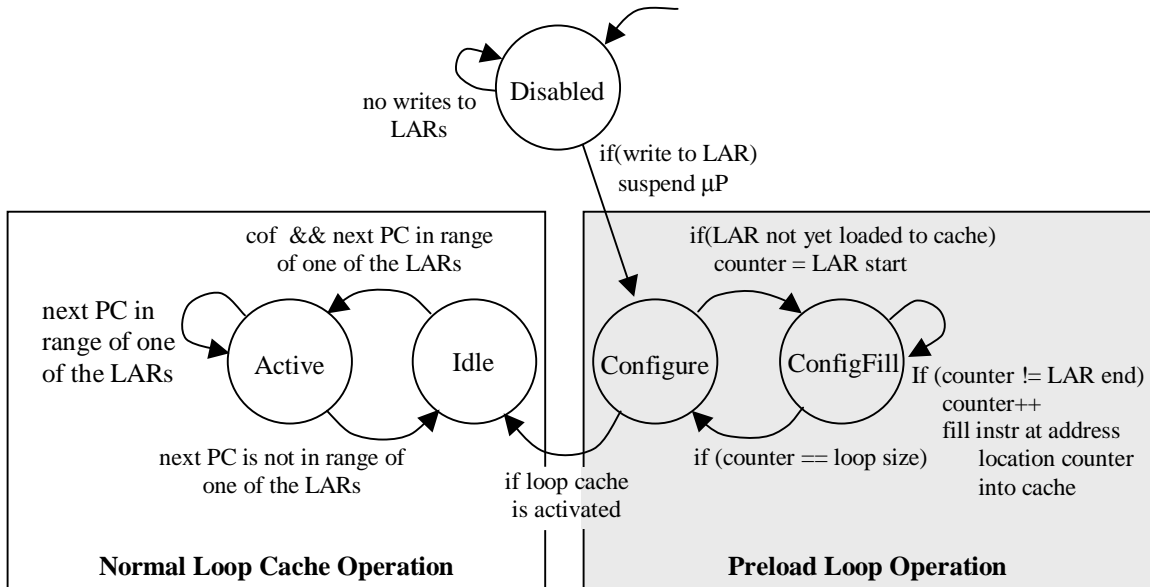
versus cold-fill strategies. In this work, we consider original and flexible dynamically loaded loop caches of various sizes. However, these loop caches solely use the cold-fill strategy.

## **2.4 Preloaded Loop Caches**

A preloaded loop cache, shown in Figure 1(c), was proposed in [11][12] to overcome cof limitations. Using profiling information gathered for a particular application, the loops that comprised the largest percentage of execution time and fit within the loop cache are selected and preloaded into the loop cache during system reset, along with extra bits indicating whether control of flow changes exit the loop. In addition, loop address registers (LAR) denoting the starting and ending addresses of the preloaded loops are also loaded to indicate which loops are located in the loop cache. After this initialization, the contents of the loop cache do not change for the duration of program execution. By preloading, situations with cof could be handled.

Figure 3 shows the loop cache controller for basic preloaded loop caches. The loop cache controller contains two phases – preload loop operation, which occurs before the processor starts execution, and normal loop cache operation, which occurs during processor execution. During the preload loop operation, loops that are selected for the preloaded loop cache are loaded along with the start and end addresses. Once this is done the loop cache can be activated transitioning into the normal loop cache operations. Once loops are selected and loaded into the preloaded loop cache, the contents of the loop cache remain static during processor execution. In normal loop cache operation, the loop cache controller checks for a loop address whenever an sbb is executed. If the loop

**Figure 3:** Loop Cache Controller for Basic Preloaded Loop Caches.



address is within the bounds of one of the LARs the controller transitions to the active state, thereby fetching instructions from the loop cache instead of memory. Each subsequent instruction is compared to the LARs to ensure it is within the bounds of the preloaded loops. If an instruction is encountered which is not within the bounds of the LARs the loop cache controller transitions back to the idle state. Since loops are preloaded, loop cache fetching can begin on the second rather than third loop iteration as in the cold-fill dynamically loaded loop cache. 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 the first iteration. This approach is referred to as the *preloaded loop cache (sa)*.

Because loops are preloaded into the loop cache 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 variations considered in this work for preloaded loop caches include cache size, number of supported loops, and loop address checking strategy.

## **2.5 Loop Cache Selection**

For each of the loop caching styles, a variety of parameters exists, each with their corresponding tradeoffs. For example, different sizes are possible – larger loop caches can hold bigger loops thereby increase cache hits but at the expense of more power per access. Additionally, preloaded loop caches can support different numbers of loops. The more loops supported the better the hit rate but at the expense of more control logic. 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 contained within the application.

# Chapter 3

## Loop Cache Simulation

### 3.1 Overview

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 [23], shown in Table 1. 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 (sbb)* - cache sizes ranging from 8 to 1024 entries, with 2 to 6 loop address registers
- *preloaded loop cache (sa)* - 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

**Table 1: Benchmark Descriptions.**

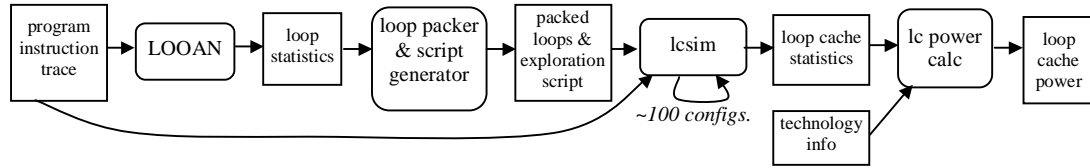
| Benchmarks | Size of Assembly (bytes) | Description              |
|------------|--------------------------|--------------------------|
| adpcm      | 1912                     | Voice Encoding           |
| blit       | 1045                     | Graphics Application     |
| compress   | 1870                     | Data Compression Program |
| crc        | 1062                     | Cyclic Redundancy Check  |
| des        | 1531                     | Data Encryption Standard |
| engine     | 1110                     | Engine Controller        |
| fir        | 1058                     | FIR Filtering            |
| g3fax      | 1096                     | Group Three Fax Decode   |
| jpeg       | 1492                     | JPEG Compression         |
| summin     | 1036                     | Handwriting Recognition  |
| ucbqsort   | 1212                     | U.C.B Quick Sort         |
| v42        | 1599                     | Modem Encoding/Decoding  |

of loop address registers available indicates the maximum number of loops that can be preloaded into the loop cache.

### 3.2 Loop Cache Simulation Methodology

We developed a suite of tools, shown in Figure 4, 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 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 [35]. 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 executed 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

**Figure 4:** Loop Cache Simulation-Based Tool Chain.



generates a script that will explore each possible loop cache configurations with associated loop packings for the preloaded caches. We developed a loop cache simulator, called `lcsim`, which 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. The `lcsim` tool generates the 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

**Table 2:** 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 |

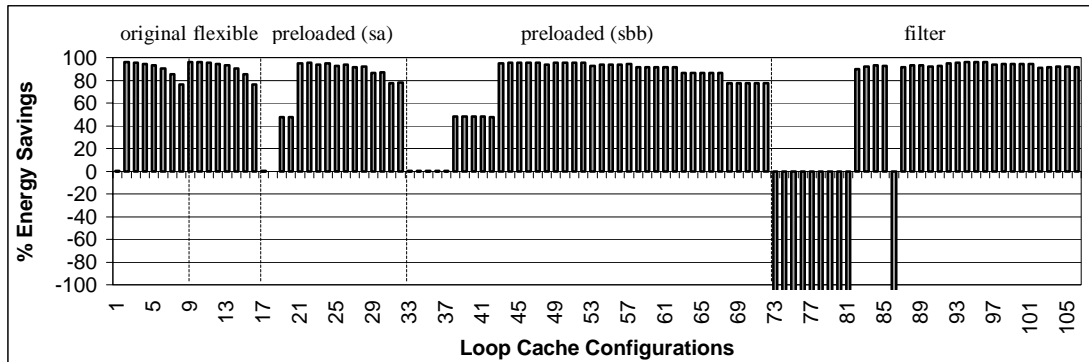
technology. A designer interested in determining how each cache configuration performs for a specific technology can simply set the performance value to the technology of interest.

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.).

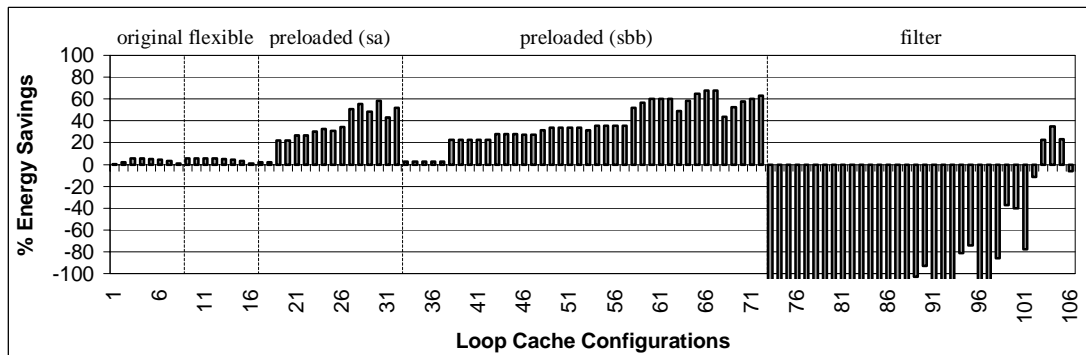
### **3.3 Simulation Results**

To facilitate plotting of so many configurations, we associate each configuration with a numerical code, with Table 2 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 caches 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 the value of 73. Furthermore, 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.

**Figure 5: Energy Savings for *blit* Given Various Cache Configurations.**



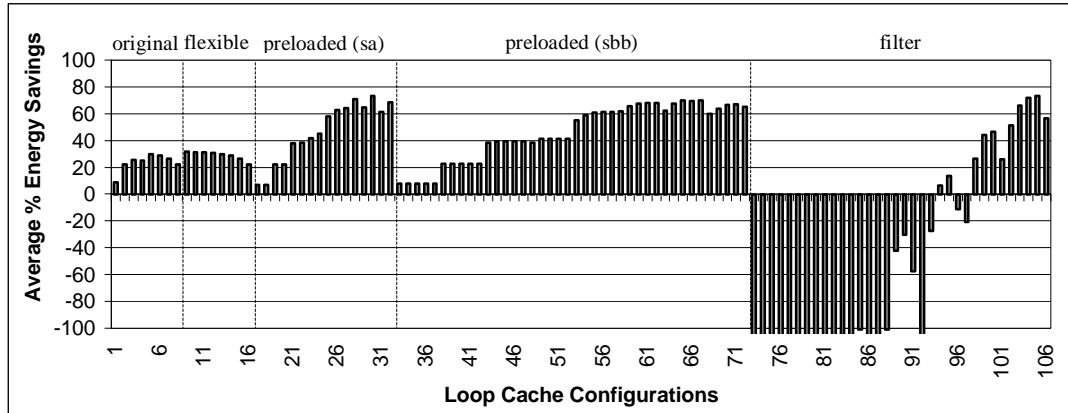
**Figure 6: Energy Savings for *v42* Given Various Cache Configurations.**



We measured the average energy savings for each cache configuration for each benchmark compared to the energy consumed by a configuration that contains no filter/loop cache. Figure 5 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 6 shows the savings for each cache configuration for the *v42* benchmark (a modem encoding/decoding application). Although the dynamic caches performed well

**Figure 7: Average Energy Savings for Various Cache Configurations.**



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 accept 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 of the benchmarks considered.

Figure 7 shows the average savings of each cache configuration over all benchmarks. Cache configuration 30, a preloaded 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 8:** Average Performance Penalty for Filter Cache.

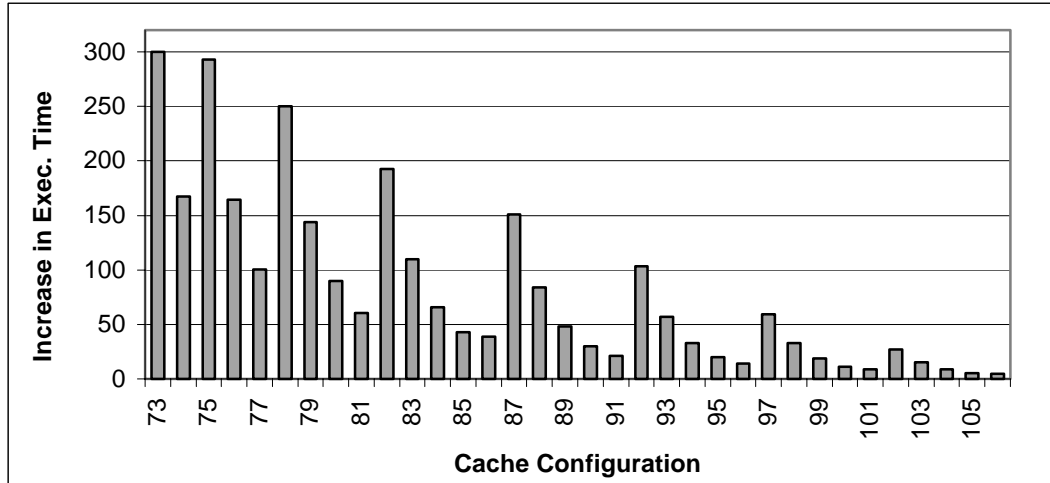
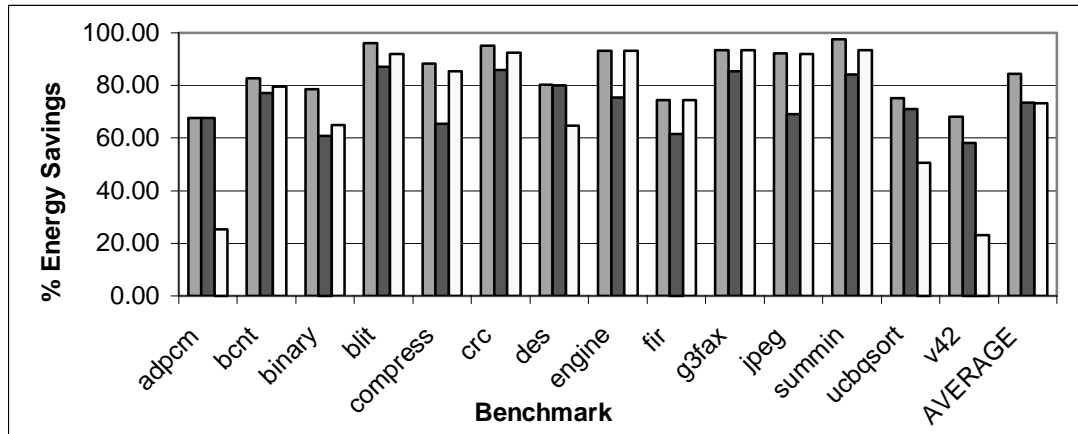


Figure 8 shows the average increase in execution time given the various filter cache configurations (though remember that degradation for certain applications was larger).

If the memory architecture could not be customized to a particular application, as is the case for pre-fabricated microprocessors or even cores 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 particular 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 9 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* have an increased savings of 23% for each over configuration 30. In addition, for *adpcm*, *ucbqsort*, and *v42*, the best

**Figure 9:** Best Configuration for a Benchmark (left bar) vs. Best Average Configurations: Configuration 30 (middle bar) and Configuration 105 (right bar).



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

# Chapter 4

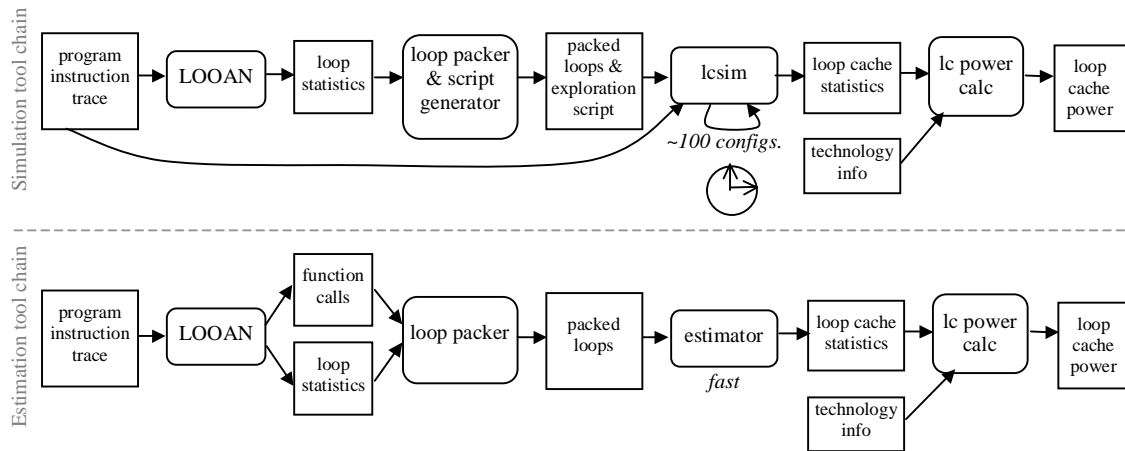
## Loop Cache Estimation

### 4.1 Overview

In the previous section, we demonstrated that a significant instruction fetch energy can be achieved through loop cache customization on a per application basis. However, the previous 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 [19]), 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. We sought to develop a faster method than loop cache simulation for exploring the loop cache configuration space, specifically an estimation based methodology. In the loop cache estimation methodology, we evaluated a number of cache architecture configurations on a set of Powerstone benchmarks [23]. 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).

**Figure 10: Loop Cache Simulation- and Estimation-Based Tool Chain.**



- *preloaded loop cache (sa)* – cache sizes ranging from 8 to 1024 entries (by powers of 2), with either 2 or 3 loop address registers
- *preloaded loop cache(sbb)* – cache sizes ranging from 8 to 1024 entries (by powers of 2), with 2-6 loop address registers

For the estimation-based methodology, we omitted the filter cache results as accurately determining the behavior of the filter cache is difficult to do with the current information provided by the tool chain. Augmenting the current tool chain to provide additional information to estimate the filter cache behavior more closely is left as future work.

## 4.2 Loop Cache Estimation Methodology

Our estimation methodology is similar to the simulation methodology previously discussed, as shown in Figure 10. 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. Note that *lcsim* follows the paradigm of traditional cache simulators, like Dinero [7]. Thus, we could look into traditional cache simulation speedup methods, like examining multiple configurations per pass [32] or compacting the trace size using statistical methods [36]. 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. Finally, the *lc power calc* tool reads these statistics, as well as technology parameters, and the loop cache power information, to generate loop cache power data. As in the estimation methodology, we calculated power and energy based on the switching activity of each operation utilizing VHDL models and the relative capacitance of various components.

#### *4.2.1 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 triggered 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.

#### *4.2.2 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 than 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.

#### *4.2.3 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 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.

#### *4.2.4 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.

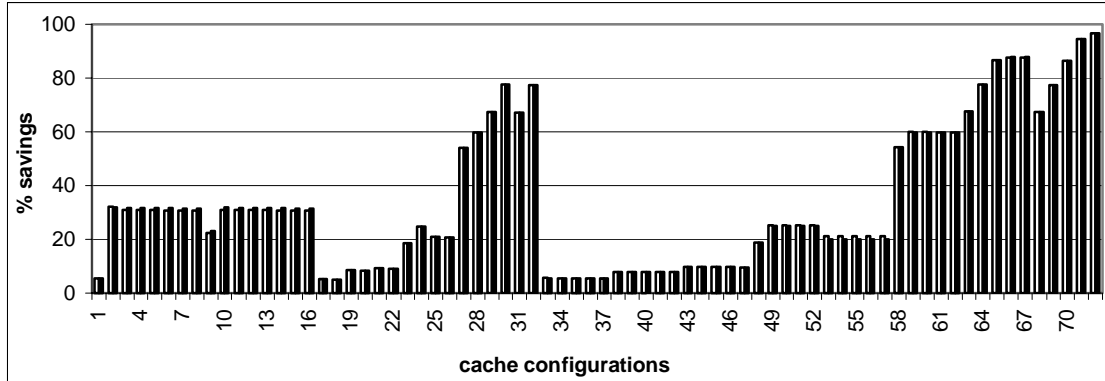
The number of detect operations corresponds to the number of control of flows in the given program. There exists a control of flow at the end of a loop, when calling a function, and when returning from a function. Thus for every loop we find, we add a detect operation to represent the sbb at the end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

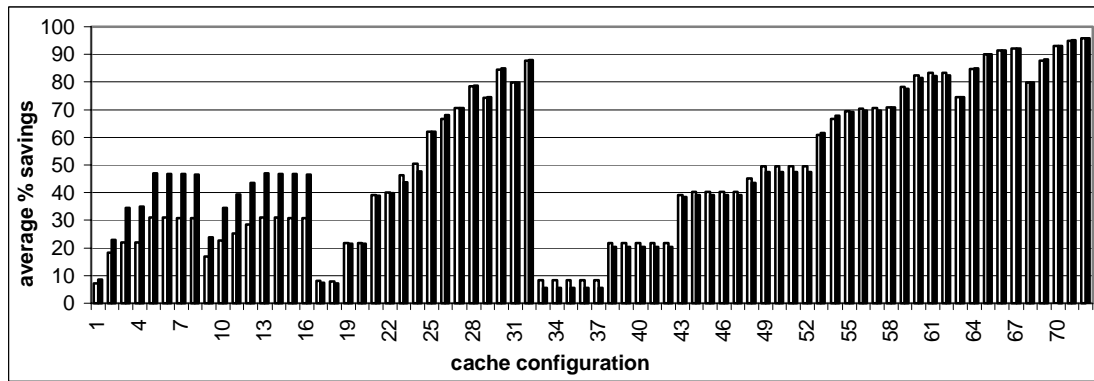
### **4.3 Estimation Results**

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

**Figure 11:** Percent Savings For Various Cache Configurations, Savings Using Simulation (left) Versus Savings Using Estimation (right) For The *jpeg* Benchmark.



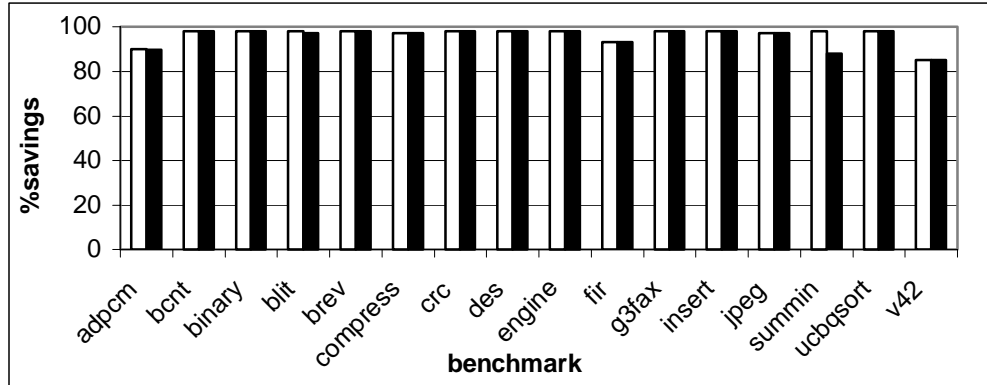
**Figure 12:** Percent Savings For Various Cache Configurations, Savings Using Simulation (left) versus Savings Using Estimation (right) For Averages Over All Benchmarks.



facilitate plotting of so many configurations, we associate each configuration with the same numerical code, shown in Table 2.

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 11 compares the reported power savings

**Figure 13:** Power Savings Using Cache Configurations from Simulation Approach (left) versus Estimation Approach (right).



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 for the *jpeg* benchmark differs 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 12. 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.

**Table 3: Simulation Times versus Estimation Times (in seconds).**

| Benchmark | Number Instr Executed | Simulation Tool Chain |            |         |               |                       | Estimation Tool Chain |           |               |                       | Speedup       |
|-----------|-----------------------|-----------------------|------------|---------|---------------|-----------------------|-----------------------|-----------|---------------|-----------------------|---------------|
|           |                       | LOOAN                 | Script Gen | lcsim   | lc Power Calc | Total Simulation Time | LOOAN                 | Estimator | lc Power Calc | Total Estimation Time |               |
| adpcm     | 63891                 | 0.31                  | 0.01       | 32.15   | 0.01          | 32.48                 | 0.31                  | 0.16      | 0.01          | 0.48                  | <b>67.67</b>  |
| bcnt      | 1938                  | 0.01                  | 0.01       | 1.17    | 0.01          | 1.20                  | 0.02                  | 0.06      | 0.01          | 0.09                  | <b>13.33</b>  |
| binary    | 816                   | 0.01                  | 0.01       | 0.87    | 0.01          | 0.90                  | 0.01                  | 0.08      | 0.01          | 0.1                   | <b>9.00</b>   |
| blit      | 22845                 | 0.07                  | 0.01       | 7.26    | 0.01          | 7.35                  | 0.07                  | 0.06      | 0.01          | 0.14                  | <b>52.50</b>  |
| brev      | 2377                  | 0.01                  | 0.01       | 1.20    | 0.01          | 1.23                  | 0.01                  | 0.06      | 0.01          | 0.08                  | <b>15.38</b>  |
| compress  | 138573                | 0.85                  | 0.01       | 82.50   | 0.01          | 83.37                 | 0.85                  | 0.14      | 0.01          | 1                     | <b>83.37</b>  |
| crc       | 37650                 | 0.15                  | 0.01       | 16.03   | 0.01          | 16.20                 | 0.15                  | 0.07      | 0.01          | 0.23                  | <b>70.43</b>  |
| des       | 122214                | 0.44                  | 0.02       | 45.28   | 0.01          | 45.75                 | 0.44                  | 0.07      | 0.01          | 0.52                  | <b>87.98</b>  |
| engine    | 410607                | 2.12                  | 0.02       | 214.99  | 0.01          | 217.14                | 2.12                  | 0.08      | 0.01          | 2.21                  | <b>98.25</b>  |
| fir       | 16211                 | 0.07                  | 0.02       | 7.60    | 0.01          | 7.70                  | 0.07                  | 0.09      | 0.01          | 0.17                  | <b>45.29</b>  |
| g3fax     | 1128023               | 3.54                  | 0.02       | 385.44  | 0.01          | 389.01                | 3.54                  | 0.09      | 0.01          | 3.64                  | <b>106.87</b> |
| insert    | 1942                  | 0.01                  | 0.01       | 1.18    | 0.01          | 1.21                  | 0.01                  | 0.06      | 0.01          | 0.08                  | <b>15.13</b>  |
| jpeg      | 4594721               | 17.57                 | 0.01       | 1837.28 | 0.01          | 1854.87               | 17.57                 | 0.12      | 0.01          | 17.7                  | <b>104.79</b> |
| summin    | 1909787               | 11.42                 | 0.01       | 903.73  | 0.01          | 915.17                | 8.25                  | 0.09      | 0.01          | 8.35                  | <b>109.60</b> |
| ucbqsort  | 219978                | 0.93                  | 0.01       | 82.62   | 0.01          | 83.57                 | 0.89                  | 0.09      | 0.01          | 0.99                  | <b>84.41</b>  |
| v42       | 2442551               | 12.07                 | 0.01       | 1252.48 | 0.01          | 1264.57               | 12.27                 | 0.12      | 0.01          | 12.4                  | <b>101.98</b> |

AVERAGE    **66.62**

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 13 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 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 3 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 3, it can be seen that the majority of time for the simulation-based method is spent running *lcsim*, the loop cache simulator. 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.

# Chapter 5

## Conclusions

### 5.1 Summary

Incorporating a single tiny instruction cache can result in instruction fetch energy savings but leaves room for significant improvement in many cases. By considering a variety of filter/loop cache architectures and variations, we obtained an average additional savings of 10%, with savings as high as 40% in several cases. 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 implemented a simulation-based tool that finds the loop cache configuration yielding the best energy savings for a given program. Although the simulation-based environment is automated, this approach requires several hours to find the best configuration due to rerunning the cache simulator for every configuration. We have also implemented a fast estimation-based method that obtains nearly the same savings but in seconds rather than hours.

### 5.2 Future Work

Our future work includes investigating additional loop cache structures such as warm-fill dynamically loaded loop caches, hybrid dynamic/preloaded loop caches, and estimation approaches for filter caches. Additionally, we plan to examine the impacts loop caches have on the performance of first level instruction caches, as well as extend our exploration approach to assist designers in choosing the right combination of loop and instruction caches. Furthermore, we plan to investigate different configuration exploration strategies such as utilizing estimation to narrow down possible choices followed by simulation to obtain accurate instruction fetch energy saving values. Lastly, we plan to consider a wider variety of benchmarks to see how such benchmarks effect the exploration environment as well as how they effect the cache configurations selected.

# References

- [1] Aditya, S., B. Rau, V. Kathail. Automatic Architectural Synthesis of VLIW and EPIC Processors. International Symposium on System Synthesis, 1999.
- [2] Bahar, R., G. Albera, S. Manne. Power and Performance Tradeoffs using Various Caching Strategies. International Symposium on Low Power Electronics and Design, 1998.
- [3] Bellas, N., I. Hajj, C. Polychronopoulos, G. Stamoulis. Energy and Performance Improvements in Microprocessor Design 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, 1999.
- [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] Dutt, N. Memory Organization and Exploration for Embedded Systems-on-Silicon. International Conference on VLSI and CAD, 1997.
- [7] Elder, J., M.D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. <http://www.cs.wisc.edu/~markhill/DineroIV>.
- [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, 1996.
- [10] Gonzales, R. Xtensa: A Configurable and Extensible Processor. International Symposium on Microarchitecture, 2000.

- [11] Gordon-Ross, A., S. Cotterell, F. Vahid. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. *Computer Architecture Letters*, Volume 1, 2002.
- [12] Gordon-Ross, A., S. Cotterell, F. Vahid. Tiny Instruction Caches For Low Power Embedded Systems. *IEEE Transactions on Embedded Computing Systems*, to appear.
- [13] Kavvadias, N., A. Chatzigeorgiou, N. Zervas, S. Nikolaidis. Memory Hierarchy Exploration For Low Power Architectures in Embedded Multimedia Applications. *International Conference on Image Processing*, 2001.
- [14] 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.
- [15] Kim, S., N. Vijaykrishnan, M. Kandemir, A. Sivasubramaniam, M. Irwin, E. Geethanjali. Power-aware Partitioned Cache Architectures. *International Symposium on Low Power Electronics and Design*, 2001.
- [16] Kin, J., M. Gupta, W. Magione-Smith. The Filter Cache: An Energy Efficient Memory Structure. *International Symposium on Microarchitecture*, 1997.
- [17] Kirovski, D., J. Kin, W. Mangione-Smith. Procedure Based Program Compression. *International Symposium on Microarchitecture*, 1997.
- [18] Ko, U., P. Balsara. Characterization and Design of A Low-Power, High-Performance Cache Architecture. *International Symposium on VLSI Technology, Systems, and Applications*, 1995.
- [19] Lee, C., M. Potkonjak, W. Magione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. *International Symposium on Microarchitecture*, 1997.
- [20] 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*, 1999.
- [21] Lee, L., B. Moyer, J. Arends. Low-Cost Embedded Program Loop Caching – Revisited. *University of Michigan Technical Report CSE-TR-411-99*, 1999.
- [22] Lekatsas, H., J. Henkel, W. Wolf. Code Compression for Low Power Embedded System Design. *Design Automation Conference*, 2000.

- [23] 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.
- [24] Mehta, H., R. Owens, M. Irwin. Some Issues in Gray Code Addressing. IEEE GLS-VLSI-96, March 1996.
- [25] 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.
- [26] Panda, P., N. Dutt, A. Nicolau. Architectural Exploration and Optimization of Local Memory in Embedded Systems. International Symposium on System Synthesis, 1997.
- [27] 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, Volume 29, Number 3, pp. 167-178, 2001.
- [28] 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.
- [29] Stan, M., W. Burlison. Bus Invert for Low Power I/O. IEEE Transactions on VLSI, 1995.
- [30] Su, C., C. Tsui, A. Despain. Cache Design Trade-offs for Power and Performance Optimization: A Case Study. International Symposium Low Power Design, 1995.
- [31] Su, C., C. Tsui, A. Despain. Saving Power in the Control Path of Embedded Processors. IEEE Test and Design of Computers, Volume 11, Number 4, 1994.
- [32] 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.
- [33] Vahid, F., T. Givargis, Platform Tuning for Embedded Systems Design. IEEE Computer, Vol. 34, No 3, 2001.
- [34] Villarreal, J., D. Suresh, G. Stitt, F. Vahid, and W. Najjar. Improving Software Performance with Configurable Logic. Design Automation of Embedded System, 2002.

- [35] 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.
- [36] Wu, Z, and W. Wolf. Iterative Cache Simulation of Embedded CPUs with Trace Stripping. International Conference on Hardware/Software Co-Design, 1999.