Efficient Resource Allocation on a Dynamic Simultaneous Multithreaded Architecture

Ortiz-Arroyo, Daniel

Efficient Resource Allocation on a Dynamic Simultaneous Multithreaded Architecture

Daniel Ortiz-Arroyo[1]

Computer Science and Engineering Department,
Aalborg University Esbjerg,
Niels Bohrs Vej 8, 6700 Esbjerg, Denmark

ABSTRACT

In this paper we describe the *Dynamic Simultaneous Multithreading (DSMT)* architecture, together with a simple resource allocation policy that may be employed to improve its performance. DSMT exploits TLP and ILP available in single applications on a SMT architecture core. In DSMT, threads are speculatively created and executed, while special hardware mechanisms keep track of misspeculations and dependence violations through registers and memory.

KEYWORDS: Thread Level Parallelism; Simultaneous Multithreading; Speculation

1 Introduction

In multithreaded architectures, TLP can be exploited using *implicit* or *explicit* mechanisms. *Implicit* multithreaded architectures aim is to improve the execution time of a single program, whereas *explicit* multithreading goal is to improve the execution time of a multiprogrammed workload. Implicit multithreading comprises either *static* methods such as parallelizing compilers or binary annotators to identify threads or *dynamic* techniques that extract threads speculatively at run time from a single program. *Explicitly* multithreaded architectures such as *Simultaneous Multithreading (SMT)* exploit efficiently the resources in the processor by executing multiple independent threads simultaneously, at the cost of a small increment in complexity. However, despite its advantages, being SMT an explicit architecture, it does nothing to improve the performance of a single program. Moreover, SMT may cause cache pollution because different threads are competing for the shared cache and, as is described in [Caz04], the fetching policy employed greatly impacts the performance of the processor.

This paper briefly describes an implicitly multithreaded architecture called *Dynamic Simultaneous Multithreading (DSMT)* based on a SMT microarchitecture. We also describe a resource allocation policy that can be used to improve DSMT's performance. This paper is organized as follows. Section 2 briefly describes previous related work. Section 3 provides a detailed description of the DSMT microarchitecture. The resource allocation policy proposed is briefly presented in Section 4. Finally Section 5 presents some conclusions.

[1] Email: do@cs.aaue.dk

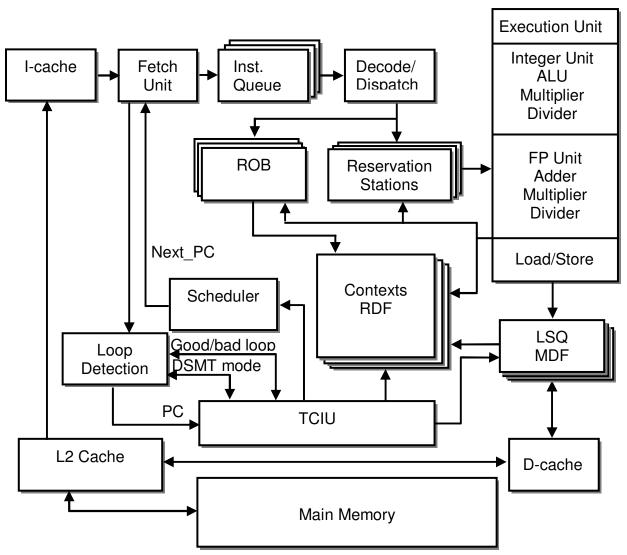

Figure 1: DSMT Microarchitecture.

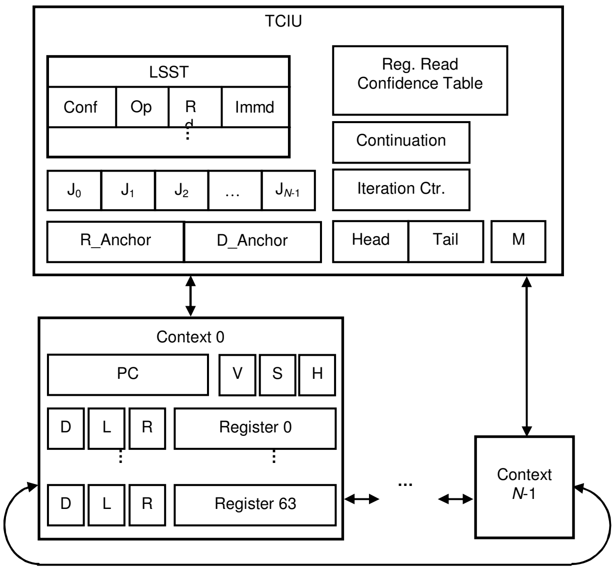

Figure 2: TCIU and Multiple Contexts.

2 Related Work

In Dynamic Multithreaded (DMT) processor [Akk98], threads are generated from *loops* and procedure *continuations*, i.e., instructions from either after the call or backward branch. Thus, it precludes the exploitation of the potentially high-degree of parallelism that exists across inner loop iterations. Moreover, threads are allowed to be spawned out of program order, which results in lower branch and return address prediction accuracies. These factors lead to long threads that increase the chance of data dependence misspeculation. In contrast, the *Clustered Speculative Multithreaded (CSM)* processor [Mar99] generates threads from target of backward branches, in particular innermost iterations of a loop. In CSM, inter-thread data dependencies through registers are resolved by performing data value prediction. When a thread finishes, its output predictions are verified and mispredictions are handled by selective re-execution. Inter-thread memory dependence speculation is performed by means of a *multi-value* cache. This special cache memory stores, for each address, as many different data words as the number of thread units. Among the SMT-based implicitly multithreaded processors proposed, DSMT employs less complex mechanisms comparatively with other similar architectures [Akk98, Mar99] specifically: a) a simple mechanism based on utility bits to detect and resolve inter-thread dependencies through registers and memory, b) no trace buffers or specialized cache structures, c) state bits and simple flushing mechanisms perform control of speculation, and d) a simple loop stride value prediction mechanism.

Diverse resource allocation policies for SMT processors have been proposed. The goal of these mechanisms is to improve SMT's performance, avoiding resource contention by giving each thread fair access to resources. The dynamic resource allocation policies presented in [Caz04] (DCRA) classify threads according to their demands on resources as *fast-slow, active-inactive*. Special counters keep track of resource utilization.

3 DSMT Microarchitecture

DSMT operates either in *non-DSMT*, *pre-DSMT* or *DSMT mode* [Ort03]. In the non-DSMT mode, there is only a single thread of execution and thus the processor behaves as a superscalar processor. When a loop is detected, the processor transitions to the pre-DSMT mode to detect not only live registers, intra- and inter-thread dependent registers, but also those registers that are most likely used as induction variables in the loop. Later, if it is determined that multithreaded execution will improve performance, the microarchitecture transitions to the DSMT mode of execution where the overlapped execution of multiple threads occurs. Figure 1 shows the organization of the DSMT microarchitecture. It consists of a SMT architecture organized into nine pipelined stages: Fetch, Decode/Dispatch, Inter-Thread Register Resolution, Issue, Execute, Memory, Write-back, Commit and Thread Spawning. The Fetch stage fetches a block of instructions from a thread, but can also fetch instructions from different threads based on the scheduling policy offered by the *Scheduler*. The fetched instructions are then decoded, renamed, and dispatched to the *Reservations Stations (RSs)* associated with each functional unit. At the *Inter-Thread Register Resolution* stage, inter-thread dependencies through registers are resolved. Finally, during the *Thread Spawning* stage speculative contexts are spawned and the control of thread-level speculation is performed. To support simultaneous execution of multiple threads, each thread has its own set of *Instruction Queue (IQ), Reorder Buffer (ROB),* and *Context* registers. Each *Context* represents the state of a thread and the multiple *Contexts* are interfaced to the *Thread Creation and Initiation Unit (TCIU),* which controls how threads are spawned and executed. The *Register* and *Memory Data Flow (MDF and RDF)* mechanisms associated with the Contexts and *Load Store Queues (LSQ)* keep track of inter-thread dependencies through registers and memory respectively. The *Loop Detection Unit* is responsible for detecting loops and supplying target addresses so that the *Thread Creation and Instatiation Unit (TCIU)* which ultimately spawns multiple threads. Whenever a taken backward-branch instruction is detected, its branch and target addresses are recorded in the *Loop Detection Unit*. Later, a second branch instruction with the same branch target address causes the processor to enter its pre-DSMT mode of execution. A loop is classified as *good* or *bad* based on a "break even" policy, where the IPC measured for a loop during pre-DSMT mode is compared against the observed *Sustained IPC (SIPC)* during previous run(s) of the same loop in DSMT-mode. DSMT aims to guarantee that the DSMT mode of execution will result in better performance comparatively with the non-DSMT mode of execution regardless of misspeculations. The control of nested loop execution is handled by a special stack structure associated with the *Loop Detection Unit*.

Figure 2 shows the multiple contexts in the processor physically interconnected in a ring fashion. In DMST mode the thread spawning process starts by copying the target address of a loop in the Continuation register to the PCs of each spawned context. To *"jump start"* each thread, values detected during pre-DSMT mode of immediate addressing mode instructions of the form addi rd,rd,#immd are predicted and stored in the *Loop Stride Speculation Table* (LSST). The speculation performed by DSMT is to predict the content of a source register rd in LSST as rd=rd+iteration*immd, where iteration is the current iteration number speculatively executed. To avoid blind speculation on rd, each entry in LSST is associated with confidence (Conf) bits based on 2-bit saturation counters. To keep track of inter-thread dependent registers, each physical register is associated with a set of *utility bits*. They are: *Ready* (R) bit that indicates some instruction(s) logically preceding the current one in the thread's program order has committed a value to the register; *Dependency* (D) bits that keep track of registers that have inter-thread dependencies; *Load* (L) bit that indicates that the register has been speculatively read from a predecessor context. Since register reads are speculative, a confidence based on 2-bit saturation counter is associated with each register and stored in the *Register Read Confidence Table*. Finally, to synchronize the execution of multiple threads, each context has associated a *Join* (J) bit, which is set to indicate that a thread has completed the execution of a single iteration in a loop. When this occurs, speculative contexts that could have finished earlier due to *dynamic behavior* must wait until the non-speculative context commits its results and transfers the non-speculative status to a new successor context. Transferring the non-speculative status comprises, copying the register values generated locally by the non-speculative context to the successor context, and updating the TCIU with the tag identification of the new non-speculative context (the head). In DSMT, loads from different threads, can be executed speculatively. However, only the non-speculative threads are allowed to perform stores. The *Memory Dataflow Mechanism* ensures that the sequential semantics is not violated.

4 Resource Allocation Policy for DSMT

Our experiments presented in [Ort03] show that DSMT suffers from resource contention among threads. DSMT fetching policy modifies the original SMT's policy by assigning a fetching port to the *non speculative* thread and sharing the other available port among the *speculative* threads using ICOUNT. However, as pointed out by [Caz04], when loops have dynamic behavior i.e. diverse resource demands, resource allocation policies can improve performance by assigning, dynamically, critical resources among the speculative and non-speculative threads. Our experiments show that DSMT's critical resources are the integer and floating point units, additionally to the load/store ports. The dynamic resource allocation policy proposed for DSMT is described by the following equation:

$$E = \begin{cases} \dfrac{R(1+SD)}{T-1} \; for\ non-speculative\ thread \\ R - \dfrac{R(1+SD)}{T-1} \; for\, speculative\, threads \end{cases}$$

where E is the number of resource entries allocated to each thread, R total number of entries of a resource, T the total number of contexts in the processor, S is the sharing factor (to be determined experimentally), D is the number of threads with dynamic behavior.

5 Conclusions

We have presented the DSMT architecture and a resource allocation policy that dynamically allocates critical resources. The performance obtained by the use of this policy in loops with dynamic behavior will be evaluated experimentally.

References

[Akk98] Akkary H. and Driscoll M., "A Dynamic Multithreading Processor," *31st Int'l Symp. On Microarchitecture*, Dec 1998.

[Mar99] Marcuello P. and Gonzales A., "Clustered Speculative Multithreaded Processors," *Proc. of the Int'l. Conference on Supercomputing (ICS'99)*, 1999.

[Caz04] Cazorla, F.J.; Ramirez, A.; Valero, M.; Fernandez, E., "Dynamically Controlled Resource Allocation in SMT Processors," *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on* , vol., no.pp. 171- 182, 04-08 Dec. 2004

[Ort03] Ortiz-Arroyo D. and Lee B., "Dynamic Simultaneous Multithreaded Architecture," *16th International Conference on Parallel and Distributed Computing Systems (PDCS-2003)*, August 13-15, 2003.