

Instruction Fetch Energy Reduction Using Forward-Branch Bufferable Innermost Loop Buffer

Bin-Hua Tein, I-Wei Wu, and Chung-Ping Chung

Dept. of Computer Science, National Chiao Tung University

1001 Ta Hsueh Road, Hsinchu, Taiwan 300, ROC

binhua.cis93g@nctu.edu.tw, gis93808@csie.nctu.edu.tw, cpchung@cs.nctu.edu.tw

Phone: 886-3-5731828 ext:54741 Fax: 886-3-5724176

Abstract—Recently, several loop buffer designs have been proposed to reduce instruction fetch energy due to size and location advantage of loop buffer. Nevertheless, on design complexity dictates most loop buffer designs to store only innermost loops without forward branch or instructions within innermost loops before a forward branch. While program modeling shows that typical programs can best be represented with a simple loop model, many of them contain forward branches in their innermost loops. For example, MiBench spends 71% of execution time on innermost loops, and 27% of these innermost loops consist of forward branch(es). Hence, existing designs lead to limitation in reduction of instruction fetch energy. We propose a simple and effective way to cope with this complexity: since using BTB is a norm in most designs, if we add an extra bit in BTB, indicating if the loop buffer stores the fall-through or target trace after a within-the-innermost-loop forward branch, then much of the complexity can be avoided. Results with MiBench indicate that up to 14.1% of further reduction in instruction fetch energy, and only 1.8% hardware overhead in BTB, is introduced compared with the design without forward branch handling.

Index Terms— low power, loop buffer, instruction cache, instruction fetch.

I. INTRODUCTION

Power consumption has become an increasingly greater concern in digital system designs, especially for battery powered devices. Among all power components within such a system, memory access energy contributes the largest portion. In many researches [1-4], loop buffer has been proposed to reduce instruction fetch energy. A loop buffer is a memory located between CPU core and L1 instruction cache, called IL1 hereafter, as shown in figure 1. CPU core

fetches instructions from either IL1 or loop buffer. Due to its limited, a loop buffer can provide instructions to CPU core at a very low energy level. And the best pieces of code to be placed in a loop buffer will be innermost loops, since their executions tend to repeat many times. As an evidence, MiBench spends 71% of execution time on innermost loops.

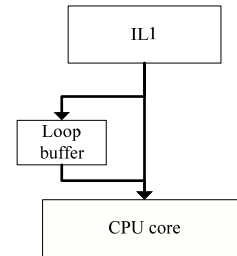


Fig 1. Organization of CPU core, IL1, and loop buffer

To maximize the energy advantage and avoid design complexity, most loop buffer designs are capable of storing only innermost loops without forward branch [1, 2] or instructions within innermost loops before a forward branch [2]. Since many applications consist of forward branch(es) in their innermost loops, utilization of loop buffer and reduction in instruction fetch energy in [1, 2] is limited. To increase utilization of loop buffer, [3, 4] propose a loop buffer consisting of an additional address generator to store any kinds of code segment. Before fetching instruction from loop buffer, loop buffer controller must generate an address and use this address to determine whether this instruction has been stored in loop buffer and if it is, where it is located. Consequently, this address generator leads to a significant increase in energy and fetch latency. In addition, most designs [1, 3, 4] require compiler help to insert special instruction(s) in program to start filling instructions into loop buffer [1, 4] or to determine which code segments should be stored in loop buffer [3]. Recompile and code compatibility issues hence arise.

To increase utilization of loop buffer without introducing much overhead, we use BTB to assist loop buffer in storing the innermost loops with forward branch(es). In our design,

an extra bit is added in branch target buffer (BTB) to record forward branch outcome. This bit indicates whether the loop buffer stores the fall-through or target trace after a forward branch. Notice also that different from previous designs [1, 3, 4], our approach does not need special branch instruction or compiler to assist loop buffer controller in innermost loop detection. Results with MiBench indicate that our design can further reduce 14.1% and 7.7% instruction fetch energy compared with [1] and [2], respectively. And only 1.8% of hardware overhead in BTB is introduced in our design.

The paper is organized as follows: Section 2 examines related the previous work. We present our proposed loop buffer design in Section 3. The simulation results and discussion are presented in Section 4. And finally, Section 5 concludes the paper.

II. RELATED WORK

Using loop buffer to reduce instruction fetch energy has been addressed in many researches [1-4]. [1] is capable of storing only innermost loops without forward branch. To indicate where a backward branch exists, [1] uses a special branch instruction “sbb”. If a “sbb” is detected and taken, loop buffer controller starts to fill instructions into loop buffer. Only if a “sbb” is detected and taken twice successively, CPU core starts to fetch instructions from loop buffer until this “sbb” is detected and not taken. To reduce design complexity, [1] uses a counter to generate loop buffer addresses, called loop buffer program counter (LPC). Consequently, this causes that [1] is only capable of storing a loop without any forward branch so that utilization of loop buffer and the reduction in instruction energy are limited.

Instead of using a special branch instruction “sbb” in [1], [2] deploys a special register to record the address of backward branch. Once a backward branch is detected and taken twice successively, loop buffer controller starts to fill instructions into loop buffer. After successively filling, CPU core begins to fetch instructions from loop buffer. We also use this method to detect an innermost loop in this paper. Unlike [1], [2] can also store instructions within an innermost loop before a forward branch. This is because instruction addresses before a forward branch are sequentially. To reduce design complexity, [2] also uses a counter to generate LPC so that [2] encounters the same limitation with [1].

[3] is capable of storing any kinds of loop and subroutine call. In [3], which code segment can be stored in loop buffer are analyzed statically. After the CPU booting, several code segments and their start address (start_addr) and end address (end_addr) are filled into loop buffer and special registers, called loop address registers (LARs), respectively. CPU core then continuously compares each instruction address with LARs to determine whether start and terminate to fetch instructions from loop buffer. This leads to inflexible usage of loop buffer. Since code segments stored in loop buffer may consist of forward branch, [3] uses an address generator

to cope with non-sequential instruction fetch. Before fetching one instruction from loop buffer, loop buffer controller must wait for LPC calculated by address generator and use LPC to determine whether this instruction has been stored in loop buffer and where this instruction is. Consequently, this address generator leads to a great increase on instruction fetch latency and hardware cost.

To dynamically fill code segments into loop buffer, [4] uses a special instruction “lbon n ”, where n is number of instruction should be filled into loop buffer, to indicate where start to fill instructions into loop buffer. [4] also uses the same method to calculate LPC so that [4] encounters the same difficulties with [3]. Except for [2], all designs [1, 3, 4] use compiler to assist in loop detection. This causes that program(s) must be recompiled in order to execute on a CPU containing one of these designs.

III. OUR APPROACH

3.1 Architecture of our approach

The architecture of our approach is illustrated in figure 2. The main components include loop buffer, loop buffer controller, P-bit in BTB, and a multiplexer. Loop buffer is a small and tagless memory so that it enables tight integration with the CPU core and very low power per access. Each entry in loop buffer only stores one instruction. Loop buffer controller is responsible for: (1) determining that the requested instruction should be fetched from loop buffer or IL1; (2) creating or updating P-bit in BTB; (3) determining when start or terminate to fill instructions into loop buffer; (4) innermost loop detection. According to the control signal from loop buffer controller, multiplexer decides that a requested instruction should come from loop buffer or IL1. P-bit records forward branch outcome, such as taken or no taken, of an innermost loop during filling instructions into loop buffer. It assist loop buffer controller in determining the requested instruction should be fetched from loop buffer or IL1.

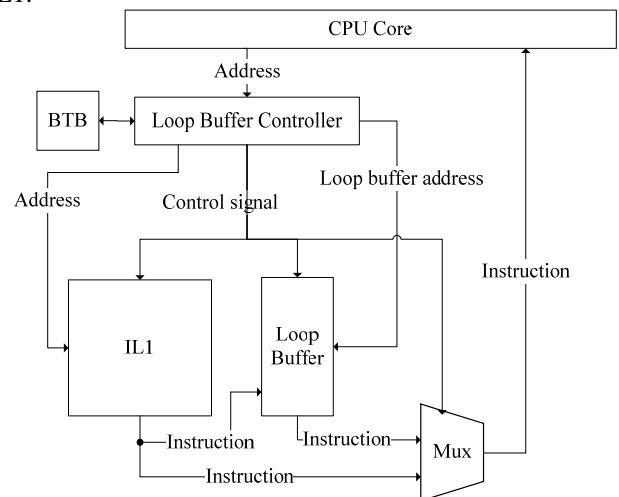


Fig 2. Architecture of our approach

3.2 Operation

The operation mechanism, consists of three states: IDLE, FILL and ACTIVE, is similar with [1, 2]. In the figure 3, the gray rectangle and the solid black line are currently accessing block and bus respectively during different states.

The state diagram of the loop buffer controller's finite state machine (FSM) is shown in the figure 4. When CPU core initializes or resets, loop buffer controller enters IDLE state first. During IDLE state, loop buffer controller continuously detects whether exist an innermost loop in program, as action A in figure 4. Action B is taken if and only if an innermost loop has been detected and this innermost loop does not have been stored in loop buffer. To determine whether an innermost loop exists in loop buffer, we use a special register, called S_addr, to record the start address of an innermost loop been stored in loop buffer. If the start address of an innermost loop is not same with S_addr, this innermost loop does not exist in loop buffer. Otherwise, loop buffer controller would enter ACTIVE state, as action C in figure 4. Noticeable, since comparison would cause one cycle delay, CPU core will start to fetch instructions from loop buffer in next two cycles.

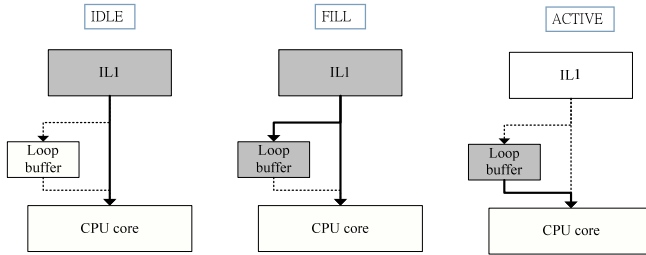


Fig 3. Memory accessing in different states

Two innermost loop detection strategies are employed in our design. First, a backward branch is detected and taken. Second, a backward branch is detected and taken twice successively. Since first loop detection strategy is more aggressive than second one, it enters ACTIVE state in advance and increases the utilization of loop buffer. But first strategy may cause higher energy consumption due to higher mis-detection. The performance simulation of both strategies is shown later.

During FILL state, instructions are sequentially filled into loop buffer from first entry, as action D in figure 4. In the meanwhile, the branch predicted result of each forward branch stored in loop buffer and the start address of loop are also recorded into P-bit and S_addr respectively. After successfully filling all instructions of an innermost loop, loop buffer controller enters ACTIVE state, as action E in figure 4.

Two situations would let loop buffer controller return to IDLE state. First, loop buffer is full caused by a BIG loop which the size of an innermost loop is larger than loop buffer's capacity, as action F in figure 4. To cope with BIG loop, loop buffer only stores M instructions of BIG loop. M is the size, i.e. number of entry, of loop buffer. Second, a branch misprediction occurs and branch predictor changes

its state from strongly taken/not-taken to weakly taken/not-taken, as action G in figure 4. Since the branch misprediction must be resolved at execution stage of pipeline, several instructions on another execution flow has been filled into loop buffer. Those instructions may result in the wrong execution of program. To repeat filling instructions before a miss-predicted forward branch, the loop buffer controller can count back P entries from current position to refill instructions located at another control path. P is the number of pipeline stage between instruction fetch (IF) and execution stage (EXE). To avoid ping-pong effect, i.e. a branch outcome changes between taken and not-taken iteratively, the loop buffer controller only refills instructions only if branch predictor changes its state from weakly taken/not-taken to strongly taken/not-taken, as action H in figure 4.

During ACTIVE state, CPU core fetches instructions from loop buffer instead of IL1, as action I in figure 4. Loop buffer miss occurs when P-bit is different with the branch predictor's result, i.e. branch predictor has changed its direction prediction of this forward branch. There are two actions to handle loop buffer miss, are action J and M in figure 4. To avoid unnecessary refilling, action M only takes when branch predictor changes its state from weakly taken/not-taken to strongly taken/not-taken. Otherwise, we use action J to handle loop buffer miss. In action M, loop buffer controller returns to FILL state and only refills instructions located on another execution flow. When CPU core has already fetched last instruction of loop buffer (action L in figure 4) or loop buffer controller encounters a branch misprediction (action K in figure 4), loop buffer controller enters IDLE state. First condition occurs when loop buffer encounters a BIG loop. Since loop buffer dose not store a whole BIG loop, only partial instructions can be stored in loop buffer. After CPU core has fetched the last instruction of loop buffer, loop buffer controller must enter IDLE state to let CPU core fetch other instructions of BIG loop from IL1. The second condition, a branch misprediction occurs, indicates that the execution flow has changed and instructions located on another execution flow do not exist in loop buffer.

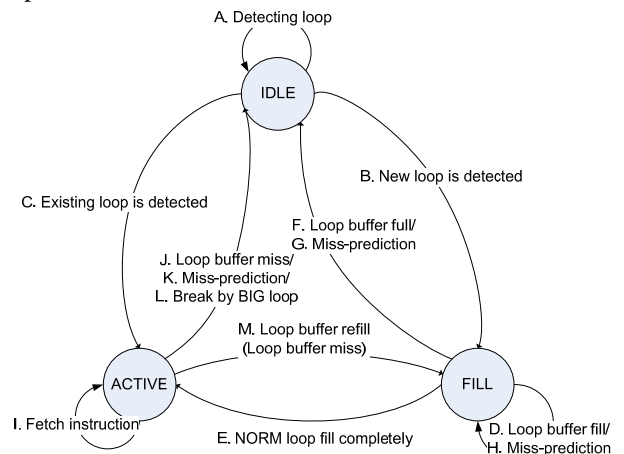


Fig 4. State diagram of loop buffer controller

An example is introduced to explain our design in detail. Figure 5 shows a code segment which is an innermost loop with one forward branch. Each column, C1~C9, represents one assemble code. Here, we make several assumptions: (1) C3 is always taken during the first ten iterations, afterward C3 is not taken; (2) loop buffer controller enters FILL and ACTIVE state at second iteration and third iteration, respectively; (3) one-bit predictor is used; (4) loop buffer is capable of storing whole code segment depicted in figure 5.

During third to tenth iteration, the P bit is same with branch predictor’s result so that loop buffer controller stays in ACTIVE state, i.e. CPU core fetches instructions from loop buffer instead of IL1. At eleventh iteration, the program changes its execution flow and a misprediction occurs at this iteration. This leads loop buffer controller to entering IDLE state. CPU core therefore fetches instructions from IL1. At begin of twelfth iteration, loop buffer controller still returns to ACTIVE state due to the start address of loop is same with S_addr. But when CPU core fetches C3, loop buffer controller detects that the result of P bit is not same with branch predictor’s result and then enters FILL state. Hence, the following instructions of C3 are refilled and P-bit is also updated. At thirteenth iteration, CPU core fetches instructions from loop buffer until a backward branch misprediciton occurs.

C1	L:
C2	
C3	bne	A
C4	
C5	
C6	A:
C7	
C8	
C9	bne	L

Fig 5. An example code segment

IV. PERFORMANCE SIMULATION

4.1. Energy Model

The instruction fetch energy (E_{IF}) per fetch dissipated by the loop buffer and lower level instruction memories can be expressed by the equation (1):

$$E_{IF} = E_{IC} * R_{IC} + E_{LB} * R_{LB} \quad (1)$$

where E_{IC} and E_{LB} are fetch energy of lower level instruction memories and loop buffer respectively, R_{IC} and R_{LB} are access ratio of lower level instruction memory and loop buffer respectively. In this paper, R_{IC} (R_{LB}) is defined as the ratio of number of instruction fetch from of lower level instruction memory (loop buffer) to total number of instruction fetch, as expressed by following equation (2): Since E_{IC} is not affected by operation strategy and size of

loop buffer, we assume E_{IC} as a constant in this paper. E_{LB} depends on the size of loop buffer and hardware overhead of loop buffer controller. In addition, the size of loop buffer has also a great effect on R_{LB} .

According to equation (1), R_{IC} and R_{LB} dominate E_{IF} . If we enlarge R_{LB} without introducing a large increase on R_{IC} and E_{LB} , the instruction fetch energy (E_{IF}) can be reduced.

4.2. Simulation Environment

We use SimpleScalar/ARM simulator [5] and MiBench benchmark [6] to evaluate each design. SimpleScalar is an execution-driven simulator and used to simulate modern processor architectures. MiBench is a set of 35 embedded applications for benchmarking purposes. The power consumption evaluation of loop buffer, BTB and IL1 is based mainly on the Wattch [7] power modeling tool. XTREM is a framework for architectural-level power analysis. We experimented with the different sizes of loop buffer, included 64, 128, 256, 512, 1024 and 2048 bytes (B), in this paper. Other parameters used in SimpleScalar are shown in the table 1. In this simulation, R_{IC} and R_{LB} are calculated by SimpleScalar, and E_{LB} and E_{IC} are calculated by Wattch.

TABLE I
PARAMETERS SETTING IN SIMPLESCALAR

Parameter	Value
Loop buffer	64B, 128B, 256B, 512B, 1KB and 2KB
IL1	8KB, direct-mapped, 32B line
Branch predictor	Bimodal
BTB	512-set, 4-way

4.3. Simulation Result

Four loop buffer designs, included [1], [2] and our approach with different instruction fill strategies, are evaluated in this paper. DLC and 2-way DLC are loop buffer designs proposed in [1] and [2], respectively. DLC is capable of storing innermost loops without forward branch. Except innermost loops without forward branch, 2-way DLC is also capable of storing instructions within innermost loops before a forward branch. HCLB-1 and HCLB-2 are our proposed designs and they are difference in instruction fill strategy. In HCLB-1, loop buffer controller starts to fill instruction once a backward branch is taken. HCLB-2 is less aggressive than HCLB-1, loop buffer controller only starts to fill instruction after a backward branch is taken twice successively.

Figure 6 and 7 shows simulation results of the MiBench benchmark for the different loop buffer designs. We examined total loop buffer sizes ranging from 64 to 2048 bytes, i.e. 16 to 512 instructions. Our results shows that HCLB-1 and HCLB-2 averagely further increase R_{LB} by

$$R_{IC}(R_{LB}) = \frac{\text{number of instrucion fetch from lower level memory (loop buffer)}}{\text{total number of instruction fetch}} * 100\% \quad (2)$$

27.6%/19.8% and 16.7%/9.0%, respectively ([1]/[2]). Noticeable, since operation mechanism of each design is similar, both loop buffer and IL1 are accessed during filling instructions into loop buffer so that the sum of R_{LB} and R_{IC} would exceed over 100% in each designs.

Both HCLB-1 and HCLB-2 can store the same percentage of innermost loop into loop buffer. But since HCLB-1 can let the CPU core start to fetch instructions from loop buffer one cycle in advance, R_{IC} of HCLB-1 is smaller than HCLB-2. On the other hand, HCLB-2 has higher accuracy in loop detection. This results in less unnecessary access in HCLB-2 so that R_{LB} of HCLB-2 is smaller HCLB-1.

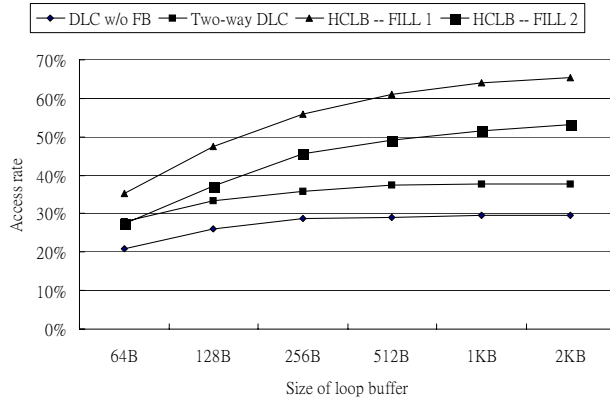


Fig. 6. Access ratio of loop buffer of different designs

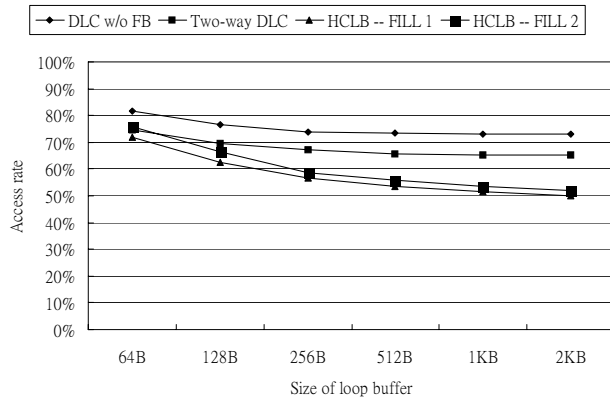


Fig. 7. Access ratio of IL1 of different designs

According Wattch power modeling tool, the ratio of E_{LB} and E_{IC} is shown in table 2. Here, E_{LB} also takes loop buffer controller into account. The loop buffer controller is synthesized with Synopsys design tools. In order to understand the cost of adding a loop buffer controller to a loop buffer, we also synthesize a 256 bytes loop buffer. The gate count ratio of loop buffer controller and a 256 bytes loop buffer is about 83.33%. Since loop buffer controller of our approach is very similar with [1, 2], we use 83.33% as the energy consumption ratio of loop buffer and loop buffer controller in each design. Each entry in BTB mainly consists of 3 fields which are instruction address (about 20-bit), branch target address (32-bit) and branch direction prediction (1~2-bit). Since P-bit is only 1-bit, the hardware

overhead is about 1.8% so that the area overhead introduced by P-bit can be negligible.

TABLE II
RATIO OF E_{LB} AND E_{IC}

Size of loop buffer	64B	128B	256B	512B	1KB	2KB
$(P_{LB}/P_{IC})^*$	6.91	7.44	8.65	11.53	18.8	34.23
100%						

The reduction in instruction fetch energy of different designs is shown in the figure 8. We see that for our design of 1KB (HCLB-1) and 512B (HCLB-2) has the maximum energy reduction. Compared to [1, 2], HCLB-1 achieves more than 14.1% [1] and 7.7% [2], and HCLB-2 achieves more than 12.7% [1] and 6.3% [2].

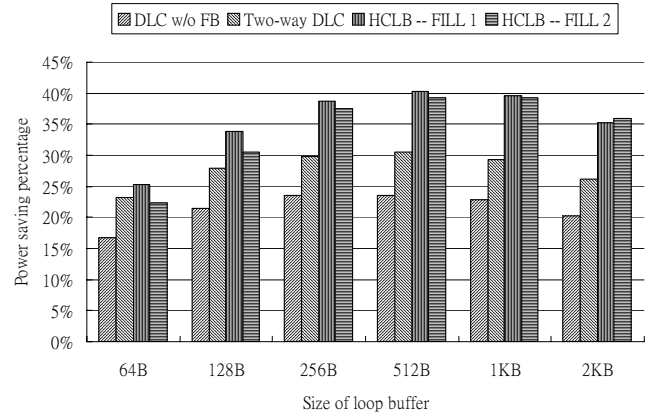


Fig. 8. Reduction in instruction fetch energy of different design

V. CONCLUSION AND FURTHER WORK

Adding our design to the instruction memory hierarchy can significantly reduce instruction fetch energy. Previous designs, although simple, fail to capture a large percentage of innermost loops, make energy saving results unsatisfactory. We propose to correct this shortcoming by using BTB to assist loop buffer in storing innermost loops. The benefits of our design are: (1) significant increase in loop buffer utilization and hence instruction fetch energy saving; (2) almost negligible hardware overhead; (3) no need for extra branch instruction or compiler to assist loop buffer controller in innermost loop detection. Experiment results show that our design can further reduce up to 14.1% of instruction fetch energy, and only introduce 1.8% of hardware overhead in BTB.

Increasing utilization of loop buffer has another significance: it provides better changes for the lower level instruction memories to conserve static energy leakage. With the advances in deep-sub-micro semiconductor processing, static energy consumption is becoming dominant. Several researches show that the static energy currently accounts for about 15%-20% of the total energy consumption in the 130 nano process, and will exceed 50% in the 65 nano process. Although we only address reduction in instruction fetch

energy, i.e. dynamic energy, in this work, it will be an interesting if we study the static energy effect of our design. We are already working on this topic.

REFERENCES

- [1] L. Lee, B. Moyer and J. Arends, "Low-Cost Embedded Program Loop Caching – Revisited," University of Michigan Technical Report CSE-TR-411-99, 1999.
- [2] T. Anderson and S. Agarwala, "Effective hardware-based two-way loop cache for high performance low power processors," *International Conference on Computer Design*, 2000.
- [3] A. Gordon-Ross, S. Cotterell and F. Vahid, "Tiny Instruction Caches For Low Power Embedded Systems," *ACM Transactions on Embedded Computing Systems*, 2003.
- [4] M. Jayapala, F. Barat, T. V. Aa, F. Catthoor, H. Corporaal and G. Deconinck, "Clustered Loop Buffer Organization for Low Energy VLIW Embedded Processors," *IEEE Transactions on Computers*, 2005.
- [5] T. Austin, E. Larson and D. Ernst, "SimpleScalar: an Infrastructure for Computer System Modeling," *IEEE Computer*, 2002.
- [6] D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *IEEE 4th Annual Workshop on Workload Characterization*, 2001.
- [7] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *ISCA*, 2000.