Microarchitecture Support for Interconnect Power-aware Instruction Permutation

Hui Lin, Md. Sajjad Rahaman and Masud H Chowdhury
ECE Department, University of Illinois at Chicago, Chicago, IL 60607, USA
Email: {hlin, mrahaman, masud}@ece.uic.edu

***Abstract*—This paper proposes a new instruction permutation encoding mechanism to deal with power dissipation issue on interconnect bus between processor and memory. When program executable code is loaded into memory, instructions are re-scheduled, block by block, into a power efficient format. As a result, on cache miss when instruction block from memory is required, power consumption during transmission of instruction between memory and processor end on interconnect bus is reduced. Reorder operation does not require any encoding/decoding circuitry. In order to restore instructions into its original order, an index table is employed with a reasonable area, power and access time overhead. Results based on the SPEC CPU2000 benchmark suit show that self- and coupling capacitance switching is reduced by 40% and 30%, respectively, with an area overhead of 14% at 65 nm technology node. It is also shown that intra-group instruction permutation after opcode permutation does not reduce self- and coupling switching activities by a significant amount.**

I. Introduction

With advancement of VLSI technology into the deep sub-micrometer (DSM) level, large reduction on component spacing and complication of interconnect network increases self and coupling capacitance, which badly affects the power and energy efficiency. Many researches turn to encoding to reduce power consumption in interconnect bus. Cheng and Pedram provide a summary on the different types of encoding design, most of which introduce complicated encoding/decoding circuit [1]. Murgai et al. first design a theory model of a permutation encoding with no circuit overhead [5]. They generalize Data Ordering Problem (DOP), prove its equivalency to the TSP problem and provide three different heuristic algorithms to solve it. Many researches encode instruction bus data based on a DOP problem. Macchiarulo et al. presents a permutation mechanism on the physical interconnects buses [4]. Their work largely depends on circuit's work-loads, which requires reprogram-able circuit. Kuo et al. provide post compiler level approach to use code optimization methods to reduce the number of 4*C* and 3*C* crosstalk effect [2][3]. But their algorithm is effective within a basic block range and works only for delay latency reduction. Petrov and Orailoglu provide another original encoding on the instruction bus which focuses on single bit stream occurred on every individual bus line [6]. However, decoding circuit is comparatively complex in this approach.

Former researches usually ignore a fact that interconnect bus transmits instruction as well as data. Interleaving of instruction and data makes bus behavior less predictable. We focus on every instruction cache miss during which a block of instructions transmits sequentially. Such a block is stored in a cache line. Within this period, instruction order and type becomes deterministic based on executable code itself.

We reformat missed instructions block such that it can reduce the power consumption transmitted in interconnect bus. After instructions reach processor end, we recover the original executable code based on an index table constructed during former reorder operation. The idea in this paper is to separate bus behavior to behavior in the processor. Our contributions in this paper are as follows:

- Instruction permutation encoding breaks data and control dependencies. With little restriction to reorder algorithm, a big potential budget to reduce switching activity and crosstalk can be made.
- Instruction is usually divided into several parts, such as opcode, register and functions fields. Since operation and register used is usually limited in a program, we propose a heuristic to reduce the bit switching activity on each instruction field recursively.
- There is no encoding/decoding hardware overhead in our technique. Later in the paper, we will show that Index table only introduces a reason area, power and performance overhead.

The rest of paper is organized as follows. In Section II, we describe bus as *RC* interconnect model and provides our objective. In Section III, we include details on permutation algorithm and introduce index table. The experiment setup and result are presented in the Section IV. Finally, we conclude at Section V.

II. Problem Statement

In this paper, we use *RC* interconnect model to describe interconnect bus connecting processor chip and memory. C_g describes self capacitance while C_c is used to represent coupling capacitance. As technology goes to sub-micron level, C_c is contributing more to power dissipation than C_g is. In *RC* interconnect model, dynamic power, contributed by both type of capacitances, is given by:

$$P=\alpha(C_c+C_g)V_{dd}^2 f \quad (1)$$

where α describes average number of transmission, V_{dd} is the supply voltage and f is the clock frequency.

As C_g, V_{dd} are fixed for a given technology process, effective values of C_c is also affected by switching activity occurred on the adjacent bus lines. With defined ratio $\lambda=C_c/C_g$, power consumption expression is function of λ, such as

$$P=(1+N_c\lambda)\alpha C_g V_{dd}^2 f \quad (2)$$

where N_c is used to describe dynamic factor as bit values change.

The problem is defined as: setting power model as the objective function, our approach reorders program's binary code stored in memory in the size of cache line into the power efficient format. Since instruction size and cache line size are known beforehand, algorithm can be made offline. To restore instructions into its original format, the algorithm constructs an index table during the instructions permutation process.

III. POWER REDUCTION OF INSTRUCTION RESCHEDULING IN INTERCONNECT BUS

A. Overview of permutation algorithm

A program usually follows similar execution procedure: first load executable binary code from disk into main memory; then the processor begins execution by fetching instructions from cache. If cache miss happens, processor fetches needed instructions from memory. We define program's processor behavior as instruction traces that are executed in processor core [10]. Processor behaviors are different with different executable code and restricted by data and control dependency.

Similarly, we define interconnect bus behavior by observing bit values transmitted via the bus. The bus behavior is triggered by cache miss and its values are defined by instructions in program's address. However, bus behavior is different from processor behavior in following aspects: 1) bits values transmitted via bus is interleaved by the instruction bit, data bits and other management data, such as the page table information; 2) When cache miss occurs, processor follows the locality rule and requires transmission of a instruction block instead of a single missed one. Processor stores fetched block in cache as a cache line or cache block.

As long as instructions stored in cache line matches the code in memory, proper execution is guaranteed. Based on this observation, we separate interconnect bus behavior from the processor behavior. While processor behavior decides program's function, interconnect bus behavior is rearranged to reduce power consumption. Although bus behavior becomes unpredictable due to interleaving of data and instructions, it becomes deterministic during instruction cache miss. Memory searches required instruction block and transmits it without interruption. In this paper, we focus on bus behavior from instruction block. Since instructions are known from executable file before processor loads them into the memory, we can easily apply the algorithm during the compiler level, which further reduce performance overhead. Similarly, our algorithm can also apply on data block during runtime, which increases data cache miss penalty.

The algorithm starts when processor loads program's executable code into main memory. We divide program into different sections based on size of a cache line. Then we perform rescheduling operation on each section. Aforementioned power consumption model decides algorithm's optimization objective. In order to restore original instruction set for the proper execution, we construct an index table during permutation. The index table is an one dimensional (1D) table which is described in detail later in the paper. By introducing a small area overhead, index table eliminates complex encoding/decoding circuitry both in the memory end and processor. Fig. 1 shows the basic procedure to perform the instruction permutation.

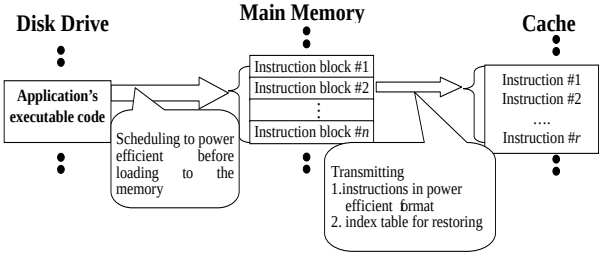


Figure 1 Framework of Power Efficient Instruction Rescheduling

B. Instruction permutation

Re-sequencing instructions block is equivalent to TSP problem, which is NP-complete. Usually, instructions' structure is divided into several fields to represent opcode, register, function, etc. Our heuristic adopts "divide & conquer" concept to reduce switching actively on each field of instruction recursively. For better explanation, we use RISC instruction set in which instructions share the same length. To apply the algorithm in CISC is also applicable by considering about alignment of different instructions.

1) Grouping same opcode

First, we group instructions with the same opcodes together. For example, first 6 bits of MIPS and Alpha instructions are opcodes. Although, a program can have thousands instructions, the number of opcodes values is restricted to 64 (2^6) in this situation.

On the other hand, modern assembler will also do some optimization which can help reduce the number of opcodes values. For instance, many assemblers will transform the operations such as, LI (load immediate) and SUBU (subtract unsigned), to the adding operation directly [11].

Two benefits are achieved by grouping instructions with same opcode. Firstly, adjacent instructions within the group have exactly zero self- and coupling-capacitance for opcode field. Crosstalk effect only happens between different groups. This already reduces large number of switching activities.

The second benefit is that further recursive permutation made within a group will not affect adjusted crosstalk effect on the opcode field, and only further reduce switching activity on other field, such as destination or function field.

We are using existing sorting algorithm to group instructions in $O(N\log_2 N)$ running time. The reason is that sorting opcode value puts instructions with same opcode together.

2) Intra group permutation

The size on every instruction group is thus reduced after the first stage of permutation and this provides larger flexibility to make the intra group permutation. At this stage, if group size is still large, we can simply recursively apply grouping on other field. For example, for the alpha instruction set, bit 21 to bit 25 indicates the destination register for the most (except PALcode) instructions [12]. Once group size

becomes small enough, an intra-group permuation is applied to explore the possibility of further reduction of both self- and coupling switching activities within the group. In this case, a greedy algorithm is applied. Starting from the first instruction in the group, the next instruction is selected based on the minimum number of switching in this exhaustive search method.

Table I shows the pseudo code for our complete heuristic, which includes both the grouping on opcodes and the intra group permutation.

TABLE I. INSTRUCTION PERMUTATION ALGORITHM

Procedure Grouping(*I, field*)
Sort *I* according to the values in the designated *field*.
Group the instruction with the same *field* values

input: $I = (I_1, I_2, \ldots, I_k)$ is the block of instructions with the size of *k*.
Define constant Groupsize
procedure PowerFormat(*I*)
Grouping(*I*, "opcode")
while 1 do
for each group *G* in *I*
if size of *G* is larger than Groupsize **then**
Grouping(*I*, next-filed)
else
intra-group permutation() //exhaustive search in the group
endfor
endwhile
Output: $I = (I_1, I_2, \ldots, I_k)$ block of instructions in power efficient format

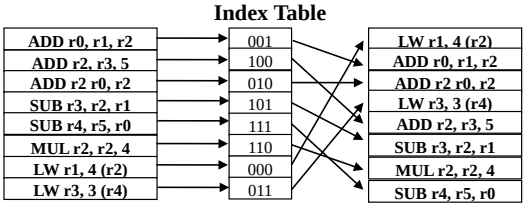


Figure 2 Index Table Example

C. Constructing index table for instruction restoration

1) Index table structure

During instruction permutation, a one dimensional index table is constructed. The index table contains relationship between instructions in its original format and ones in its power efficient format.

Fig. 2 shows an example of index table with the eight instructions block. The table contains the same number of the entries as the instruction block. Each entry records original location of an instruction in power efficient format. In order to describe relative location, number of bits in every entry has to be at least $\log_2 N$, with *N* as the size of instruction block. In this example, cache line contains 8 instruction, the table size is 8 entries with each entry has $\log_2 8=3$ bits.

2) Index table overhead analysis

After instructions reach processor, index table is consulted. As a result, memory has to send index table before sending the instructions. We propose to store index table in the processor end to reduce power consumption of transmitting index table itself on the bus. By introducing small area overhead in the processor, index table eliminates complicated encoding/decoding circuit on both processor and memory.

Storing index table also introduces extra power consumption in processor. As technology advances into smaller scale, such consumption becomes smaller while interconnect bus power becomes dominating. Later in the paper, we show that our approach trade-off only small power increase in the processor while reduce large part of the power in the interconnect bus.

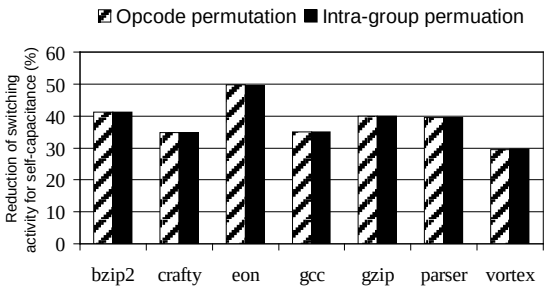
Finally, when cache miss occurs, instruction block accesses index table. This also results in extra clock cycles in instruction cache miss. Since we focus performance loss only in instruction cache miss penalty, this introduces ignorable performance loss for program's execution.

IV. SIMULATION RESULTS

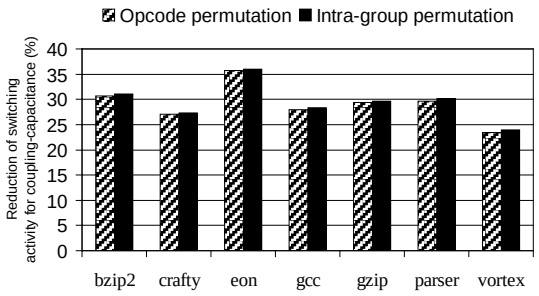
In this section, we provide experiment result of power reduction from our heuristic and then analyze overhead made by index table.

A. Interconnect bus power dissipation

We count switching activities on on-chip interconnect for both self- and coupling capacitances. The executable file is divided into different blocks and block size (thus the cache line size) is varied from 32 instructions to 256 instructions. Power consumption is calculated block by block and the average value is calculated to make comparison. SPEC CPU 2000 suite is used for simulation here [9]. The benchmark is directly compiled to little-endian alpha-executables. Seven integer program benchmarks (*bzip2*, *crafty*, *eon*, *gcc*, *gzip*, *parser* and *vortex*) are chosen. As seen in the Fig. 3 self- and coupling capacitance switching are reduced by about 40% and 30%, respectively. Besides, it is shown that intra-group permutation within blocks does not reduces power dissipation by a greater margin.



(a)



(b)

Figure 3. Percentage of self- and mutual-capacitance switching reduction with respect to original and scheduled instruction technique on SPEC CPU2000 benchmark for instruction buses

As mentioned before, the complexity for the heuristic that groups the instructions based on the opcode fields is $O(N\log_2 N)$ and the intra-group permutation imposes a complexity of $O(N^2)$. Since the each group size is very small in size compared to the entire executable file, the complexity for the intra-group permutation can be neglected. Thus, the entire heuristic basically has a complexity of $O(N\log_2 N)$.

B. Index table overhead

To recover instruction block to its original format, index table is stored in the processor end. We construct an index table as an extra level-2 cache. As mentioned before, each entry is $\log_2 N$ bits, so table size is $(a*\log_2 N)/8$ bytes, in which N is cache line size and a is number of blocks in the executable file. For the benchmark that we experiment, *vortex* requires largest index table with the size of 19.75k bytes. Consequently, we choose table size as 20k bytes. Even with larger executable files, this size can also reduce table transmission to only 3 or 4 times. We simulate area, power consumption and access time of index table under CACTI 5.0 [13] and provide result in Table II.

TABLE II. AREA, POWER AND ACCESS TIME OVERHEAD FOR INDEX TABLE

Technology node	Overhead criterion	L2 Cache (256k)	Index Table (20k)	Overhead
90nm	Area (mm^2)	7.6317	1.0682	14.0%
	Power (W)	0.62060	0.16960	27.3%
	Access time (ns)	1.6857	0.86934	51.6%
65nm	Area (mm^2)	3.9857	0.55717	14.0%
	Power (W)	0.47897	0.13750	28.7%
	Access time	1.1932	0.58601	49.1%
32nm	Area (mm^2)	0.96800	0.13506	14.0%
	Power (W)	0.43274	0.08099	18.7%
	Access time (ns)	0.60690	0.25402	41.9%

We choose the size of level-2 cache as 256k bytes, with 1-set associativity and 1 bank. This cache size is smaller than what is used in current processor design. Area overhead stays at the level of 15%. At present technology nodes, interconnect bus power dissipation dominates total chip power level as processor's power consumption is decreasing. Even though index table requires power, a net power reduction is achieved with increasing level of power dissipation on interconnect. Besides, it is also seen that access time for index table decreases with technology nodes. Even with largest delay of access time of 0.86934 nsec, in a processor with 2 GHz frequency, this is equivalent to not more than 2 clock cycles. This access delay is less than 10% of the original cache-miss penalty. As a result, accessing index table delay overhead has very little impact on the program's execution.

V. SUMMARY AND CONCLUSION

In this paper, we present an original instruction permutation heuristic to reduce self- and mutual-capacitance switching activity. We break all data and control constraints on instruction execution order and transfer them into a totally different power efficient format. A linear-dimension index table is constructed to restore instruction's original order. So permutation reduces power consumption in interconnect bus while maintaining program's proper execution. Index table is built as an extra cache in processor side and every time a block of reordered instructions is transmitted, it consults the corresponding index table and restores instruction order.

With different cache line size, our heuristic shows great reduction on switching activities, resulting in promising reduction in power consumption. Our simulation on SPEC CINT 2000 benchmark shows that self- and coupling capacitance switching are reduced by 40% and 30%, respectively. On the other hand, the index table only introduces a reasonable area overhead by comparing its size to the cache size. Our estimation is actually pessimistic and in the general purpose computer the large level-2 cache will make the area overhead ignored.

REFERENCES

[1] W. Cheng, and M. Pedram, "Memory bus encoding for low power: a tutorial," in *International Symposium on Quality Electronic Design, 2001*, p. 199–204.

[2] W. Kuo, Y. Chiang, T. Hwang and A. Wu, "Performance-driven crosstalk elimination at post-compiler level," in *Proc. of ISCAS, 2006.*

[3] W. Kuo, Y. Chiang, T. Hwang and A. Wu, "Performance-Driven Crosstalk Elimination at Postcompiler Level—The Case of Low-Crosstalk Op-Code Assignment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, pp. 564–573, Mar. 2007.

[4] L. Macchiarulo, E. Macii, and M. Poncino, "Low-energy encoding for deep-submicron address buses," in *International Symposium on Low Power Electronics and Design, 2001*, p. 176–181.

[5] R. Murgai, M. Fujita, and A. Oliveria, "Using complementation and resequencing to minimize transitions," in *Proc. of DAC, 1998*, p. 694–697.

[6] P. Petrov and A. Orailoglu, "Low-power instruction bus encoding for embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 812–826, Aug. 2004.

[7] M. Stan and W. Burleson, "Bus-invert coding for low-power I/O," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, pp. 49–58, Mar. 1995.

[8] "International Technology Roadmap for Semiconductor (ITRS)", Semiconductor Industry Association, 2001 Edition, 2001.

[9] SPEC CPU2000 benchmark. [online]. Available at: http://www.spec.org/

[10] W. Stallings, *Operating Systems: Internals and Design Principles*, 6th ed., Prentice Hall, 2008

[11] (2009) MARS (MIPS Assembler and Runtime Simulator): An IDE for MIPS Assembly Language Programming, http://courses.missouristate.edu/KenVollmar/MARS/

[12] "Alpha Architecture Handbook," Compaq Computer Corporation, Oct. 1998

[13] Available at: *http://www.hpl.hp.com/research/cacti/*