Reducing Instruction Fetch Energy with Backwards Branch Control Information and Buffering

Jude A. Rivers, Sameh Asaad, John-David Wellman, and Jaime H. Moreno
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA
{jarivers,asaad,wellman,jhmoreno}@us.ibm.com

ABSTRACT

Many emerging applications, e.g. in the embedded and DSP space, are often characterized by their loopy nature where a substantial part of the execution time is spent within a few program phases. Loop buffering techniques have been proposed for capturing and processing these loops in small buffers to reduce the processor's instruction fetch energy. However, these schemes are limited to straight-line or innermost loops and fail to adequately handle complex loops.

In this paper, we propose a dynamic loop buffering mechanism that uses backwards branch control information to identify, capture and process complex loop structures. The DLB controller has been fully implemented in VHDL, synthesized and timed with the IBM Booledozer and Einstimer Synthesis tools, and analyzed for power with the Sequence PowerTheater tool. Our experiments show that the DLB approach, on average, results in a factor of 3 reduction in energy consumption compared to a traditional instruction memory design at an area overhead of about 9%.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Primary Memory

General Terms

Algorithms, Design, Performance

Keywords

low-power, instruction fetch, loop buffer

1. INTRODUCTION

Many emerging applications, especially in the embedded and digital signal processing space, are characterized as having a loopy nature, where a substantial part of their execution time is spent within a few program phases or loop nests. Loop buffering techniques have been proposed [1, 3] for capturing such instruction loops in small buffers, thus reducing the microprocessors instruction fetch energy by not powering up the primary instruction memory unit when executing these loops. Although these prior schemes have been shown to be effective in reducing the energy consumed for some applications, they are often limited by either the method of loop implementation and/or the technique of loop identification, capture and buffer management.

Some schemes (e.g. [3]) employ pure hardware dynamic identification, capture and loop management while others depend on instruction profiling (e.g. [1]) for static preloading of identified loops, or compiler support for dynamically loading targeted loops. Profiling and compiler support based methods tend to be application/platform specific, and not general enough for portability across different microprocessor platforms and instruction set architectures (ISAs). A dynamic hardware loop caching scheme like the one proposed by Lee et al. (henceforth referred to as the LMA scheme [3]), though general and portable across microprocessor platforms, is limited in its ability to capture various complex loop structures like nested loops with if-then-else constructs and/or exception condition checks. The real benefit of dynamic loop buffering is yet to be fully realized.

As computational devices grow more pervasive, and the roles of portable, battery-powered devices keeps expanding, there is an increasing focus on issues that affect battery life. Further, even in the traditional realm of desktops and servers, there is a growing focus on controlling both the amount of energy consumed and the resulting power/heat dissipated, particularly in order to control packaging/cooling requirements.

There is therefore the need for future systems to provide sufficient processing power to efficiently execute emerging workloads at the lowest possible energy consumption levels. Hence, an instruction fetch scheme is necessary that efficiently, economically, and dynamically exploits instruction fetch redundancy to save energy.

1.1 Previous Work

Various methods for reducing instruction fetch energy consumption of loopy workloads have been proposed. Significant among them include dynamic approaches, like the LMA loop cache [4, 3] and the filter cache [2], and static approaches like the pre-loaded loop buffer [1]. Each of these approaches offers some benefits but also have some limitations.

In [4, 3], Lee et al. describe a simple loop buffering scheme that dynamically identifies, captures, and manages a small loop body during processor execution. The loop body, cap-

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.
ISLPED'03, August 25–27, 2003, Seoul, Korea.
Copyright 2003 ACM 1-58113-682-X/03/0008 ...$5.00.

tured in a small tagless memory unit, then feeds the processor instruction stream directly if the loop is subsequently retaken after the capture. The processor is able to power-down the primary instruction memory unit during the cycles where the instruction stream is fed from the loop buffer, thus providing a significant savings in instruction fetch energy.

While the LMA loop cache scheme does reduce the average instruction fetch energy, there are limitations on its applicability. The scheme fails to capture loops which contain internal branches (e.g. an if-then-else construct within the loop body) and loops which do not use a single loop-ending branch but instead have a number of separate tail-sections which all return to the same loop start. Many ISAs without predicated instructions tend to form loops with internal branches. In addition, the LMA controller may not capture loops longer than the loop cache. Our analysis [6] of the LMA scheme on various workloads (not shown here) demonstrate that many very loopy codes do not benefit enough, especially loops with more complex inner bodies and longer larger loop nests.

Gordon-Ross et al. [1] exploits the fact that some embedded systems are designed to run a single application. Taking advantage of workload-specific information in the hardware design, a mechanism is provided for the user or compiler to specify a set of loop bodies to be pre-loaded into a loop buffer. These loop bodies will remain resident, will include all the executable code for the loop body, and thus can always be used to serve the execution of that loop. This has several advantages, including the ability to avoid the initial dynamic detection overhead, and the ability to capture more complex types of loops.

Another approach which indirectly captures loop bodies is the filter cache [2]. A filter cache is simply a tiny level zero cache located between the processor and the primary instruction cache memory. When a relatively small footprint sees repeated use, such as a small loop body, the filter cache can serve the accesses and avoid lookups in the main instruction cache. This allows for arbitrary loop bodies to be captured, but does have the drawback of the overheads associated with a cache, e.g. the tag arrays, etc. and the energy cost of cache pollution.

Our focus is on exploiting a dynamic hardware mechanism that uses behavior information to reduce the fetch energy of arbitrary loopy workloads. We introduce a scheme along the lines of the LMA approach and do not further consider other cache design issues.

2. THE DYNAMIC LOOP BUFFER

Our work proposes an improved dynamically-managed loop buffer (DLB). The DLB structure consists of a loop buffer memory structure, where upon detection of a loop, copies of the loop instructions are dynamically filled into the loop buffer, which is then used to feed subsequent matching fetch requests. The DLB controller dynamically detects and captures loop nests (i.e. multiple levels of looping), loops with complex internal control-flow (e.g. loops which include conditional code within the loop body) and portions of loops that are too large to fit completely in a loop buffer.

2.1 The Loop Buffer Memory

The loop buffer memory consists of three elements. The first is a small, tagless buffer array structure which can be randomly accessed. This structure, which may be built out of low-power register arrays, sits in parallel with the SRAM memory arrays of the primary instruction memory unit. The next element consists of two registers that record the range of the captured loop (i.e. the loop start and end addresses). These registers determine when addresses fall within the captured loop range. The final element is a set of bits (forming part of the DLB controller logic) used to indicate whether a loop buffer entry is valid/filled or not. This allows the loop buffer memory to contain unfilled slots, providing the ability to capture more complex loop bodies and structures.

2.2 The DLB Controller

The DLB controller is a finite state machine that provides more sophisticated utilization of the loop buffer memory. The DLB state transitions are as shown in Figure 1.

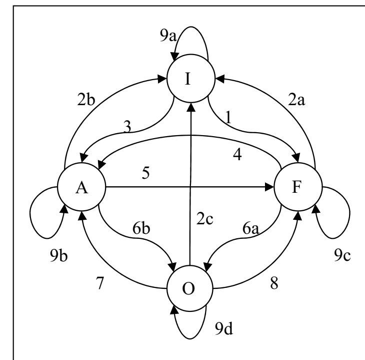

Figure 1: The DLB Controller

The DLB controller is a four state finite state machine. In addition to similar Active (A), Idle (I) and Fill (F) states as used in the LMA controller, there is an Overflow (O) state. The Overflow state allows the controller to identify, capture, and manage portions of a loop that is too large to completely fit within the buffer. The state transitions illustrated in Figure 1 are further described in Table 1.

Execution begins in the Idle state, and remains there until an appropriate transition is triggered. Upon initial detection of a backward branch (that is also reasonably likely to be a loop ending branch), the DLB controller declares a new loop, records the address of the branch as the current Loop_End and the target address as the current Loop_Start, and transitions to the Fill state. Loop_Start and Loop_End then defines the current DLB loop range.

The DLB controller will continue to fill the loop buffer with copies of subsequently fetched instructions until:

- execution moves outside the current DLB loop range, transitioning to Idle
- execution returns to a previously filled address within the DLB loop range, transitioning to Active
- execution moves beyond the physical size of the buffer, transitioning to Overflow

When in Active state, the primary instruction memory unit is powered-down and the loop buffer supplies requested instructions. The controller remains in the Active state until execution either moves out of the DLB loop range (to Idle), beyond the physical buffer size (to Overflow) or into a previously unfilled area of the loop buffer (Fill). In the Overflow state, requests are forwarded to the primary instruction

ID	From-To	Condition
1	I - F	[(backward branch(bb) detected & taken) OR (move within loop range & entry invalid)]
2a 2b 2c	F - I A - I O - I	[(bb not taken) OR (another cof causes move outside loop range)]
3	I - A	(move into loop range & entry valid)
4	F - A	[(bb is taken again) OR (another indented bb is taken)] & (entry valid)
5	A - F	[(resume filling rest of loop) OR (move within loop range & entry invalid)]
6a 6b	F - O A - O	(end of physical loop buffer reached)
7	O - A	[(cof caused within loop range) OR (bb detected and taken again)] & [(next-data within buffer range) & (entry valid)]
8	O - F	[(cof caused within loop range) OR (bb detected and taken again)] & [(next-data within buffer range) & (entry invalid)]
9a 9c 9d	I - I F - F O - O	(no cof)
9b	A - A	(no cof) & (entry valid)

Table 1: State Transitions for DLB

memory while monitoring for a transition back into within the buffer physical range.

The DLB controller is stateful, in that it keeps sufficient system state and history for loop detection and management. In addition to the Overflow state, the DLB approach uses transition 3 to avoid being a "capture, use, and destroy" technique like the LMA. Transition 3 allows for future reuse of abandoned loops that are revisited before they are overwritten.

3. SYSTEM EVALUATION AND ANALYSIS

To accurately estimate the effectiveness, power, area and timing characteristics of the proposed DLB as well as the LMA, the two controllers were implemented in VHDL and synthesized using the IBM 0.13 um ASIC library as a target. Each controller was implemented to intercept the path between the processor core and the primary instruction memory and, based on the algorithm, decide whether to forward the instruction fetch request to the memory or to block and serve the request from the local loop buffer. To ensure no additional delay cycles, we designed for this decision to happen within the first half cycle from the start of the access. The design operated at 450 MHz using a nominal supply voltage of 1.5V.

A new low-power DSP microprocessor, *eLite*[5], being developed at the IBM TJ Watson Research Center was the native core used in this evaluation. *eLite* fetches 8 byte long instruction words (LIWs) per cycle. The 32 Kbytes flat instruction memory was organized as four banks. The memory banks were each implemented with "compilable" 1K entries x 64 bits wide SRAM arrays, and the loop buffer with a 64 entry by 64 bits dual ported growable register array (GRA), all from the IBM technology library.

3.1 Design Entry and Verification

We constructed a VHDL testbench to simulate each of the controllers. The testbench consisted of the controller unit under test connected from one side to a model of the primary instruction memory unit and from the other side to a driver process that simulates the instruction fetch behavior of the processor as shown in Figure 2. We verified the correctness of the implementation through numerous validation simulations at the RTL level. All instructions returned to the processor (either from the main instruction memory or the loop buffer) were verified for correctness.

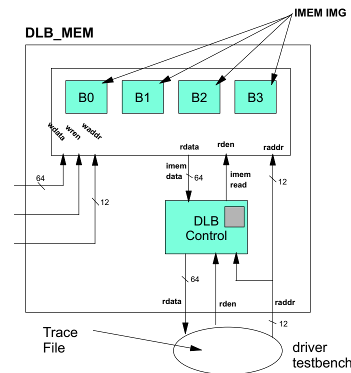

Figure 2: VHDL Testbench

The benchmarks used in this evaluation are typical signal processing functions written for the *eLite*[5]. They included: viterbi decoding (viterbi), block finite impulse response filter (bkfir), fast fourier transfom (fftr64a), huffman decoding (huffman), infinite impulse response filter (iir) and double precision finite impulse response filter (dpfir).

We ran RTL level simulations to examine the ability of the two schemes to detect, capture and manage loops. Figure 3 shows the state occupancy distribution of the DLB (right bar) and LMA (left bar) controller schemes for a 64-entry loop buffer across the six benchmarks. On the average, the DLB controller is in the Active state 83% of the time, 4% in the Fill state, and 12% in Idle. The LMA controller, however, stays in Active 56% of the time, 21% in Fill, and 23% Idle. The fact that the DLB operates more frequently in the Active state and substantially less in both Fill and Idle states shows that these simple workloads possess various complex loop formations which the LMA controller fails to adequately handle.

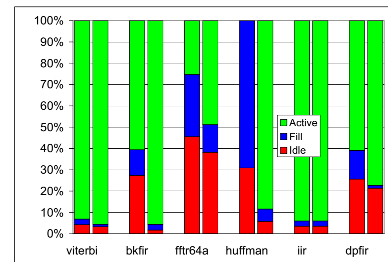

Figure 3: State Occupancy Distribution

3.2 Synthesis, Timing and Power Estimation

Each of the controllers was synthesized using IBM Booledozer synthesis tool and timed using IBM Einstimer static timing analysis tool. Synthesis results are shown in Table 2. The resulting gate level netlist was saved in Verilog format. As expected, the loop buffer schemes result in extra area overheads, adding on 57K cells for the DLB and 52.4K cells for the LMA. Even though the DLB scheme maintains adequate state information as compared to the LMA, the logic area difference among them is close to negligible.

Area Estimates (KCells)	*DLB*	*LMA*
SRAM Bank (SRAM1DN 1Kx64)	151	151
Decode Logic and Wrapper	37	37
Total Primary Memory (4 Banks + Decode Logic)	641	641
Controller	10	5.4
Loop Buffer (GRA 64x64)	47	47
Total Added Area	57	52.4
Area Overhead	9%	8%

Table 2: Area Estimates for the DLB and LMA

To estimate power, we modified our VHDL testbench slightly to instantiate the synthesized gate level description of the corresponding controller. A monitor process was also included in the testbench to monitor gate switching activity of the design. We ran our benchmarks to record switching activity.

PowerTheater (a commercial power estimating tool from Sequence Design) was used for power estimation. The tool takes as input the gate-level design netlist, the switching activity file, and a library of power models that characterizes the power consumption for the various gates in the ASIC library. The latter came from the IBM ASIC design kit along with power models for the SRAMs. Since the design was pre-layout, we configured the tool to estimate wire capacitances based on the number of cells in the design and also estimate the clock tree power based on the number of latches, the type of clock buffers and the allowable fanout limits for each level in the clock tree.

4. RESULTS

Figure 4 shows the power dissipation of the instruction memory subsystem. For each benchmark, the right column depicts the base power, which is the power without any loop buffer in place. The middle column shows the power using the LMA controller, and the left column shows the same for the DLB controller. First, the average power for the base system remains constant across the different benchmarks. This is due to the fact that the power model of our SRAM is not sensitive to data variations. In other words, the ASIC library assumes the same power for a read access regardless of the actual bit values being read. In all but the iir benchmark, the DLB controller dissipates less power than the LMA since it can capture more complex loop structures, as Figure 5 suggests. For the iir, a quick visual inspection of the assembly code reveals simple straight loop structures that are easily handled by the LMA, and since the LMA controller has less state than the DLB, it results in slightly less power. On the other hand, the huffman benchmark presents a classic scenario where the LMA controller dissipates more power than the base memory structure without a loop buffer. This is due to the complex nature of the loops in the huffman benchmark which causes the LMA controller to toggle between the "Idle" and "Fill" states while never trapping into the "Active" state, as shown by the zero LMA buffer hit rate for huffman in Figure 5.

Our DLB scheme shows, on the average, a factor of 3 improvement on energy consumption reductions over a traditional primary instruction memory, which in this case is a flat memory built out of SRAMS. The DLB scheme also improves upon the LMA technique's energy reduction by more than 50%.

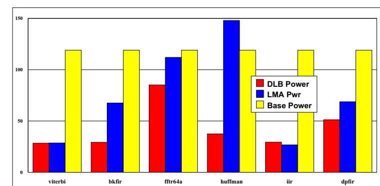

Figure 4: Power Dissipation

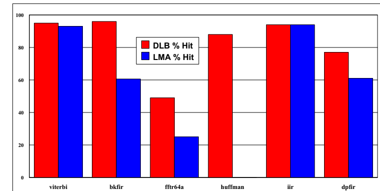

Figure 5: Hit Rate Comparison

5. CONCLUSION

Power considerations are becoming increasingly important in computer system design. Many applications, including multimedia, signal processing and high-performance computing, exhibit significant looping behavior. This observation leads one to consider reducing the instruction fetch power by the introduction of small, low-power mechanisms to capture loop code and feed processing elements from that low-power path.

Previous work on exploiting looping behavior for fetch energy reductions are limited in the types and kinds of loop structures that can be captured for low-power buffering. We have presented an improved dynamically-managed loop buffering mechanism which addresses most of these limitations. Our proposed dynamic loop buffer controller is stateful and intelligently managed and has the ability to capture the vast majority of simple and complex loop structures in today's emerging loopy workloads.

Our DLB scheme shows, on the average, a factor of 3 improvement on energy consumption reductions over a traditional primary instruction memory built out of SRAMS. The DLB scheme also improves upon the loop cache technique of Lee et al. by more than 50% on average.

6. REFERENCES

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

[2] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. In *International Symposium on Microarchitecture*, pages 184–193, December 1997.

[3] L. H. Lee, W. Moyer, and J. Arends. Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops. In *International Symposium on Low Power Electronics and Design (ISLPED)*, August 1999.

[4] L. H. Lee, W. Moyer, and J. Arends. Low-Cost Embedded Program Looping Cache - Revisited. Technical Report CSE-TR-411-99, University of Michigan, December 1999.

[5] J. H. Moreno et al. An innovative low-power high-performance programmable signal processor for digital communications. *IBM Journal of Research & Development*, 47(2/3):299–326, March/May 2003.

[6] J. A. Rivers, S. Asaad, J.-D. Wellman, and J. H. Moreno. Reducing Instruction Fetch Energy Through Dynamic Loop Buffering. Technical report, IBM TJ Watson Research Center, January 2003.