Thread Scheduling Based on Low-Quality Instruction Prediction for Simultaneous Multithreaded Processors

Houman Homayoun[1], Kin F. Li[1], and Setareh Rafatirad[2]
[1]Department of Electrical and Computer Engineering, University of Victoria, {homayoun,kinli}@uvic.ca
[2]Department of Computer Science, Azad University, rafatirad@azad.ac.ir

***Abstract*- A Simultaneous Multithreaded (SMT) Processor is capable of executing instructions from multiple threads in the same cycle. SMT in fact was introduced as a complementary architecture to superscalar to increase the throughput of the processor. Recently, several computer manufacturers have introduced their first generation SMT architecture.**

SMT permits multiple threads to compete simultaneously for shared resources. An example is the race for the fetch unit which is a critical logic responsible for thread scheduling decisions. When more threads than hardware execution contexts are available, the decision of choosing the best threads to fetch instructions from, will affect the processor's efficiency.

In this paper we present a new approach to choose the most useful threads among all available threads while they compete on a shared resource. We identify the quality of instructions based on the time they spend in the instruction queue. Low-quality instructions spend more time in the instruction queue. Accordingly threads with fewer number of low-quality instructions have a higher contribution to the entire processor throughput. In an experimental study, we identify such low-quality instructions in each thread to a maximum of 92% accuracy (average 72%). We exploit this to increase the overall processor throughput by giving higher priority to threads with lesser number of low-quality instructions. Overall we achieve an average of 11% performance improvement over the traditional algorithm that schedules threads in a round-robin fashion.

I. INTRODUCTION

A Simultaneous Multithreaded (SMT) Processor is capable of executing instructions from multiple threads in the same cycle [2, 6]. SMT introduces a complementary architecture to superscalar to increase the throughput of the processor by running different programs (which are referred to as threads) at the same time. In fact SMT exploits the independency of instructions belonging to different threads to increase the processor throughput.

Past studies show that a considerable amount of hardware resources in conventional superscalar processors are unused during execution of a single program [2, 5, 8, and 9]. SMT attempts to use all superscalar resources by assigning them to multiple threads. Accordingly threads compete every cycle to acquire shared hardware resources such as instruction and data caches, TLBs, register renaming unit, fetch bandwidth, instruction queue and functional units [4]. The desirable policy in resource competition is the one which biases toward the threads with more contribution to the overall throughput. According to [1], a major factor that makes a thread more desirable over others is its instruction behavior in the instruction queue. To achieve a higher throughput, threads with instructions that spend the least time in the instruction queue should be given the highest priority in scheduling. The problem to find such a scheduling policy is that we basically have no information about the time instructions spend in the instruction queue before they are issued. Instead, several policies proposed in the literature use feedbacks from other parts of the processor as a replacement of such oracles scheduling [3, 4, 7 and 11]. An algorithm that gives priority to threads with the fewest instructions in the pipeline, and a fetch policy which gives priority to threads with the fewest D-cache misses are two examples of such proposals.

In this work we take a step towards finding the oracles scheduler. We first study instruction behavior in SMT processors. Then we propose an approach to predict the time instruction spend in instruction queue before they are issued (we refer to this factor as issue delay). Issue delay prediction outcome can then be used for scheduling decision in any thread or instruction race condition to get processor resources. Finally, to improve processor throughput, we use our prediction outcome for scheduling decision in one of the shared resources in SMT architecture: the fetch unit.

II. LOW-QUALITY INSTRUCTION PREDICTION

Instruction queue in SMT architecture, like in superscalar machines, is the core for out-of-order execution. Instructions wait in instruction queue until their source operands become available. There are many factors influencing instruction issue delay (also referred to as IID), including instruction dependency and resource availability.

Fig. 1 shows the IID distribution for a multiprogram workload of a subset of the SPEC CPU2000 benchmark suit which includes a wide variety of typical applications with high and low IPC and those limited by memory, branch misprediction, etc [10]. For workloads with two benchmarks we simulated 200 million instructions after skipping the initial 200 million instructions. Workloads with four benchmarks were fast-forwarded for 400 million instructions and then were simulated for 400 million committed instructions. Instructions are categorized based on their issue delay to high quality (HQ)

0-7803-8935-2/05/$20.00 ©2005 IEEE.

or low quality (LQ). If an instruction's issue delay exceeds a pre-determined threshold, we refer it as a low-quality instruction otherwise it as a high-quality instruction. Through this study we define LQ-instructions as those spending at least 5 cycles in the instruction queue. This threshold is dependent on the processor configuration and it has been chosen after experimenting with different numbers of cycles. On average,

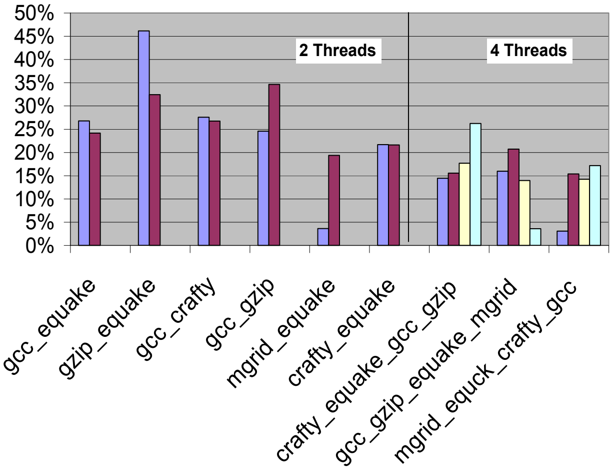

Fig. 1. Distribution of LQ-instructions in SMT processor.

when 2 threads are available, around 26% (maximum of 46%) of the instructions are LQ, i.e., they spend at least 5 cycles in the instruction queue. This drops to 15% in the case when 4 threads are available. This in fact is expected since with 4 threads the opportunity to fetch HQ instructions is higher.

Fig. 2 shows the configuration we propose to predict LQ-instructions. Fetch unit reads instructions from the instruction cache. Next, instructions are decoded and their logical register

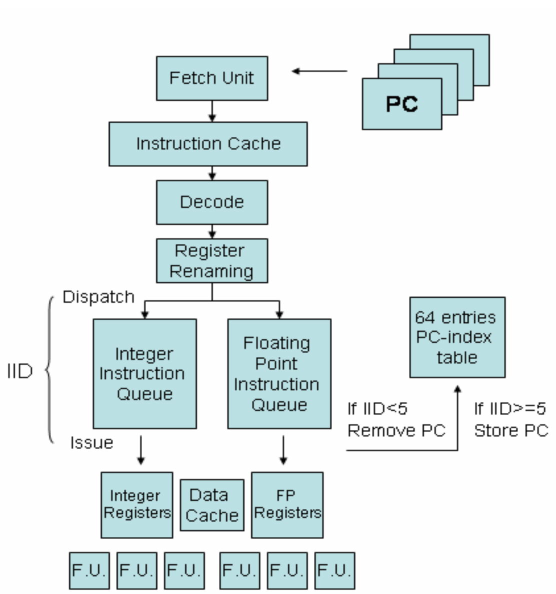

Fig. 2. SMT pipeline with logic to predict LQ-instructions.

operands are renamed to physical register. Renamed instructions are dispatched to instruction queue. This is the beginning of out of order execution. Before this stage all instructions are processed in sequential order as in the program. Once instructions pass this stage they can be processed out of order. An instruction queue is a pool of instructions waiting for their source operands to become available. When operands become available, the instruction is sent to an appropriate free functional unit. To predict LQ-instructions we use a small 64-entry program counter (PC)-indexed table. We refer to this table as the IID-table. While exploiting larger and more complex structures may improve prediction accuracy, we avoid such structures to keep power and latency overhead at a low level.

To predict whether an instruction is LQ, we probe the IID-table at dispatch stage. If the instruction PC is found in the table, which indicates that the instruction was low quality last time, we predict it to be low quality this time. An instruction with both source operands ready before being dispatched to an instruction queue is deemed HQ, and such prediction is unnecessary. At instruction issue time, if an instruction's IID is at least 5, we store its PC in the IID-table if it is not already in it otherwise nothing is done. On the other hand, the PC of an instruction having an IID less than 5 is removed from the IID-table if it has an entry in it otherwise nothing is done.

We evaluate the proposed prediction scheme using two metrics: prediction accuracy and prediction effectiveness.

LQ-instruction prediction accuracy reports how often instructions predicted to have an issue delay of at least 5, do indeed stay in the instruction queue for at least 5 cycles. This, while important, does not provide enough information, as it indicates nothing regarding the percentage of LQ-instructions identified. Therefore, we also report prediction effectiveness, i.e., the percentage of LQ-instructions identified.

A. Prediction Accuracy

In Fig. 3 we report prediction accuracy for LQ-instructions. On average, prediction accuracy is 72%. In workloads with

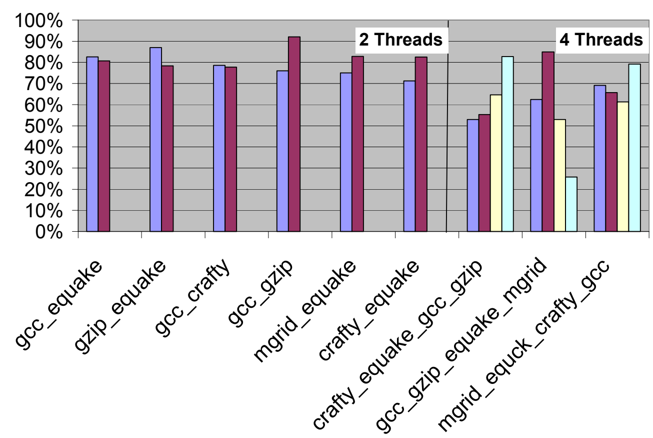

Fig. 3. LQ-instruction prediction accuracy.

two benchmarks, *gzip* has the highest accuracy (about 92%) while *crafty* has the lowest rate (about 71%). *Gzip* and *mgrid* have respectively the highest and lowest prediction accuracy (82% and 25%) for workloads with four threads.

B. Prediction Effectiveness

In Fig. 4 we report prediction effectiveness. On average, effectiveness is about 40%. In workloads with two benchmarks, the maximum effectiveness is achieved for *gzip* where we accurately identify more than 70% of LQ-instructions. Minimum effectiveness is obtained for *crafty*, where only about 18% of LQ-instructions are identified. The same benchmarks also have the highest and lowest prediction effectiveness for workloads with four threads (79% and 15%). The identification of the characteristics of applications with low prediction effectiveness will make an interesting study in the future.

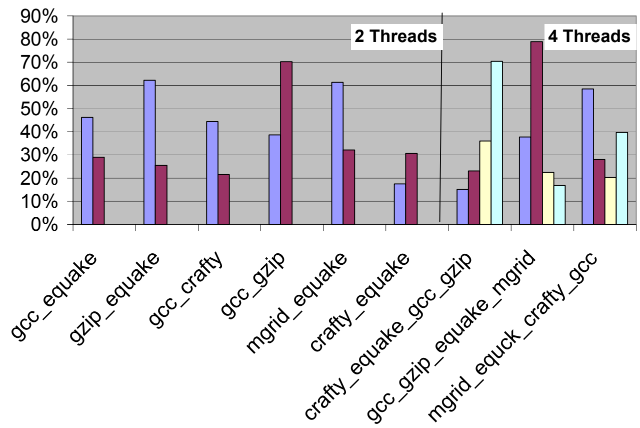

Fig. 4. LQ instruction prediction effectiveness.

Overall, our scheme correctly predicts 42% of all LQ-instructions. The average prediction accuracy is 72%. Accordingly, we can use our predictor outcome in scheduling decision when multiple threads compete to obtain common resources. Predicted Non-LQ(HQ) instructions and thus threads with lesser number of predicted LQ-instructions are expected to have more contribution to processor throughput since their instructions can be moved out of the instruction queue faster than other threads. Therefore, at thread level competition higher priority is assigned to threads with fewer number of LQ-instructions in the instruction queue.

III. THREAD SCHEDULING DECISION IN FETCH UNIT

In this section we present a scheduling algorithm based on LQ-instruction prediction. Then we compare our scheme to an instruction-quality unconscious algorithm which schedules regardless of thread behavior.

As explained earlier LQ-instructions stay in the instruction queue for long cycles and as a result they impede subsequent instructions that have data dependency to be issued. To achieve a higher throughput, scheduling policy should be able to identify threads which will use processor resources more efficiently. We give a higher priority to threads which have fewer number of predicted LQ-instructions in the instruction queue. Threads with more predicted LQ-instructions do not require as much hardware resources as the other threads since these instructions stay in the instruction queue for a relatively longer period. These threads are given lower priority in our scheduling policy to maximize resource utilization. We refer to this scheme as Low-LQ prediction based scheme.

Finally, to validate our proposed scheme we compare it to a scheduling algorithm which gives a higher priority to threads with more predicted LQ-instructions (High-LQ prediction based). It can be argued that such algorithm would clear up the overall system faster by fetching more LQ-instructions, thus making them more readily available which may result in higher overall IPC.

IV. RESULTS

In this section we report our analysis framework. For the microarchitectural simulation, we used a modified version of SMTSIM v2.0 alpha [2]. The base processor model which fetches from at most two threads in the same cycle is detailed in Table I.

Table I
Base processor configuration

Pipeline	9 stages
Fetch Policy	8 instructions/cycle, up to 2 threads
Functional Units	6 integer, 3 floating point
Instruction Queues	64-entry integer and floating point queues
Renaming Registers	100 integer and floating point
Commit Width	12 instructions/cycle
Branch Predictor	Hybrid Predictor
BTB	4K entries, 4-way set associative
I-Cache	256KB, 2-way set associative, single ported
D-Cache	256KB, 2-way set associative, dual ported
L2 cache	16 MB, direct mapped, 20-cycle latency, fully pipelined
Memory bus	128 bits wide, 4-cycle latency

In Fig. 5 we report how scheduling policy based on LQ prediction can improve performance over the traditional round-robin policy. Across all workloads, Low-LQ scheduler has higher IPC throughput over the round-robin scheme. On average, scheduling based on LQ prediction improves performance by 11%. Four-thread programs benefit more from such scheduling than 2 threads. This shows the importance of using a smart scheduler as the number of threads increases.

Fig. 6 compares the performance of the Low-LQ scheme with the High-LQ prediction based scheme. Across all work-

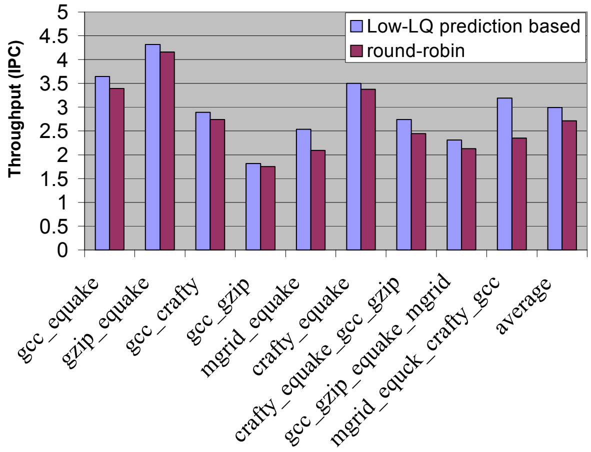

Fig. 5. Performance improvement achieved by using LQ prediction.

loads the Low-LQ scheme outperforms High-LQ scheme with an average of 26% higher IPC. Low-LQ scheduling achieves a higher IPC by smartly giving each thread the hardware resources it requires, and clearing up the whole system faster. In fact, the round-robin scheduling scheme performs better than the High-LQ scheme in every benchmark.

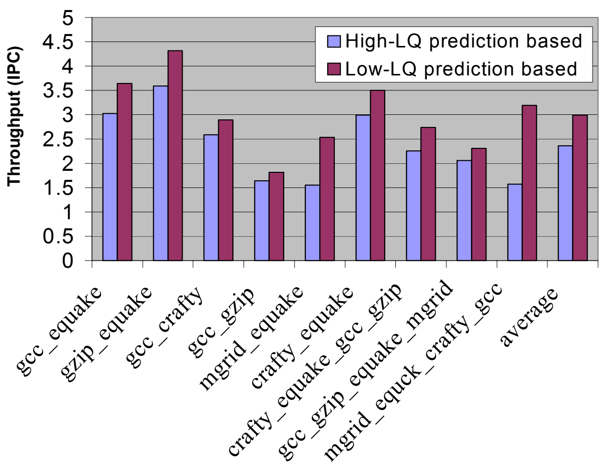

Fig. 6. Performance degradation of High-LQ scheduling compare to Low-LQ scheduling.

V. Conclusion and Future Work

In this work, we defined LQ-instructions and introduced an algorithm to predict these instructions. We then proposed an LQ prediction based policy for thread scheduling decision in the fetch unit. In thread level competition we assigned higher priority to threads with fewer number of predicted LQ-instructions. Our results showed a significant improvement over the traditional round-robin scheduling scheme. The same policy can be used in instruction level competition in which higher priority is given to predicted HQ instructions. Future work will examine LQ and HQ prediction based policies in other common SMT processor resources such as register renaming unit, instruction queue, TLBs and functional units. The usefulness of instruction execution time as a decision factor for scheduling will also be investigated. Furthermore, the tradeoff between prediction overhead and the performance gained will be studied.

References

[1] Dean Tullsen, Susan Eggers, Joel Emer, Henry Levy, Jack Lo, and Rebecca Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[2] Dean Tullsen, Susan Eggers, and Henry Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *Proceedings of the 22rd Annual International Symposium on Computer Architecture*, June 1995.

[3] Josh Redstone, Susan Eggers, and Henry Levy, "Mini-threads: Increasing TLP on Small-Scale SMT Processors", *Proceedings of the International Conference on High-Performance Computer Architecture*, February 2003.

[4] Sujay Parekh, Susan Eggers, Jack Lo, and Henry Levy, "Thread-Sensitive Scheduling for SMT Processors", *University of Washington Technical Report*, 2000.

[5] Daniele Folegnani and Antonio Gonzalez, "Energy-Effective Issue Logic," *Proceedings of the 28th Annual International Symposium on Computer Architecture* (*ISCA-2001*), Goteborg, Sweden, July 2001.

[6] Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen, "Simultaneous Multithreading: A Platform for Next-generation Processors", IEEE Micro, September/October 1997.

[7] Eric Tune, Rakesh Kumar, Dean M. Tullsen, Brad Calder, "Balanced Multithreading: Increasing Throughput Via a Low Cost Multithreading Hierarchy", *Proceedings of the 37th International Symposium on Microarchitecture*, December, 2004.

[8] Arun K. Somani and Joel Nickel, "REESE: A Method of Soft Error Detection in Microprocessors", *Proceedings of the International Conference on Dependable Systems and Networks* (*DSN-2001*), Goteborg, Sweden, July 2001.

[9] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan, "Temperature-Aware Microarchitecture", *Proceedings of the 30th Annual International Symposium on Computer Architecture* (*ISCA-2003*), June 2003.

[10] Standard Performance Evaluation Corporation: *SPEC CPU 2000 V1.2* at www.spec.org/cpu2000

[11] Chulho Shin, Seong-Won Lee, Jean-Luc Gaudiot, "Dynamic Scheduling Issues in SMT Architectures", *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003).*