Low-Cost Embedded Program Loop Caching - Revisited

Lea Hwang Lee[†], Bill Moyer*, John Arends*

Advanced Computer Architecture Lab[†]
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
leahwang@eecs.umich.edu

M•CORE Technology Center*
Motorola, Inc.
P.O. Box 6000, MD TX77/F51
Austin TX 78762-6000
{billm,arends}@lakewood.sps.mot.com

Abstract

Many portable and embedded applications are characterized by spending a large fraction of their execution time on small program loops. In these applications, instruction fetch energy can be reduced by using a small instruction cache when executing these tight loops. Recent work has shown that it is possible to use a small instruction cache without incurring any performance penality [4, 6]. In this paper, we will extend the work done in [6]. In the modified loop caching scheme proposed in this paper, when a program loop is larger than the loop cache size, the loop cache is capable of capturing only part of the program loop without having any cache conflict problem. For a given loop cache size, our loop caching scheme can reduce instruction fetch energy more than other loop cache schemes previously proposed. We will present some quantitative results on how much power can be saved on an integrated embedded design using this scheme.

December 18th, 1999

1 Introduction and Related Work

Many portable and embedded applications are characterized by spending a large fraction of their execution time on small program loops. In these applications, instruction fetch energy can be reduced by using a small instruction buffer (also call a *loop cache*) when executing these tight loops [2,5,12, etc.].

Using this approach, however, may incur cycle penalties due to instruction buffer misses (the requested instructions are not found in the buffer). Another approach is to operate the main cache in the same cycle only if there is a buffer miss, possibly at the expense of a longer cycle time.

1.1 Compiler Assisted Loop Caching Scheme

In [3], Bellas et. al. proposed using a compiler to select and place frequently used basic blocks into a region of code by using profile information. In this approach, a special instruction is used to mark the boundary between the portion of code that is to be cached in the loop cache and the portion of code that is not. The address of this boundary is loaded into a 32-bit register by the run-time hardware. If the PC is less than this address, the loop cache is probed to determined if there is a loop cache hit. If the PC is greater than this address, the loop cache is by-passed and the request is sent directly to the main instruction cache. In this approach, a loop cache miss due to a conflict between two selected basic blocks will incur a cycle penalty [3].

1.2 Special Loop Instructions

Several commercial ISA use special loop instructions to execute program loops [1,13,14]. Specified in these instructions are the loop size, the register that is used as a loop counter, and the loop terminating condition. When a loop is encountered, the hardware can load the program loop into a loop cache and start using the loop cache for loop executions. In these systems, loop caches are used without any performance penalty.

1.3 Multiple Cache Line Buffering

In [4], multiple line buffers are used. Each of these line buffers has a set number tag, storing the set number field of the address associated with the cache line stored in the buffer. If the set number field of a requested address matches one of these set number tags, the associated line buffer is read and the access to the cache array is aborted. Figure 1 shows the cache organization using this technique with two line buffers [4]. In this approach, there is no cycle penalty nor cycle time degradation, provided that an access to the cache array can be aborted in time after it is determined that the requested instruction is not in the line buffers.

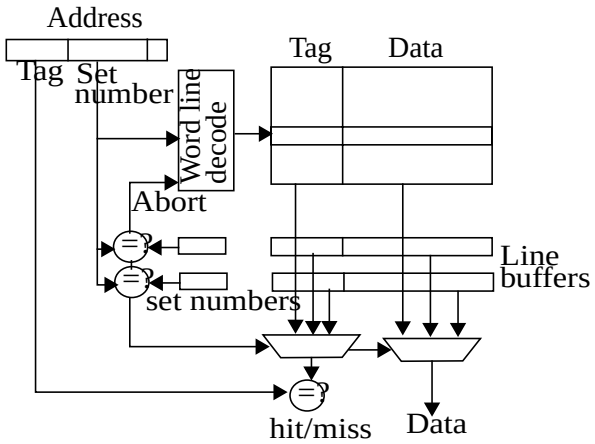


Figure 1: Multiple cache line buffering

1.4 Counter-Based Loop Caching Scheme

In [6], a counter-based loop caching scheme was proposed. In this scheme, the loop cache does not have an address tag. The loop cache array can be implemented as a direct mapped array. There is no valid bit associated with each loop cache entry. A program loop does not need to be aligned to any particular address boundary. The software is allowed to place a loop at any arbitrary starting address. Furthermore, the loop cache controller knows whether the next instruction will hit in the loop cache, well ahead of time. As a result, no performance degradation is incurred.

In this loop caching scheme, no special instruction is needed for loop executions. Any regular conditional or unconditional branch instructions can be used to construct a program loop. Program loop constructions, in this case, are more flexible (a wide variety of terminating conditions is made available by choosing from different flavors of branches available in the ISA). By eliminating the need for defining a special loop instruction, this technique also preserves some valuable opcode space for other useful instructions.

Figure 2 shows the loop cache organization using this technique with a cache size of 2^c instructions. The loop cache size used in [6], is about 32 to 64 bytes.

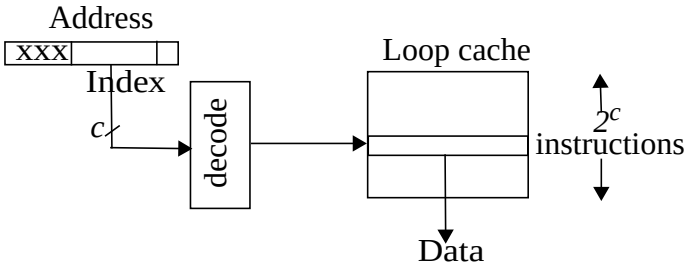


Figure 2: Loop Cache Organization

The low area cost associated with this scheme offers tight integration solution into the processor core. Figure 3 shows an example of a counter-based loop cache being integrated into an instruction buffer inside the processor core.

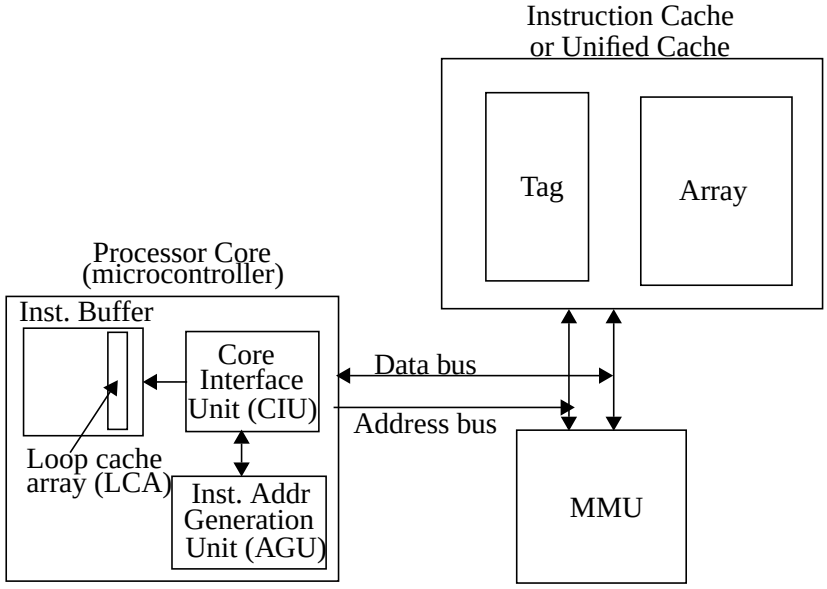
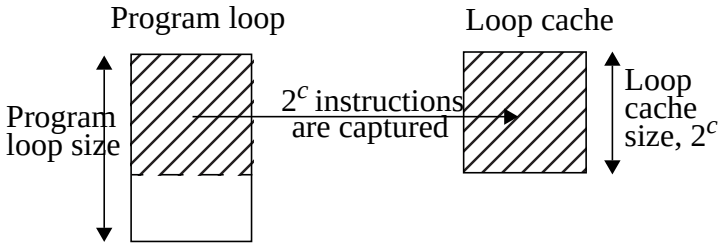


Figure 3: An Embedded Microcontroller System

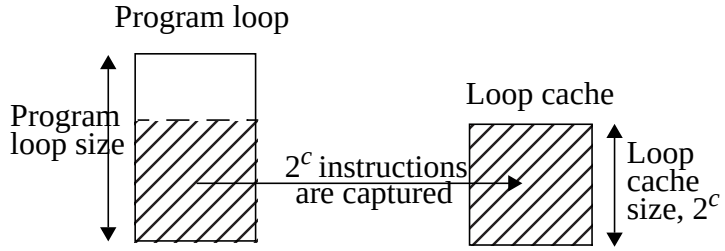
Compared to the Multiple Line Buffering approach proposed in [4], when executing a program loop, this scheme does not exercise any hardware that is associated with the instruction address generation. These hardware include the instruction address generation unit (AGU), processor core interface unit (CIU) and the instruction address bus (see Figure 3). Furthermore, this scheme does not access instruction tag memory nor perform any tag compare

In this work, we will extend the work done in [6]. In the modified loop caching scheme proposed in this work, when a program loop is larger then the loop cache size, the loop cache is capable of capturing only part of the program loop without having any cache conflict problem. For a given loop cache size, our caching scheme can reduce instruction fetch energy more than other loop caching schemes previously proposed.

Figure 4 shows a scenario where the program loop size is larger than the loop cache size. It shows that only part of the program loop is being captured in the loop cache. Depending on how the loop cache is filled up, either the upper part of the program loop or the lower part of the program loop is captured in the loop cache. Thus a loop cache size of 2^c instructions, can not only capture all the loops that are equal to or smaller than 2^c instructions, it can also capture a portion of all the larger loops. In the latter case, we can achieve a loop cache hit rate of 2^c instruction references per iteration.



(a) Upper part of the program loop is being captured



(b) Lower part of the program loop is being captured

Figure 4: Program loop size is larger than the loop cache size

The rest of this paper is organized as follows. Section 2 describes the extended counter based loop caching scheme based on [6]. It describes the implementation of the loop cache controllers and how the hardware monitors the program loop execution. Section 3 describes the implementation of two different loop cache fill strategies, namely, the cold-fill strategy and the warm-fill strategy. Section 4 describes the benchmark suite used in this study and some branch characteristics of these benchmarks. Section 5 and Section 6 give some experimental results and quantify how much power saving can be achieved using this scheme. Section 8 summarizes this paper.

Without loss of generality, we will assume, in this paper, that each instruction is 2 byte long. Also, in this paper, we will refer to the main instruction cache, whether split or unified, as the “main Icache”. We will denote the size of the program loop as LS, in number of instructions in the loop.

Example 1

L1:

A_0 I_0

A_1 I_1

A_2 I_2 } Loop body

:

A_{n-2} I_{n-2}

A_{n-1} br L1 // branch

2 Counter Based Program Loop Caching

Consider Example 1 shown above. A loop contains n instructions (i.e. LS=n). The last instruction of the loop is a backward branch instruction, br, with target L1.

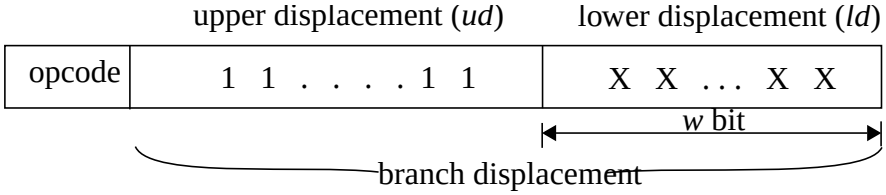
2.1 Loop Cache Organization

Figure 2 shows the organization of a 2^c-entry loop cache, for some integer *c*. When the size of the program loop being captured is smaller than or equal to the loop cache size ($LS \leq 2^c$), the entire loop is captured. In this case, indexing into the loop cache is unique and non-conflicting. That is: (i) an instruction in the program loop will always be mapped to a unique location in the loop cache array; and (ii) there can never be more than one instruction from a given program loop to compete for a particular cache location.

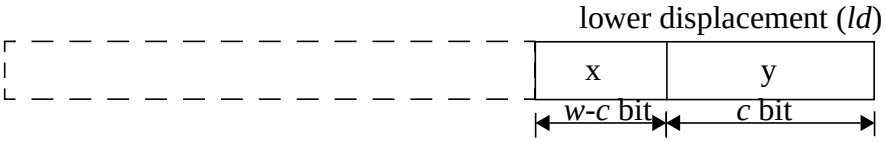
When the size of the program loop being captured is larger than the loop cache size ($LS > 2^c$), then only part of the program loop is captured in the loop cache. In this case, indexing into the loop cache is unique but can be conflicting. That is, multiple instructions in the loop can be mapped into the same location in the loop cache. Thus when accessing the loop cache, additional hardware is needed to determine which part of the program loop is actually stored in the loop cache. We will describe this further in Section 3.

2.2 Short Backward Branch Instruction

The notion of a *short backward branch instruction* (sbb) was introduced in [6]. A sbb is any PC-relative branch instruction that fits the instruction format shown in Figure 5(a). It can be conditional or unconditional. A sbb has negative branch displacement, with its upper displacement field that is all ones and its lower displacement field that is *w*-bit wide, where $w \geq c$.



(a) sbb instruction format



(b) Further division of lower displacement field

Figure 5: sbb Instruction Format

The lower displacement field, *ld*, can be further divided into two fields: x field (*w*-*c* bit wide) and y field (*c* bit wide). By definition, the maximum backward branch distance of a sbb is 2^w instructions. Thus the maximum program loop size that can be recognized by the hardware is 2^w instructions. Since $w \geq c$, this maximum loop size can be larger than the loop cache size. The latter is given by 2^c instructions. Furthermore, *if the x field of the sbb are all ones, it indicates that the program loop size is equal to or smaller than the loop cache size*. Conversely, if the x field of the sbb are not all ones, it indicates that the program loop size is larger than loop cache size. For the case where *w=c*, this work degenerates into those in [6].

When a sbb is detected in an instruction stream and found to be taken, the hardware assumes that the program is executing a loop and initiates all the appropriate control actions to utilize the loop cache. The sbb that triggers this transition is called the *triggering sbb* [6].

2.3 Monitoring the sbb Executions

In [6], a scheme for monitoring the sbb executions was proposed. For the sake of completeness, we will describe this monitoring scheme in this Section.

In order to determine in advance, whether the next instruction fetch will hit in the loop cache, the controller needs the following information on a cycle-to-cycle basis: (a) is the next instruction fetch a sequential fetch or is there a *change of control flow* (cof)? (b) if there is a cof, is it caused by the triggering sbb? (c) is the loop cache completely warmed up with the program loop so we could access the loop cache instead of the main cache?

Information pertaining to (a) can be easily obtained from the instruction decode unit as well as the fetch and branch unit in the pipeline. In Section 3, we will describe how we could obtain information pertaining to (c).

A counter could be used to obtain information pertaining to (b). In this scheme, when a sbb is encountered and taken, its lower displacement field, *ld*, is loaded into a *w*-bit increment counter called *loop_count* (see Figure 6). The hardware then infers the size of the program loop from this branch displacement field. It does so by incrementing this negative displacement by one, each time an instruction is executed sequentially. When the counter becomes zero, the hardware knows that we are executing the triggering sbb. If the triggering sbb is taken again, the increment counter is re-initialized with the *ld* field from the sbb, and the process described above repeats itself. Using this scheme, the controller knows whether a cof is caused by the triggering sbb, by examining the value of *loop_count* [6].

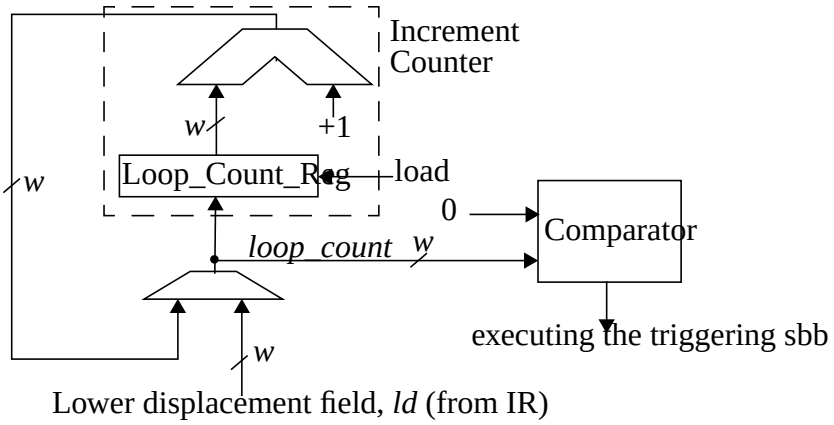


Figure 6: A Counter To Monitor sbb Executions

3 Cache Fill Strategies

Depending on how a program loop is being captured and stored in the loop cache, two loop cache fill strategies are identified: (i) *Cold-Fill Strategy*; and (ii) *Warm-Fill Strategy.*

In the cold-fill strategy, the loop cache controller will only try to fill up the loop cache *after* it is confirmed that we are executing a program loop.

In the warm-fill strategy, the loop cache controller fills up the loop cache whenever possible and tries to keep the loop cache warm at all times.

If the program loop is smaller than the loop cache size, the entire program loop will be captured by the loop cache. However, if the program loop is larger than the loop cache size, the cold-fill strategy will only capture the *upper part* of the loop (Figure 4(a)); while the warm-fill strategy will only capture the *lower part* of the loop (Figure 4(b)).

3.1 Cold-Fill Strategy

In the cold-fill strategy, the controller begins to fill up the loop cache during the second iteration, after the sbb of the first iteration is taken. During the second iteration, the controller fills up the loop cache either until the entire loop is captured (for the case where the program loop is equal to or smaller than the loop cache size) or until the loop cache is completely full (for the case where the program loop is larger than the loop cache size). From the third and subsequent iterations, the controller then directs some or all of the instruction fetches to the loop cache.

If the program loop is larger than the loop cache size, the controller will only access the loop cache for the first 2^c instructions of the program loop. For the rest of the program loop, all instruction accesses will be directed towards the main Icache directly, bypassing the loop cache.

A state diagram for the loop cache controller using cold-fill strategy is shown in Figure 7 [6].

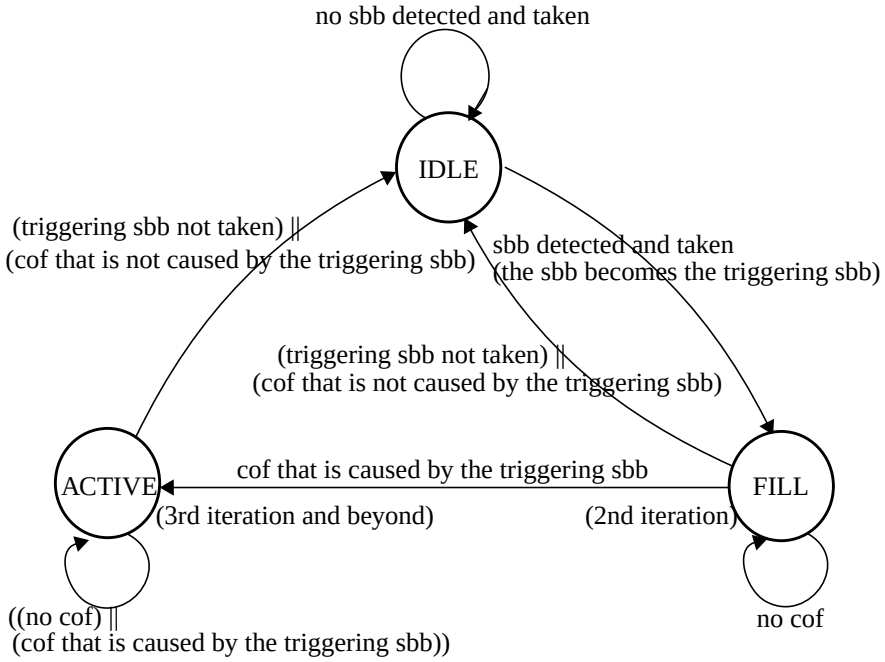


Figure 7: Loop cache controller with cold-fill strategy

The controller starts off with an IDLE state. When a sbb is decoded and taken, the controller enters the FILL state. While in the FILL state, the controller fills up the loop cache with all the instructions being

fetched from the main Icache, either until the entire loop is captured or until the loop cache is full. While in the FILL state, one of the following two conditions will cause the controller to return to the IDLE state: (i) the triggering sbb is not taken; or (ii) there is a cof within the loop body that is not caused by the triggering sbb.

While in the FILL state, if the triggering sbb is taken again, the controller then enters an ACTIVE state. While in the ACTIVE state, one of the following two conditions will cause the controller to return to the IDLE state: (i) the triggering sbb is not taken; or (ii) there is a cof within the loop body that is not caused by the triggering sbb.

To monitor the number of instructions filled in the loop cache during the FILL state, and the number of instructions accessed from the loop cache during ACTIVE state, a dual purpose saturating counter, called *inst_count*, can be used. This counter is *c*-bit wide. This is in addition to the counter used for monitoring the sbb executions, shown in Figure 6.

The *inst_count* counter is reset to zero each time the triggering sbb is taken and is incremented by one for each sequential instruction requested. While in the FILL state, the loop cache fill activities will cease when the triggering sbb is taken again, or when *inst_count* saturates, indicating that we have already fetched 2^c instructions.

While in the ACTIVE state, all instruction requests will be directed to the loop cache, unless *inst_count* saturates. In the latter case, all the subsequent instruction requests will be directed to the main Icache directly.

If a program loop size is smaller than the loop cache size, indicated by the "x" field of the sbb being all ones (see Figure 5), then the *inst_count* counter needs not be exercised at all during the execution of that loop.

3.2 Warm-Fill Strategy

In the warm-fill strategy, after the sbb of the first iteration is taken, some or all of the program loop is already captured in the loop cache. The loop cache can be accessed starting from the second iteration.

Similar to its cold-fill counterpart, an additional counter, called the *inst_count*, is used. This counter is *w*-bit wide. While in the FILL state, *inst_count* keeps track of the number of instructions being captured into the loop cache. It is incremented by one for each instruction being captured sequentially and is reset to zero when a sbb is detected and taken. While in the ACTIVE state, *inst_count* keeps track of the number of instructions being requested by the execution pipeline. It is incremented by one for each instruction being requested sequentially and is reset to zero when the triggering sbb is taken

.

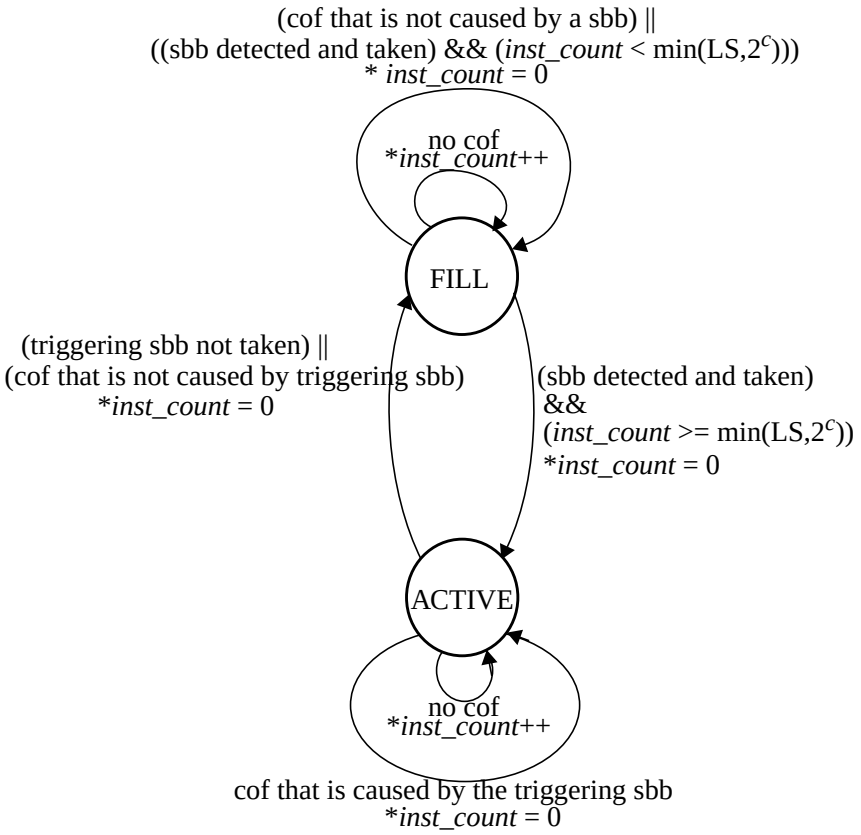


Figure 8: Loop cache controller with warm-fill strategy

A state machine for the loop cache controller using warm-fill strategy is shown in Figure 8. All the update actions for *inst_count* counter are marked in Figure 8 as "*". The controller starts off with a FILL state. While in the FILL state, the controller tries to fill up the loop cache. While in the FILL state, if a sbb is detected and found to be taken and *inst_count* ≥ min(LS,2^C), the controller enters an ACTIVE state. Otherwise, it remains in the FILL state. The condition *inst_count* ≥ min(LS,2^C) guarantees that one of the following two conditions holds true before the controller enters the ACTIVE state: (i) the entire program loop is being captured by the loop cache (for the case where the program loop size is smaller than the loop cache size); or (ii) loop cache is completely full (for the case where the program loop size is equal to or larger than the loop cache size).

While in the ACTIVE state, one of the following two conditions will cause the controller to return to the FILL state: (i) the triggering sbb is not taken; or (ii) there is a cof within the loop body that is not caused by the triggering sbb.

For the case where the program loop size is smaller than the loop cache size, the controller directs all instruction requests to the loop cache, immediately starting from the second iteration. For the case where the program loop size is larger than the loop cache size, the controller directs an instruction request to the loop cache only if *inst_count* > LS-2^C. The size of the un-captured upper portion of the loop is given by LS-2^C. Condition *inst_count* > LS-2^C indicates that the number of instructions being requested has surpassed the size of the upper portion of the program loop. From that point on, all subsequent requests will hit in the loop cache.

Table 1: Powerstone benchmark suite

Benchmark	Dynamic Inst. Count	Inst. Ref. vs. Total Ref.	Description
auto	17374	0.67	Automobile control application
blit	72416	0.75	Graphics application
compress	322101	0.68	A Unix utility
des	510814	0.80	Data Encryption Standard
engine	955012	0.63	Engine control application
fir_int	629166	0.70	Integer FIR filter
g3fax	1412648	0.76	Group three fax decode
g721	231706	0.55	Adaptive differential PCM for voice compression
jpeg	1342076	0.65	JPEG 24-bit image decompression standard
map3d	1228596	0.82	3D interpolating function for automobile control applications
pocsag	131159	0.72	POCSAG communication protocol for paging applications
servo	41132	0.50	Hard disc drive servo control
summin	1330505	0.77	Handwriting recognition
ucbqsort	674165	0.76	U.C.B. Quick Sort
v42bis	1488430	0.76	Modem encoding/decoding
Average	-	0.70	-

4 Benchmarks

The benchmark suite used in this study, called the Powerstone benchmark suite, includes paging, automobile control, signal processing, imaging, and fax applications. It is detailed in Table 1. These benchmarks are compiled to the M•CORE™ ISA using the Diab 4.2.2 compiler.

The M•CORE ISA has a fixed 16-bit instruction format [11]. In this ISA, all conditional change of control flow instructions are PC-relative, each with an offset specified in the branch instruction. Unconditional cof instructions can be PC-relative, jump register indirect, or jump indirect through a memory location.

These benchmarks exhibit significant tight loop structures. The sizes of these loops are predominately less than 16 instructions. Furthermore, PC-relative branch instructions are extensively used to construct these tight loops[6].

5 Main Instruction Cache Access Rates

Instruction level simulations were performed using the Powerstone benchmarks to quantify the effectiveness of our loop caching technique. As mentioned earlier, there is no cycle count penalty nor performance degradation associated with this technique.

We define the *main instruction cache access rates* (MCAR) as the number of instruction references made to the main Icache when using a loop cache, as a percentage of number of instruction references

made to the main Icache without using a loop cache.

5.1 Limited Loop Size Scheme vs. Flexible Loop Size Scheme

For comparison purposes, we will present both the results for the extended loop caching scheme presented here, and those for the original scheme proposed in [6]. In [6], the maximum allowable program loop size is limited by the loop cache size. i.e. $w=c$. We will call this scheme the *Limited Loop Size Scheme*.

In this work, we allow the program loop size to be larger than the loop cache size. i.e. $w \geq c$. We will call this extended scheme the *Flexible Loop Size scheme*. For the rest of this paper, we will assume that for the Flexible Loop Size scheme, w=7. i.e. all program loops are assumed to be 128 instructions or less. Any loop larger than 128 instructions will be ignored by the hardware.

5.2 Overall MCAR

Figure 9 shows the average MCAR for the entire benchmark suite, as a function of loop cache size, c. Four sets of MCAR are shown. They are the results of the combination of loop cache fill strategies and the Limited versus Flexible Loop Size schemes.

All of these access rates decrease drastically between c=2 and c=4. From c=5 onwards, there is virtually no improvement. This is consistent with the observation made in Section 4 where the program loops were dominated by sizes of 16 instructions (32 bytes) or less.

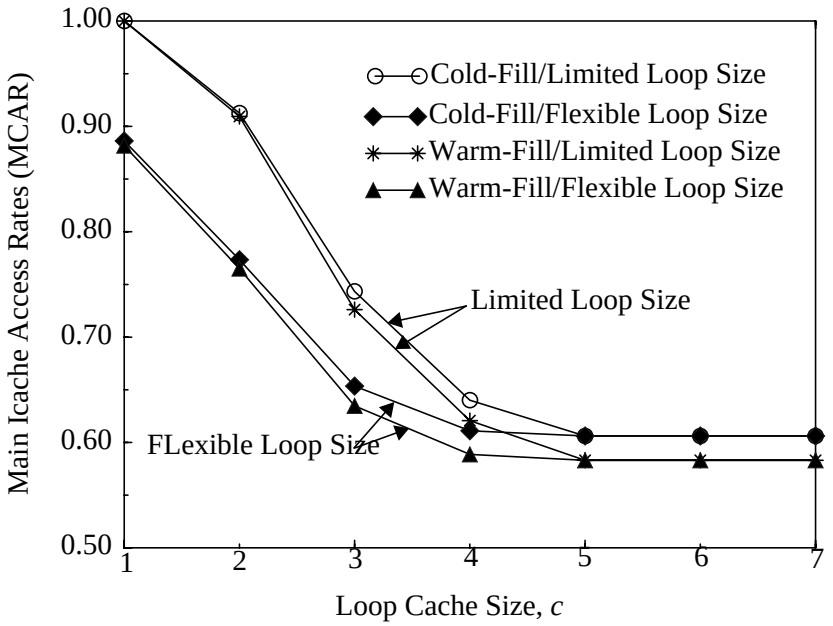


Figure 9: Main Icache access rate (MCAR)

5.1.1 Cold-Fill Strategy vs. Warm-Fill Strategy

Warm-fill strategy offers only slightly lower MCAR than the cold-fill strategy. The warm-fill strategy will offer significant advantage over its cold-fill counterpart only when the number of iterations per loop invocation are small (about two or three iterations per loop invocation). In the benchmarks that we have evaluated, this situation did not happen very often. That is, when a small loop is invoked, it is executed relatively large number of times. As a result, for benchmarks that could benefit greatly from the warm-fill strategy, they also benefited greatly from the cold-fill strategy.

5.1.2 Limited Loop Size vs. Flexible Loop Size

The flexible loop size scheme offers significantly lower MCAR than the limited loop size scheme, for loop cache size of 16 instructions or less ($c \leq 4$). For such small loop cache sizes, cache conflict is the major cause for having poor loop cache utilization, even for programs that exhibit tight loop structures. Take a loop cache size of 8 instructions, for example. Not only can the flexible loop size scheme capture the entire loop for all the loops with size of 8 instructions or less, but it can also capture 8 instructions per iteration for all the loops with size of up to 127 instructions. Since all the program loops are dominated by size of 4-16 instructions, loop cache sizes of 2 to 8 instructions stand to gain the most benefits from using the flexible loop size scheme.

For cold-fill strategy with c=2, moving from limited loop size scheme to flexible loop size scheme reduces the average MCAR from 0.92 to 0.77; for cold-fill strategy with c=3, the move reduces the average MCAR from 0.74 to 0.65.

6 Power Saving

To quantify the power saving of using the loop caching scheme proposed in this work, power measurements were taken from an actual CMOS 0.25um, 1.75v, integrated design. This design consists of a M-CORE micro-controller, a 8 KB, 4-way set-associative unified cache, a MMU unit and some other Modules. The block diagram of this system is very similar to that shown in Figure 3. It is targeted for various mobile applications.

The power measurements were obtained using a combination of Powermill^{TM} simulations on a back annotated design, and by using high level c/c++ simulations. Powermill simulations were performed on each benchmark program to collect power measurements on various parts of the design. The high level c/c++ simulations were performed to obtain information such as number of instruction and data references made to the unified cache, the bus activities, the loop cache utilization, etc.

We will present the power consumption in terms of the percentage of the total power consumed by the whole design (the micro-controller, the cache, the MMU and a few other modules).

6.1 Instruction Versus Data References

Table 1 shows the ratio between the number of instruction references and the total number of references (instruction and data included). Overall, instruction references constitutes about 70% of all references made to the unified cache. Instruction references contribute about 63% of switching activities on the address bus, and about 75% on the data bus (not shown in this Table). In this integrated system, the capacitive loading on the data bus versus the address bus is about 3:1.

6.2 Power Measurements

Table 2 shows the average percentage power consumption, over all benchmarks, for some of the blocks shown in Figure 3. In this table, the loop cache array size is assumed to be 16-entry. The power consumptions for the loop cache controllers are estimated based on the estimated logic usage and the average power density found in this design. Some additional margin has been added to these estimates.

Table 2: Breakdown of Percentage Power

Blocks	% of Total Power
Address Generation Unit (AGU)	0.6228%
Core Interface Unit (CIU)	2.2699%
Loop Cache Array (LCA16)	0.3552%
Loop Cache Controller (LCC)[a] (Cold-fill)	0.7500%
Loop Cache Controller (LCC)[a] (Warm-fill)	1.5000%
Main Unified Cache	51.3728%

a. Estimated only

The power saving due to the loop cache can be approximated as follows:

Power saving = $(P_{cache}+P_{AGU}+P_{CIU})*I_{ref}*(1\text{-}MCAR)$
$- (P_{LCA16}/16*LCS + P_{LCC})$

where: P_X denotes the percentage power for block "X";
I_{ref} denotes the percentage of instruction references of all cache accesses;
LCS denotes the loop cache size in number of instruction entries.

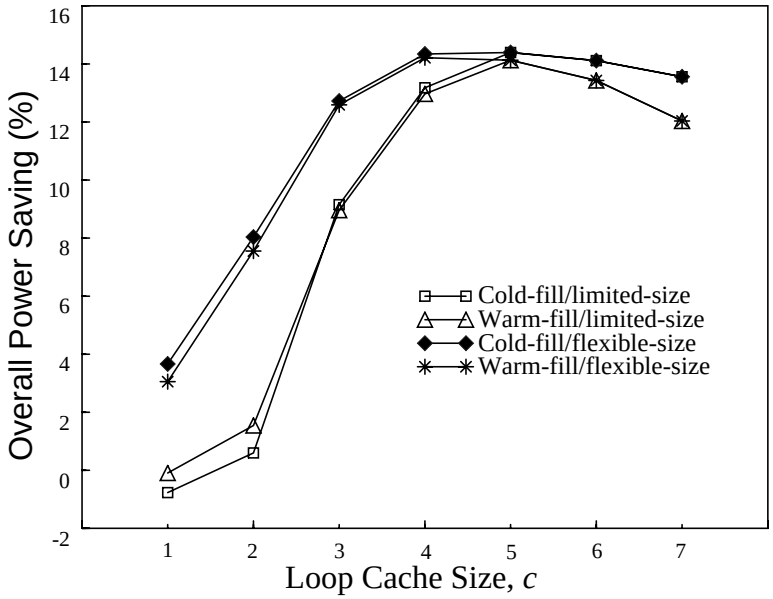


Figure 10: Overall Power Saving

Figure 10 shows the overall power saving for four combinations of cache fill strategies and loop size schemes. This figure shows that the flexible-sizing scheme outperforms the limited-sizing scheme, especially when the loop cache size is small ($c \leq 4$). However, as the loop cache size increases, the power due to the loop cache array starts to increase, causing the overall power saving to decrease.

Comparing the warm-fill and cold-fill strategies, warm-fill has higher loop cache hit rates but lower power saving, primarily due to its constant utilization of the loop cache array and its controller. Cold-fill,

on the other hand, is simpler to implement and achieves higher power savings.

Figure 11 shows the power saving for using a loop cache with cold-fill, flexible loop size scheme, with *c*=3 (8-entry) and *c*=4 (16-entry). The power saving, in this case, varies greatly across the benchmark suite. For *c*=4, the power saving ranges from -0.87% (wasting power) for `ucbqsort`, to 39% for `blit`. Overall, the saving is 12.72% for *c*=3 and 14.34% for *c*=4.

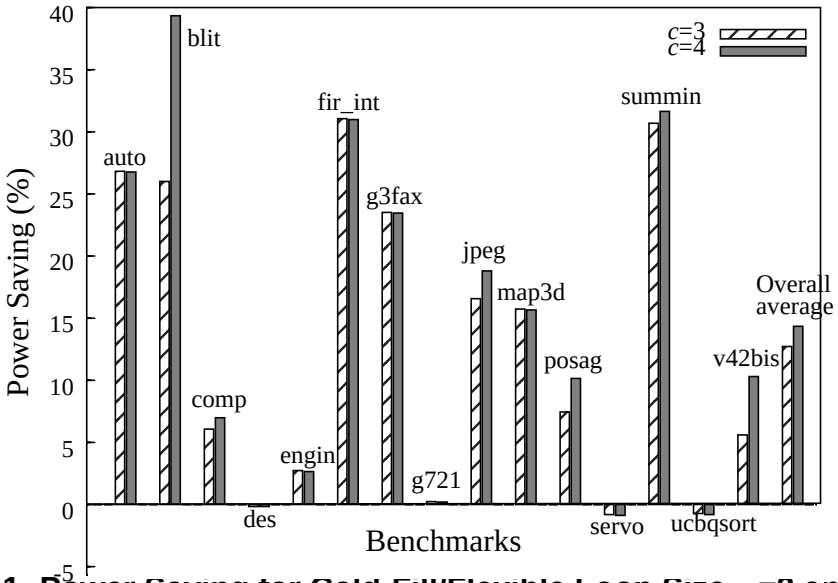


Figure 11: Power Saving for Cold-Fill/Flexible Loop Size, *c*=3 and c=4

7 Acknowledgements

The authors would like to thank Steve Layton and Afzal Malik, both from M-CORE Technology Center, Motorola, for all their assistance in Powermill simulations.

8 Summary

Low system cost and low energy consumption are two important factors to consider in designing many portable and handheld systems. To reduce memory costs, more and more memory is now integrated on-chip. As a result, the area and power consumption due to on-chip memory will continue to increase in the near future.

In this work, we proposed a low-cost instruction caching scheme to reduce the instruction fetch energy when executing small tight loops. Our proposed technique is unique in the following ways [6,7,9,10]:

- Low area costs: the loop cache does not have an address tag. The array is direct mapped. There is no valid bit associated with each array entry.
- The scheme does not need any special loop instruction. Furthermore, it allows a program loop to be constructed using different types of branch instructions available in the ISA.
- With cold-fill strategy, the loop cache array and its controller are not exercised when we are not executing a program loop.
- The program loops are naturally aligned. That is, a program loop does not need to be aligned to any address boundary in order to take full advantage of this caching scheme.
- When executing a program loop, all hardware associated with instruction address generation is not exercised. Also, the instruction address tag access and compare are not performed.
- Since the controller knows precisely whether the next sequential instruction request will hit in the loop cache, well ahead of time, there is no cycle count penalty nor cycle time degradation associated

with this technique.

- Small area overhead has other benefits as well. It allows tight integration of the loop cache array into the processor core. Tight integration further reduces energy consumptions due to instruction fetches. It also reduces the adverse effect on timing due to the presence of the loop cache.

With a 16-entry 32-byte (c=4) loop cache, we can reduce the overall power by about 14.34%.

9 References

[1] *ADSP-2106x SHARC™ User's Manual*, Analog Devices Inc., 1998.

[2] N. Bellas, I. Hajj and C. Polychronopoulos, "A New Scheme for I-Cache energy reduction in High-Performance Processors," *Power Driven Microarchitecture Workshop*, held in conjunction with ISCA 98, Barcelona, Spain, June 28th 1998.

[3] N. Bellas, I. Hajj, C. Polychronopoulos and G. Stamoulis, "Energy and Performance Improvements in Microprocessor Design Using a Loop Cache," *Proc. Int'l Conf. on Computer Design*, Austin, Texas, October 1999.

[4] K. Ghose, M. Kamble, "Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation," *Proc. Int'l. Symp. on Low Power Electronics and Design*, August, 1999.

[5] J. Kin, M. Gupta and W. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *Proc. Int'l. Symp. on Microarchitecture*, pp. 184-193, December, 1997.

[6] L. H. Lee, B. Moyer, J. Arends, "Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops," *Proc. Int'l. Symp. on Low Power Electronics and Design*, August, 1999.

[7] L. H. Lee et. al., Patent Pending in US, Germany, France, GB, Italy, Netheland, Japan, China, Korea, Taiwan, "*Data Processor System Having Branch Control and Method Therefor,*" filed 19th June 1998, Motorola Incorp.

[8] *M•CORE Reference Manual*, Motorola Inc., 1997.

[9] B. Moyer, L. H. Lee and J. Arends, US Patent number 5,920,890, "*Distributed Tag Cache Memory System and Method for Storing Data in the Same,*" April, 6th 1999.

[10] B. Moyer, L. H. Lee and J. Arends, US Patent number 5,893,142, "*Data Processing System Having a Cache and Method Therefo,*" July, 6th 1999.

[11] B. Moyer and J. Arends, "RISC Gets Small," *Byte Magazine*, February 1998.

[12] C. Su and A. M. Despain, "Cache Design Trade-offs for Power and Performance Optimization: A Case Study,", *Proc. Int'l. Symp. on Low Power Design*, pp. 63-68, 1995.

[13] *TMS320C2x User's Guide*, Texas Instruments Inc., 1993.

[14] *TriCore™ Architecture Manual*, Siemens Incorp., 1997.