

Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops

Lea Hwang Lee, Bill Moyer, John Arends

M•CORE Technology Center, Motorola, Inc.

P.O. Box 6000, MD TX77/F51

Austin, TX 78762-6000

{leahwang, billm, arends}@lakewood.sps.mot.com

1. ABSTRACT

A fair amount of work has been done in recent years on reducing power consumption in caches by using a small instruction buffer placed between the execution pipe and a larger main cache [1,2,6]. These techniques, however, often degrade the overall system performance. In this paper, we propose using a small instruction buffer, also called a loop cache, to save power. A loop cache has no address tag store. It consists of a direct-mapped data array and a loop cache controller. The loop cache controller knows precisely whether the next instruction request will hit in the loop cache, well ahead of time. As a result, there is no performance degradation.

Keywords: Low cost, low power, embedded systems, small program loops, instruction buffering

2. INTRODUCTION

Many embedded applications are characterized by spending a large fraction of execution time on small program loops. A fair amount of work has been done in recent years on reducing power consumption in caches by using a small instruction buffer placed between the execution pipe and a larger main cache [1,2,6]. 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. In portable embedded systems, these performance degrading techniques could have an adverse affect on the energy consumption at the system level [3].

In this paper, we propose using a small instruction buffer, referred to here as a *loop cache*, to reduce the instruction fetch energy when executing tight program loops. This is achieved without degrading the performance, as measured by increased number of execution cycles or longer cycle

time. In this technique, the instruction fetch datapath consists of two levels of instruction caching: an on-chip main cache (unified or split), and a small loop cache. Instructions are supplied to the execution core either from the loop cache or from the main cache. The loop cache controller knows precisely whether the next instruction request will hit in the loop cache, well ahead of time. The loop cache has no address tag store. Its cache array can be implemented as a direct-mapped array. Furthermore, there is no valid bit associated with each loop cache entry.

Our proposed technique is based on the definition, detection and utilization of a special class of branch instructions, called the short backward branch instruction (sbb). When a sbb is detected in an instruction stream and is found to be taken, the loop cache controller assumes that we are starting to execute the second iteration of a program loop, and tries to fill up the loop cache. Starting from the third iteration, the controller directs all the instruction requests to the loop cache and shuts of the main cache completely.

3. Short Backward Branch Instructions

A *short backward branch instruction* (sbb) is any PC-relative branch instruction that fit the following instruction format. It can be conditional or unconditional.

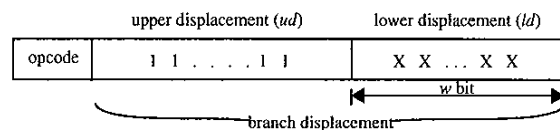


Figure 1. sbb Instruction Format

The upper portion of the branch displacement field of a sbb are all ones (indicating a negative branch displacement). The lower portion of the branch displacement field, *ld*, is *w* bit wide. By definition, a sbb has a *maximum* backward branch distance given by 2^w instructions. The size of the loop cache is also given by 2^w instructions. This definition ensures that if a sbb is decoded and recognized as such by the decoding hardware, the loop size in question is guaranteed to be no larger than the loop cache size.

Triggering sbb

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*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISLPED99, San Diego, CA, USA
©1999 ACM 1-58113-133-X/99/0008..\$5.00

4. Loop Cache Organization

Figure 2 shows the organization of a 2^w -entry loop cache and how it is being accessed with an instruction address $A[31:0]$. A loop cache does not have an address tag store. The loop cache array can be implemented as a direct-mapped array. It is indexed using the $\text{index}(A)$ field of the instruction address. The $\text{index}(A)$ field is w -bit wide. The array can store 2^w instructions. By definition of a sbb, the maximum program loop size that can be recognized and captured by the loop cache is 2^w instructions. Thus accessing the loop cache during program loop execution is guaranteed to be *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 could never be more than one instruction from a given program loop to compete for a particular cache location. When the program loop being captured is smaller than the loop cache size, only part of the loop cache array is utilized.

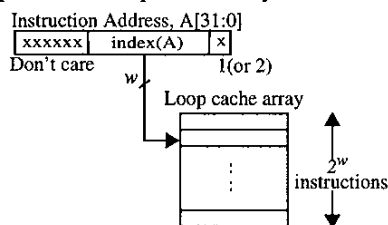


Figure 2. Loop Cache Organization

Unlike many other loop caching schemes, our loop caching scheme does not require the program loop to be aligned to any particular address boundary. The software can place a loop at any arbitrary starting address.

5. Determining Loop Cache Hit/Miss Early

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.

5.1 Monitoring sbb Execution

To obtain information pertaining to (b), a counter based scheme similar to those proposed in [3] could be used. In this scheme, when a sbb is encountered and taken, its lower displacement field, ld , is loaded into a w -bit increment counter called Count_Register (see Figure 3). 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 in the Count_Register .

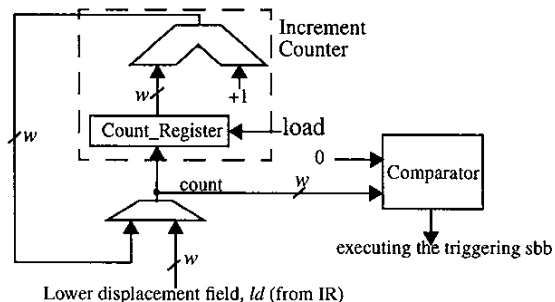


Figure 3. Counter based scheme to monitor sbb executions

5.2 Loop Cache Controller

The state transition diagram for the loop cache controller is shown in Figure 4. The controller ensures that the loop cache is warmed up before it is being utilized. It begins with an IDLE state. When a sbb is decoded, its ld field is loaded into the Count_Register . If the sbb is taken, the controller enters a FILL state. The sbb becomes the triggering sbb. When in the FILL state, the controller fills the loop cache with the instructions being fetched from the main cache. As the negative value in the Count_Register increases and becomes zero, the controller knows that the instruction currently being executed is the triggering sbb. If the triggering sbb is not taken, the controller returns to the IDLE state. Otherwise, it enters an ACTIVE state. While in the FILL state, if there is a cof that is not caused by the triggering sbb, the controller also returns to the IDLE state.

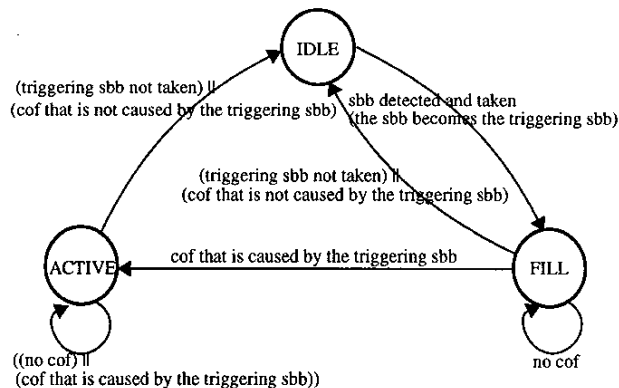


Figure 4. Loop Cache Controller

When in the ACTIVE state, the controller directs all the instruction requests to the loop cache. When in the ACTIVE state, the controller will return to the IDLE state if one of the following two events occurs: (i) the triggering sbb is not taken (the loop sequentially exits through the sbb); or (ii) there is a cof that is not caused by the triggering sbb.

6. Experimental Results

Instruction level simulations were performed using the Powerstone benchmarks (shown in Table 1) to quantify our

caching technique. These benchmarks were compiled to the M•CORE™ ISA [4] using the Diab 4.2.2 compiler. There is no performance degradation associated with this technique. We define the *main cache access rates* as the number of instruction references made to the main cache using a loop cache, divided by the number of instruction references made to the main cache without a loop cache.

Table 1: Powerstone Benchmarks

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

Figure 5 shows the average main cache access rates for the entire benchmark suite, as a function of w . The access rate decreases most dramatically from $w=2$ to $w=5$. This is because the program loops found in these benchmarks were dominated by loops with size of 32 instructions or less. With a 32-entry ($w=5$) loop cache, the main cache access rate is reduced by about 37.9%

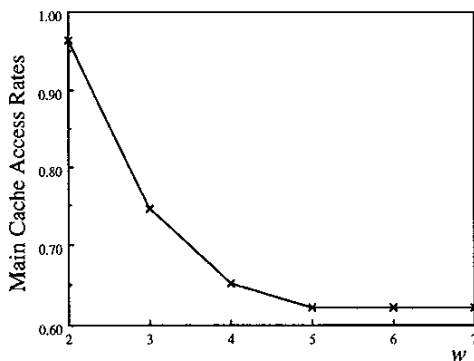


Figure 5. Main Cache Access Rates

7. Summary

Low system cost and low energy consumption are two important factors to consider in designing many embedded

systems. In this paper, 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 [3,5]:

- This technique is based on the definition, detection and utilization of a special class of branch instructions, called the short backward branch instruction (sbb). By definition of a sbb instruction, the size of the program loop that the hardware is trying to capture is guaranteed to be no larger than the loop cache size. Furthermore, the hardware infers from the sbb instruction, how many instructions are actually contained in the loop.
- Our caching scheme does not have an address tag store. The loop cache array can be implemented as a direct mapped array. It does not have a valid bit associated with each loop cache entry. The low area cost associated with the loop cache allows it to be tightly integrated into the microprocessor core. Tight integration not only further reduces the instruction fetch energy, it could also reduce or even eliminate the adverse impact on memory access time due to the presence of the loop cache.
- Unlike many other loop cache implementations, our scheme does not require the program loops to be aligned to any particular address boundary. The software is allowed to place a loop at any arbitrary starting address.
- Lastly, and most importantly, there is no cycle count penalty nor cycle time degradation associated with this technique.

8. REFERENCES

- [1] 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.
- [2] 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.
- [3] L. H. Lee, J. Scott, B. Moyer and J. Arends, "Low-Cost Branch Folding for Embedded Applications with Small Tight Loops," *Int'l Workshop on Compiler and Architecture Support for Embedded Computing Systems*, Washington D.C., December 1998.
- [4] M•CORE Reference Manual, Motorola Inc., 1997.
- [5] B. Moyer, L. H. Lee and J. Arends, "Data Processing System Having a Cache and Method Thereof," *US Patent*, no. 5,893,142, 6th April, 1999.
- [6] 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.

M•CORE is a trademark of Motorola Inc.