Dynamic Loop Caching Meets Preloaded Loop Caching – A Hybrid Approach

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

Abstract

Dynamically-loaded tagless loop caching reduces instruction fetch power for embedded software with small loops, but only supports simple loops without taken branches. Preloaded tagless loop caching supports complex loops with branches and thus can reduce power further, but has a limit on the total number of instructions cached. We show that each does well on particular benchmarks, but neither is best across all of those benchmarks. We present a new hybrid loop cache that only preloads the complex loops, while dynamically loading other loops, thus achieving the strengths of each approach. We demonstrate better power savings than either previous approach alone.

Keywords: Loop cache, low power, embedded systems, architecture.

1. Introduction

Instruction fetch power may consume nearly 50% of an embedded microprocessor system's power [1][11]. Reducing instruction fetch power can thus result in significant system power savings, important in embedded systems to increase battery lifetime or reduce cooling requirements.

Several methods focus on reducing instruction fetch power such as bus encoding [1] and program compression [6]. A complementary approach capitalizes on embedded system software that tends to spend much time in small loops by using a tiny and hence low-power cache to store such loops. One such method [1] uses a simple but clever controller that dynamically detects and loads only loops, and that conservatively detects when the loop is exited, thus eliminating the need for tag comparisons. Such a dynamically loaded loop cache method is transparent to the designer, but is limited to supporting loops without taken branches or subroutine calls. In [5], we proposed another tagless method that overcomes this limitation by observing that embedded programs are typically fixed, and hence the most frequent loops can be preloaded into the loop cache. The preloaded loop cache method can support loops that are more complex, and thus provides greater power savings, but has a limit as to how many loops can be preloaded. Each approach has been shown to excel in different situations.

We present the design of a hybrid dynamically-loaded/preloaded loop cache that gains the advantages of both methods. The hybrid loop cache can operate as a dynamically loaded cache only, providing transparency in cases where designers do not wish to perform preloading or if the application running will not be fixed. The hybrid cache can also be preloaded with only those loops that would not work well if dynamically loaded, thus avoiding the preloading of dynamically loadable loops and hence effectively increasing the size of the preloaded loop storage. We demonstrate power improvements on several benchmarks.

2. Related Work

An extremely small cache of 32 to 64 words, tightly integrated with a processor, has very small access power compared to accessing a standard first level cache or a standard on-chip or off-chip program memory. Kin et al [7] first proposed such a small cache, called a *filter cache*, to reduce power, at the expense of reduced performance due to frequent filter cache misses. Bellas et al [2] used a profile-guided compiler to map frequently executed loops to a special address range, and discussed architecture extensions that would only load items in that range into the filter cache, thus reducing (but not eliminating) misses. We refer to this approach as a *selective filter cache*. Two approaches were further developed that eliminate the power-costly tag comparisons of the above approaches, which we now discuss.

2.1 Dynamically Loaded Tagless Loop Caching

The dynamically loaded loop cache, designed at Motorola [1], exploits the fact that many loops are small and are formed by the last loop instruction jumping back to the loop start. This instruction is denoted as a *short backwards branch* (sbb) instruction and can be any program-counter relative branch instruction, i.e., an sbb is not a special instruction. During program execution, the instructions being fetched from instruction memory are monitored. A taken sbb triggers the loop cache controller to begin filling the loop cache. During the next loop iteration, the instructions fetched from instruction memory are fed to both the processor and the loop cache. On the third loop iteration, the instruction memory is bypassed and instructions are instead fetched from the loop cache. If a loop does not completely fit in the loop cache, only the first instructions are cached.

Fetching continues from the loop cache until the triggering sbb is not taken. Loop cache filling or fetching also terminates if there is a *control of flow change (cof)*, namely a taken jump. A cof causes termination because a cof during filling prevents the entire loop from being loaded into the loop cache, and a cof during loop cache fetching could cause execution to leave the loop cache.

Unlike filter caches, a dynamically loaded loop cache does not suffer any misses and hence imposes no performance overhead. It involves no tag comparisons, resulting in even less power per access. Furthermore, it is completely transparent to a designer, requiring no special profile-guided compilation step.

However, the dynamically loaded loop cache cannot cache loops with cofs, even cofs resulting from simple if statements that would never cause execution to leave the loop cache, nor

cofs caused by calls from the loop to simple subroutines. In some cases, it may increase power due to extensive thrashing, caused by particular loop nestings.

2.2 Preloaded Tagless Loop Caching

We introduced the preloaded tagless loop cache in [5] to increase the percentage of frequently executed code that could be captured in the loop cache. Based on the observation that embedded system products typically have a fixed application, meaning that the program running on the microprocessor will not change during the lifetime of the product (e.g., a digital camera), the critical regions of code could be determined ahead of time and loaded into a cache whose contents would not change. The cache could now capture critical regions of code including loops with cofs, subroutines, and nested loops. By keeping the starting and ending addresses of each critical region and the starting location of the instructions in the cache, any number of regions could be stored. Like the dynamically loaded loop cache, the cache would be tagless, small, tightly integrated with the microprocessor and provide low power instruction fetching. Several methods for preloading the loop cache are possible; most are carried out during system reset.

Fetching from the preloaded loop cache begins when a cof causes the next instruction to be within the range of one of the loops in the loop cache. Since the loop cache is prefilled, cofs do not pose a fill problem. During loop cache fetching, a few pre-determined *exit bits* associated with each instruction in the loop cache are used to determine if a cof exits the current loop. Static code analysis sets these bits, which are then preloaded with the loop. In rare cases where static analysis cannot determine the target branch address, such as the case of an indirect jump, the exit bits are conservatively set to indicate a loop exit.

The preloaded tagless loop cache has the added benefit of supporting subroutines and more loops than the dynamic approach. However, the loop cache size is fixed and so can only hold a limited number of instructions. In contrast, the dynamic loop cache is continually refilled. The preloaded method also requires the designer to perform the extra design step necessary for profiling the code and filling the loop cache.

3. Hybrid Loop Caching

Table 1 provides data on the two tagless loop caching approaches for several benchmarks. Notice that in some cases dynamic is best, while in others preloaded is best – and the power difference between the two can be large. What is needed is a loop caching scheme that achieves the benefits of both approaches.

We thus designed a hybrid loop cache, consisting of a main loop cache loaded either dynamically or from a second level of preloaded storage. As with the earlier approaches, the main loop cache is very small and tightly integrated with the microprocessor. The second level of preloaded storage is not as power critical because accesses to it will be infrequent – thus, the second level can be as big as size constraints allow.

3.1 Architecture

The hybrid loop cache architecture can be seen in Figure 1. Its main components consist of two levels of storage, two controllers, a loop match memory (comparators), and loop address registers (LARs). The first level of storage is the main loop cache where instruction fetches occur. The second level of storage contains the loops that have been determined as critical regions of code, have been preanalyzed to determine exit bits, and have been preloaded during system reset. The two controllers are responsible for controlling the operation of the loop cache and will be discussed below. The loop address registers hold the starting and ending addresses of the loops stored in the second level storage and the starting position of the loop in the second level storage.

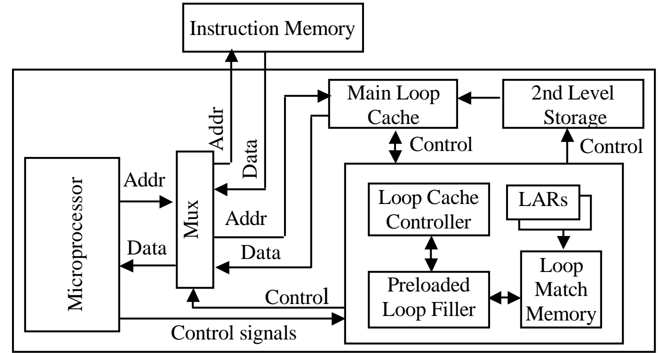


Figure 1: Basic architecture of the hybrid loop cache.

3.2 Operation

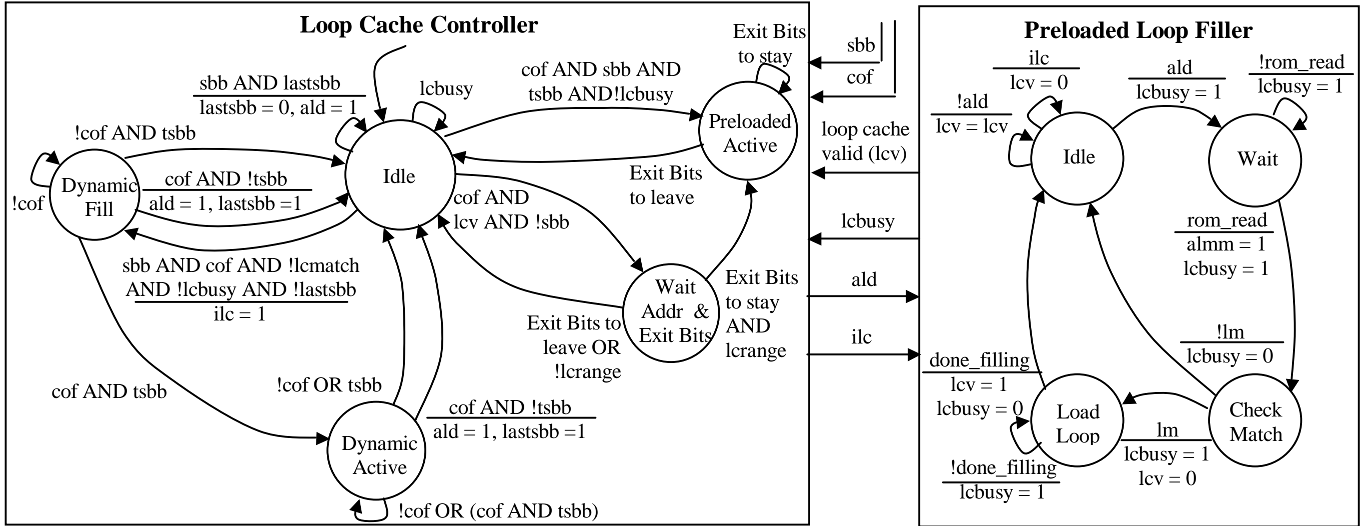
The hybrid loop cache consists of two controllers: the loop cache controller and the preloaded loop filler. The loop cache controller is the master controller and is responsible for dynamically filling the main loop cache, determining when to check for a preloaded loop, and for orchestrating instruction fetches. This controller can operate in either a dynamic or a preloaded caching mode. The preloaded loop filler is responsible for both detecting the execution of loops that have been preanalyzed and for filling the main loop cache with these loops. It is only activated when the loop cache controller requests a loop to be filled from the second level storage. Both state machines can be seen in Figure 2.

An important feature of the hybrid loop cache is its ability to function like a dynamic loop cache if the designer does not wish to take the extra step to preanalyze the application. If the designer wishes to bypass the preanalysis step and forgo the loading of the second level of storage, the loop cache controller will act as a dynamically loaded loop cache – completely transparent to the designer. This also means that the application running on the microprocessor does not need to be fixed as it was in the preloaded loop cache. In that case, only the dynamic portion of the design would function. The same architecture can be used for both fixed and changing application systems.

3.2.1 Detecting Loops / Filling the Main Loop Cache

Filling the main loop cache can be done dynamically while instructions are fetched from main memory, or from the preloaded second level of storage. Determining where the main loop cache will be filled from is the responsibility of the loop cache controller.

During the *Idle* state, memory fetches are serviced by the main memory and the main loop cache is disabled. To reduce the number of comparisons resulting from checking for preloaded loops, dynamic loading takes precedence, meaning that during the *Idle* state, the loop cache controller transitions to *Dynamic Fill* upon execution of a *taken sbb* (tsbb). At this point, new



* Unless specified, all other signals are deasserted in each state. , ** Abbreviations are described in the text.

Figure 2: State machines for the hybrid loop cache controller and preloaded loop filler and their interconnect logic.

instructions will be loaded into the main loop cache and the loop cache contents are *invalidated* (ilc).

The *Dynamic Fill* state functions the same as it does for the dynamic loop cache. During this state, if there is a cof, the loop can no longer be assumed as a candidate for the dynamic caching method and the state machine transitions to *Idle*. At this point, the preloaded loop filler is activated to determine if the loop is preloaded in the second level of storage.

The *activate loop detection logic* (ald) signal is what triggers the preloaded loop filler to transition from the *Idle* state to the *Wait* state to wait for the next instruction read. The next instruction address is compared with the LARs by *activating the loop match memory* (almm). If there is a match (lm) the loop is loaded into the main loop cache. If the loop calls any simple subroutines, one may be loaded along with the loop. The *loop cache busy* (lcbusy) signal is asserted during this time. This provides mutual exclusive filling of the loop cache.

3.2.2 Fetching From the Main Loop Cache

Fetching from the main loop cache in the hybrid approach, whether it be a dynamically loaded or a preloaded loop, is the same as it is for the separate dynamic and preloaded loop caches. The hybrid approach has two active states, one for each fetching method and depending on the active state, the main loop cache is accessed appropriately. The only differences are in the conditions for transitioning to the *Preloaded Active* state.

There are two ways to transition to the *Preloaded Active* state. The first way happens when the triggering sbb of a preloaded loop is taken. For this, execution transitions immediately to the *Preloaded Active* state and while in this state, operation is the same as it is for the preloaded loop cache.

The second way is via the *Wait for Address and Exit Bits* state. The loop cache controller transitions to this state when there is a cof and lcv is asserted. This state is necessary to resume fetching from the main loop cache after a return from a subroutine that has not been preloaded. The next instruction address must fall within the range of the loaded loop and the exit bits must indicate that loop cache fetching will continue.

4. Experiments

To examine the power effectiveness of our hybrid loop caching method, we ran tests on ten benchmarks from the Powerstone benchmark suite [10] running on a 32-bit MIPS microprocessor and three benchmarks from the MediaBench suite [8] running on SimpleScaler [3]. We ran each benchmark on an instruction set simulator to obtain an address trace of the program execution. We developed a loop cache simulator (lcsim) that reads in the instruction trace and outputs detailed statistics including the number of instruction memory fetches and loop cache operations (i.e. detect (address comparison), fill and fetch). We also modeled each loop cache controller in synthesizable VHDL and synthesized the controllers using Synopsys Design Compiler [12]. We simulated the loop cache controllers to determine the switching activity per loop cache operation.

Based on data in [1], we used a ratio of 100 to 1 for the power consumption for an access to instruction memory verses a loop cache of size 16. The power consumed by the loop cache was increased by 1.5 for each doubling of the size of the loop cache. Power of the internal nets of the loop cache controller were 1/8 of the bus wires to a loop cache of size 16. We assume that the time to access main memory and the loop cache are the same.

Our results are represented as a percentage of power savings compared to a design with no loop cache. For power consumption, we only consider the power consumed due to accesses to the instruction memory which can be nearly 50% of the total power consumption [1][11].

Our test results include running the benchmarks through the dynamically loaded, preloaded and hybrid loop caching schemes. We tested a hybrid main loop cache and second level storage sizes of 16 to 128 entries. For each test, we compared the hybrid results to a dynamically loaded and preloaded loop cache with a loop cache size equal to the size of the hybrid main loop cache.

Table 1 shows the results for our tests comparing a hybrid cache with a main loop cache size of 32 and a second level storage of size 128, with a dynamic and preloaded cache each of size 32. In the dynamic and the preloaded columns, the approach

Table 1: Percentage of power savings for instruction fetching. The bold entries indicate the better of dynamic or preloaded. Note that hybrid is as good as either in most cases.

	Dynamic 32	Preloaded 32	Hybrid 32/128
adpcm*	**0%**	-1%	35%
blit	**95%**	94%	95%
compress	9%	9%	17%
crc	-1%	**63%**	99%
des	**23%**	3%	38%
engine	20%	**26%**	19%
epic*	0%	**47%**	19%
fir	29%	**36%**	56%
g3fax	59%	**61%**	96%
jpeg*	2%	**12%**	41%
summin	**54%**	46%	76%
ucbqsort	2%	**34%**	49%
v42	5%	**25%**	22%
AVG	23%	35%	51%

* MediaBench

with the greatest savings is bold. Of the thirteen benchmarks, the hybrid approach performed the best in nine of them and preformed equally as well in ine. For the remaining three, the hybrid approach performed better or as well as a strictly dynamic approach but was outperformed by the preloaded cache.

Figure 3 examines the effects of deep sub micron technologies with increasing bus capacitances. The ratios represent the increase in power consumed by the loop cache for each doubling of its size, starting at size 16. We looked at a hybrid loop cache with a main loop cache size of 32 and a second level storage size of 1024 and preloaded loop cache of size 1024. As the power ratio increases, the power savings for the hybrid loop cache remain relatively constant, only decreasing slightly because of the small number of accesses to the large second level cache. The very small main loop cache ensures the constant power savings. However, because the preloaded loop cache executes entirely from the very large loop cache, it suffers heavily from increased bus capacitances.

A very important trait of the hybrid loop cache is its resilience when it is used as simply a dynamically loaded loop cache. To test this, we compared the results of a hybrid loop cache with no preloaded loops to the results of a strictly dynamically loaded loop cache in Table 2. In all cases, the hybrid loop cache fell less than 1% short of the dynamically loaded loop cache. If the designer does not wish to preanalyze the application, the power savings of a dynamically loaded loop cache can still be achieved transparently to the designer.

5. Conclusions

A hybrid loop cache can reduce embedded system software instruction fetch power by nearly 50%. It can be used transparently as a dynamically-loaded loop cache if the designer does not wish to take the preloading step and will provide the same power savings as a dynamically loaded loop cache. If

Table 2: Power savings for instruction fetching of a dynamically loaded loop cache verses a hybrid cache with no preloaded loops.

	Main Loop Cache Size			
	16	**32**	**64**	**128**
Dynamic	30%	30%	30%	29%
Hybrid	29%	29%	29%	28%

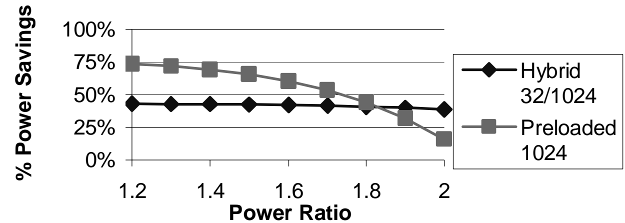


Figure 3: Power savings for increasing capacitance ratios.

preloading is an option, it can hold more loops than a preloaded loop cache of the same size because it does not need to store the loops that can be dynamically loaded. Future work includes developing an efficient loop match memory to be able to support large numbers of loops without power overhead for the comparisons and investigating the impact our design has on system performance.

6. Acknowledgements

This work was supported in part by the National Science Foundation (CCR-9876006) and a GAANN Fellowship.

References

[1] Y. Aghaghiri, F. Fallah, M. Pedram. Irredundant Address Bus Encoding for Low Power. International Symposium on Low Power Electronics and Design, Aug. 2001, pp. 82-87.

[2] N. Bellas, I. Hajj, C. Polychronopoulos, G. Stamoulis. Energy and Performance Improvements in Microprocessor Design Using a Loop Cache. Int. Conference on Computer Design, pp. 378-383, 1999.

[3] D. Burger, T. Austin, S. Bennet. Evaluating Future Microprocessors: The SimpleScaler ToolSet. University of Wisconsin-Madison. Computer Science Department. Tech. Report CS-TR-1308, July 1996.

[4] T. Givargis, J. Henkel, F. Vahid. Interface and Cache Power Exploration for Core-Based Embedded System, Int. Conf. on Computer-Aided Design (ICCAD), November 1999, pp. 270-273.

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

[6] S.C. Govindarajan. G. Ramaswamy, M. Mehendale. Area and Power Reduction of Embedded DSP Systems using Instruction Compression and Re-configurable Encoding. International Conference on Computer-Aided Design, 2001.

[7] J. Kin, M. Gupta, W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. International Symposium on Microarchitecture, December 1997.

[8] C. Lee, M. Potkonjak, W.H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. Proc 30th Annual International Symposium on Microarchitecture, December 1997.

[9] L.H. Lee, W. Moyer, J. Arends. Low-Cost Embedded Program Loop Caching – Revisited. U. Mich. Technical Report Number CSE-TR-411-99, December 1999.

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

[11] S. Segars. Low Power Design Techniques for Microprocessors. ISSCC Feb 4, 2001.

[12] Synopsys Inc., http://www.synopsys.com.

[13] J. Villarreal, R. Lysecky, S. Cotterell, F. Vahid. Loop Analysis of Embedded Applications. UC Riverside Tech. Report UCR-CSE-01-03, 2001.