Computers and Electrical Engineering 38 (2012) 1616–1626

Contents lists available at SciVerse ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Adaptive instruction dispatching techniques for Simultaneous Multi-Threading (SMT) processors ☆

Monobrata Debnath, Wei-Ming Lin *, Eugene John

Department of Electrical and Computer Engineering, The University of Texas at San Antonio, San Antonio, TX 78249-0669, USA

ARTICLE INFO

Article history:
Received 21 December 2011
Received in revised form 20 June 2012
Accepted 20 June 2012
Available online 3 August 2012

ABSTRACT

Simultaneous Multi-Threading (SMT) provides a technique to improve resource utilization ability by sharing key data-path components among multiple independent threads. When critical resources are shared by multiple threads, effective use of these resources proves to be the most important factor in fully exploiting the system potential. Transient behaviors of various threads in terms of their execution parallelism can easily affect utilization efficiency of these shared resources. To commit more resources to threads that are more active allows for better resource utilization and thus higher throughput. In this paper, we propose a real-time dynamic scheduler for the SMT which dispatches instructions from threads based on thread-activeness information gathered in real time and dynamically adjusts dispatching priorities among threads accordingly. An extensive simulation shows a significant gain in system throughput by this technique. The performance of the proposed dispatching technique is evaluated on different workload mixtures created based on instruction-level parallelism available in each thread. An average of 6.5% and maximum of 15% performance improvement is observed with the proposed dispatching technique.

1. Introduction

Based on the traditional superscalar processors, Simultaneous Multi-Threading (SMT) offers an improved mechanism to enhance overall system performance without having to invest a proportional amount of extra hardware. In a conventional superscalar processor, not only the functional units are not close to be fully utilized with only one thread running at any time, switching from one thread (task) to another involves the intervention of the operating system which leads to extra overhead in CPU power. Simple multi-threaded processors remove the necessity of task switching by OS but still do not fully exploit the capacity of the functional units since a single thread does not usually have enough instruction-level parallelism. A coarse-grained multi-threaded processor does not interleave instructions from different threads in processing, while a fine-grained one allows for a cycle-to-cycle interleaving from different threads' instructions, but neither permits instructions from different threads to be issued in the same clock cycle. SMT takes this one step further in order to exploit the full potential of the functional units. The most common characteristic of SMT processors is the sharing of key datapath components among multiple independent threads, which ensures improved resource utilization. SMT also exploits not only thread-level parallelism (TLP) among the various threads [1,2], but also equally concentrates on the advantages available at the instruction-level parallelism (ILP) in each thread. Subsequently, due to the sharing of resources, the amount of hardware required in an SMT system is significantly less than a multi-core superscalar machine without sacrificing much performance. Most common resources shared in SMT technique include components that are control-complexity-wise easier to share (such as cache

☆ Reviews processed and approved for publication by Editor-in-Chief Dr. Manu Malek.

* Corresponding author. Tel.: +1 210 4585529; fax: +1 210 4585947.
E-mail address: weiming.lin@utsa.edu (Wei-Ming Lin).

0045-7906/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compeleceng.2012.06.010

memory and physical register bank), and those that are cost-wise better to share (such as Issue Queue (IQ)/Reservation Stations and various functional units). On the other hand, other more thread-specific components (such as Instruction Fetch Queue (IFQ) Re-Order Buffer (ROB)) along the datapath are assumed to remain per-thread ownership. These shared hardware components tend to remain busy in order to accommodate more instructions. Although these instructions ensure the full performance potential afforded by SMT [3], they tend to induce extra control complexities in managing the critical timing path and processors cycle. To retain the exploitation of both TLP and ILP, a necessary solution must be introduced in order to minimize the complexity among these shared resources without affecting the ILP exploitation significantly. At the same time, proper intelligence has to be incorporated into the resource sharing mechanism to ensure that threads share these components in the most efficient and fair manner. A shared IQ provides better performance than a private issue queue entry [4] and per-thread ROB significantly reduces design complexities, which combination therefore is the model for our study.

2. Related work

Recently there have been many research activities targeted in improving SMT performance through modifying various stages in the pipeline. These include: improvement of thread co-scheduling by using probabilistic modeling for prediction in [5], avoiding register allocation by predicting transient values from branch miss-prediction in [6], packing instructions in issue queue to reduce delay and power consumption in [7], improving SMT fetching with an estimation of outstanding work in the system for each thread in [8], early de-allocation of registers in association with cache misses in [9], dynamically reconfigurable cache design for better IPC in [10], a modified fetch policy with adaptive resource partitioning in [11], and an fetch policy by considering memory-level parallelism in [12]. None of these techniques specifically addresses the contention in the issuing stage and most come with a significant requirement in extra hardware to implement the desired intelligence.

There have been many non-adaptive dispatching algorithms proposed, including simple round-robin, within clock cycle round robin [13], two-op-block [14], etc. None of these algorithms are assigned with a priority among threads, instead all are based on a pre-assigned thread index order. Traditional simple round-robin dispatching simply starts dispatching all dispatchable instructions from a thread according the index order and moves to the next thread if the dispatching bandwidth allows for more instructions. The index order is then rotated naturally to the next thread to start in order in the next clock cycle. There are another set of algorithms that consider how to effectively utilize the limited IQ entries, including capping the usage of IQ entries per thread [15] and recalling stalled instructions from IQ [16].

3. Instruction dispatch

There have been many different terminologies adopted for instruction pipelining stages (e.g. issue, dispatch, etc.) in a superscalar system, and their references became even more ambiguous in an SMT system in which additional resource sharing is required than in a superscalar system. For example, instruction "issuing" has been referred to different stages of processing by different articles. Throughout this paper, we choose to adopt the terminologies used by most SMT articles.

In a typical single-thread superscalar system, instructions are "dispatched" from ROB into the reservation stations (either centralized or functional unit-specific) whenever space is available and then "issued" to the corresponding functional unit whenever the issuing conditions are met, i.e. operands are ready and the requested functional unit becomes available. However, in a basic SMT system, each thread has its own ROB and instructions from these thread-specific ROBs have to "compete" for a shared Issue Queue (IQ) through a dispatching scheduling algorithm. This IQ can be considered as Centralized Reservation Station not only shared among the functional units but also shared among the threads in real time. A basic functional block diagram of this basic design is depicted in Fig. 1.

Due to the significantly large size of each IQ entry, the number of entries in this shared resource usually is much smaller than the number of ROB entries. Having its output sent into a tightly shared resource, the instruction dispatch stage is considered one of the most critical stages that dearly affect the overall system performance. As shown in Fig. 2, Instructions-Per-Cycle (IPC) improves with a larger IQ size, clearly indicating the need for an effective IQ management policy in order to achieve better performance with a limited IQ size.

There have been many different techniques in scheduling instructions to be dispatched from ROB intending to exploit ILP. Most techniques rely on simple instruction-level readiness to reduce the instructions' waiting time in IQ [14]. To better utilize the shared IQ entries, instructions from different threads should be given different priorities depending on the then threads' "activeness" in real time. Threads have been shown to demonstrate various types of transient behaviors throughout their life span, including stretches of time durations with fluctuating ILP. Instructions from thread(s) of lower ILP (or simply "slower" in instruction issuing) would clog the IQ if they are allowed to continue to be dispatched into IQ. On the other hand, instructions from higher ILP (or simply "faster" in instruction issuing) should be given a higher priority in utilizing the IQ. There have not been many research results on how to share the IQ in a more time-adaptive manner allowing different threads to utilize (occupy) the IQ in an "on-demand" basis. In [17], an adaptive technique is proposed to limit the use of ROBs in order to accommodate threads that are more active with higher chances of using the IQ than the threads that are less active, in which "activeness" of a thread is based on a real-time evaluation of the number of issue-bound and commit-bound instructions in a thread's ROB. Control complexities involved in adjusting the use the ROBs may be the most inhibitive factor in justifying the performance gain from such an approach. Other more advanced scheduling techniques, such as the one