RZ 3531 (# 99544) 01/23/04
Electrical Engineering 4 pages

Research Report

On Networking Multithreaded Processor Design: Hardware Thread Prioritization

Andreas Döring and Maria Gabrani

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
{ado,mga}@zurich.ibm.com

ON NETWORKING MULTITHREADED PROCESSOR DESIGN: HARDWARE THREAD PRIORITIZATION

Andreas Döring and Maria Gabrani
IBM Zurich Research Laboratory
Säumerstrasse 4, CH-8803, Rüschlikon, Switzerland
{ado, mga}@zurich.ibm.com

Abstract– *Packet processing applications in networking equipment must fulfill very high throughput requirements. At the same time, packet processing differentiation by means of packet classification, such as voice vs. e-mail, or DiffServ, must be obeyed. An efficient way to fulfill both requirements is to use numerous hardware threads combined with thread prioritization. This paper proposes a new thread prioritization method for a hardware multithreaded processor. The originality of the method is identified in the derivation mechanism of the thread priorities, which is based on inputs from three distinct sources; namely, the threads themselves, a control unit such as an operating system, and external sources such as timers or synchronization coprocessors. These sources are explicitly selected to fulfill the requirements of their distinct nature, namely software, middleware and hardware. The proposed method achieves the desired thread differentiation without hindering performance or increasing costs, as demonstrated by initial experimental results.*

I. Introduction

The main drivers for hardware multithreading are the reduction of the time a thread stalls and of the number of context switches as well as the increase of the average utilization of a processor's computation units. A thread stalls while waiting for a response from the memory or a coprocessor or due to synchronization issues. This delay is imposed by the increasing gap between the processor cycle frequency and the memory and communication latency. Software multithreading can reduce the stalls but with coarser response times. Context switches are expensive and should be performed only when no other option exists. In a processor an average of 10% of its computational units are used. With hardware multithreading and the appropriate scheduling, the average utilization of the computational units increases.

In a processor or processor core with hardware multithreading support (MTP), several threads can be executed concurrently. The threads that are not stalled compete for the processing resources such as arithmetic-logic units or memory interfaces. This competition has to be resolved by selecting a thread whose instructions will be executed next. In such processors, if there is no control over the way instructions are issued, a single thread can be executed slower than it would be on a single-threaded processor system.

MTPs are typically designed for systems with real-time constraints, such as network and media processors. In these fields the execution time requirements of threads may vary depending on several issues. In many systems, tasks of varying importance are assigned to the threads of a processor. For example a control task is typically more urgent than a data-processing task. The real-time requirements may be quite diverse. For instance, the latency of voice samples in an intermediate network system is more critical than that of an e-mail. Similarly, quality-of-service demands that certain tasks be prioritized over others. Therefore it is important to guarantee thread execution differentiation.

These requirements are even more eminent for processors in the networking domain, such as network processors and controllers. Reasons for this include the operational time horizon of these processors, which is ruled by the Gb/s rates, the additive latency per node, and the cost of buffer space, among others. To that end, a mechanism that controls the way instructions are issued from threads is of high value. This instruction selection mechanism has to guarantee that the most important tasks are executed first. At the same time, the effort of the control of instruction selection should be very low, because it impacts the total amount of processing capacities for the application.

Accordingly, the problem this paper deals with is twofold: first, how to define an effective thread prioritization scheme and, second, how to do so at minimal cost.

The paper answers the first question by (a) considering three separate sources that influence the priority of a thread in a distinct way, and (b) by introducing a novel way of combining these sources to create a priority value for each thread, called the physical priority. This value is used by the instruction issue unit in the processor. The three sources are the application, a processor control unit, for example as part of an operating system, and the external input, all of which meet the requirements of their different natures, namely software, middleware and hardware.

The paper answers the second question by (a) designing a hardware component that realizes the thread prioritization method described above, and (b) designing the component in such a way that the overall cost is limited.

The paper is organized as follows. We start by outlining the related work. In Section II we introduce the proposed method for hardware thread prioritization. Implementation issues of the method are described in Section III. The simulator used and the first results on thread differentiation, performance and cost are discussed in Section IV. We close the paper in Section V with a summary and directions for further study.

A. *Related work*

Only recently have multithreaded processors appeared in products, especially in network processors, e.g., [1], [4]. When there are few threads, the priority control is less important, because the probability that more than one thread is not stalled is strictly lower. Nevertheless, the use of priorities for instruction selection has been discussed but the aspect of low overhead for the software was not considered.

The current methodologies investigated and discussed in the literature concentrate on improving the overall throughput of a multithreaded processor, e.g., [6], [10]. In the area of thread selection few solutions based on the notion of thread prioritization exist. Those that exist are mainly based on software mechanisms, e.g., [9]. The state-of-the art study revealed only few hardware solutions. These solutions are either highly software dependent [5], or they use thread prioritization for performance enhancement and not for guaranteed thread differentiation [2], [3], [10].

II. The ACE Method

We have devised a thread-selection mechanism that determines the physical priorities of threads from three distinct sources; namely the Application, the Control unit and External, ACE. Each of these sources has a unique knowledge of the various contributors to the system's performance.

More specifically, let us assume that the normal execution of a thread uses a medium priority from the view of the thread. In some situations the thread may know that the following execution is of lower priority, for example when the thread requests an external resource it will need at a later point and has other work to do first. In such a case it may be favorable to run the thread with a lower priority, and therefore normally with a lower instruction rate, instead of running it first with normal priority and then having to wait, for the requested item. In contrast, when the thread has occupied a critical resource, such as a semaphore of a frequently used data object, it can increase its priority to a higher level to reduce the pressure on this resource.

This can be accomplished by adding a single instruction. This instruction can be introduced either by the programmer or automatically by a tool based on a formal program analysis of profiling results from simulated or actual execution runs. To allow all threads to execute the same code and reduce the overhead of the priority modification, a uniform way of accessing the thread's priority contribution is desirable. This is accomplished by means of dedicated registers accessed via existing instructions, e.g., special-purpose registers (SPR) or device control registers (DCR) in a PowerPC processor [8]. The idea is that all threads use the same register number and the hardware incorporates the identity of the thread that executed the instruction.

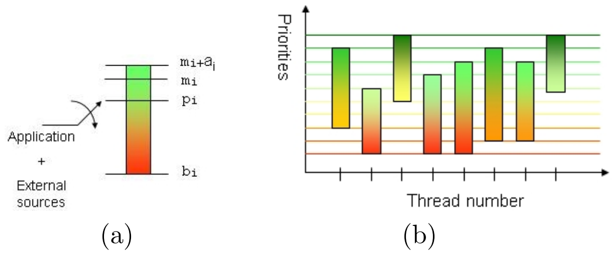

Figure 1
(a) Range of physical priorities for a single thread; (b) relative priority ranges for multiple threads;

However, a thread usually does not know which other threads run on the same processor. Therefore, there may be threads with tasks of much higher or lower importance. To take this into account, a control unit that controls the threads on the processors sets basic b_i and maximum m_i priorities for each thread i, see Figure 1. The maximum priorities are used to ensure that one tread will not starve the others. The basic priorities assist in maintaining a balance among the relative thread priorities assigned by the control unit. If the priority value set by the thread itself is s_i, the resulting priority is $r_i = \min(m_i, b_i + s_i)$.

The third contribution before the physical priority is determined comes from external sources. One example is a synchronization coprocessor. When it detects that another thread requests a semaphore occupied by the considered thread i, it may boost the priority temporarily over the normal bound m_i. In line with the mechanism employed for the threads' contribution, the control unit introduces one more register a_i. If the value delivered from the external source is e_i, the contribution to the physical priority is $p_i = \min(a_i, e_i) + r_i$, with r_i as above. Note that the maximum priority a thread can reach this way is $m_i + a_i$. If several external sources are used, their values can be combined either by adding them or using their maximum.

We use the above sources to derive a physical priority for the threads of a system. Note that this thread prioritization is possible at different stages of the thread execution, namely instruction fetch, instruction decode, register rename, operand fetch, instruction dispatch, instruction issue and instruction completion. In other words, the method we propose is processor architecture independent, as illustrated in Figure 2, where we use the processor architecture of [10].

The proposed component ACE communicates with the CPU, i.e., application and control unit, by providing thread priorities to the instruction selection and by receiving values from the CPU and from external sources. Values can be transferred from the CPU to the ACE component via registers in the proposed component. As described above, the application software and the control unit may communicate with the proposed structure by using existing instructions such as load and store or access to dedicated registers. Otherwise, the communication can take place by mapping the control registers.

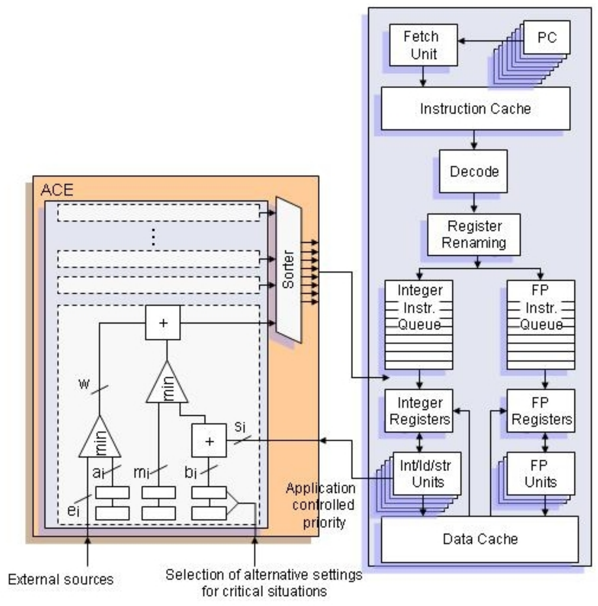

Figure 2
The ACE component and its interaction with an MTP.

III. The ACE Component

The proposed ACE component comprises a basic structure, which consists of a set of registers for storing the contributions of the several inputs, two adders, and two minimum functions; see Figure 2. This structure is repeated for every hardware context in the processor. The registers, adders and minimum functions work on bit vectors of a common resolution w. An appropriate value for the resolution depends on the intended applications and the number of threads. A range of 8 to 16 is expected to be appropriate. As the priorities are defined in terms of relative numbers, the desirable resolution can be attained with a resolution-setting circuit such as a rounder or a sorter.

In Figure 2 each register (b_i, a_i, m_i) is drawn twice. This illustrates a proposed feature of the prioritization component of containing several sets of registers that can be switched very fast. In this way in exceptional situations, e.g., error handling, an appropriate configuration can be established very quickly. The previous configuration used for normal operation is conserved and can be reactivated after the exceptional situation is resolved. In a network processor, there is frequently a control point that supervises numerous other processors and may switch between the normal and the exceptional operation register set for some or all processors.

IV. Experimental Results

Several questions have to be answered to demonstrate the value of the ACE approach for a given application. The most important questions are: (a) How strong is the steering impact of the priorities on a complex multi-issue superscalar processor core? (b) Is the total throughput impacted by the preference of a particular thread? (c) How does the priority-based thread selection relate

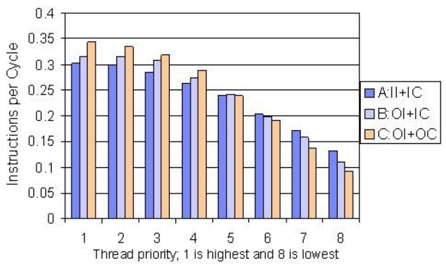

Figure 3
Three cases of thread differentiation.

to other instruction selection policies, e.g., in- or out-of-order instruction issue completion?

In order to emphasize the processor independence of this approach, we used an abstract simulation model based on a stochastic instruction stream (compare with [7]). Only the instruction issue, execution and completion are really simulated, whereas the instruction fetch, decoding and register renaming are covered by the stochastic model. The model of an instruction stream consists of a circular sequence of phases, in which each phase is characterized by the distribution of instruction types (e.g., load instruction or integer instruction) and the access probabilities for source and destination registers. The processing resources and timing (number of pipelines, acceptable instruction types per pipeline, size of issue buffer, issue and completion width etc.) can be configured via parameter data sets provided as an input to the simulator.

With respect to questions (a) and (b), Figure 3 shows the performance of eight threads for in-order issue/in-order completion(II), out-of-order-issue/in-order completion (OI), and both out-of-order issue and completion(OO). As can be seen in all three cases the performance in terms of completed instructions per cycle (IPC) is highest for the thread with the highest priority and monotonically decreases towards lower priorities. That is, the priorities (applied only at instruction issue) are effective in controlling which thread achieves the highest performance. The more freedom there is regarding the selection of instructions (out-of-order issue and/or completion), the stronger the effect, whereas the total performance remains constant (1.90, 1.92 and 1.94 IPC for the three issue/completion policies). The same applies for the percentage of discarded instructions due to mispredicted branches (0.34 for II, 0.29 for OI and OO).

In another simulation series (Figure 4) we modify the number of active threads. As can be seen, the performance of the highest-priority thread stays nearly constant while the total throughput is increased. Only for a higher number of threads is a decrease of the highest priority thread observed. The cause of this could be analyzed to lie in the constant number of slots for outstanding loads. If the percentage of loads with cache misses is decreased or the number of load slots is increased, the competition for the load slots is reduced, and there are

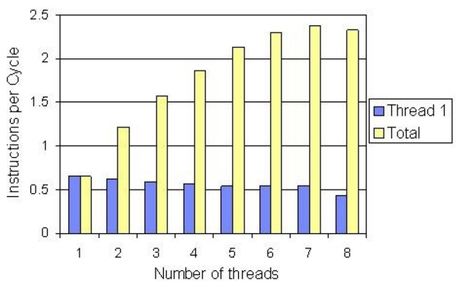

Figure 4
Total performance and performance of highest priority thread for different thread numbers.

fewer stalls at the load/store pipelines. We think that such a resource increase is a better solution than applying priorities within the pipelines. However, we plan more simulations to explore this issue. The applied workload for the results shown here is synthetic, oriented toward in-house experience and the CommBench [11]. The comparably low instruction rate is due to the combination of a high fraction of load instructions with data cache misses (1%) and the memory latency of 120 clock cycles. These assumptions are based on the expected clock frequency of 6 GHz in the next generation of processor cores and the cost-motivated use of standard memory (SDRAM), which is also used by other units. Simulations with lower memory latency (not presented here owing to space limitations) show an expected higher performance for the first thread, and consequently nearly reach the limit of three IPC imposed by the assumed issue width.

The prototype implementation with 10-bit resolution of the ACE unit in 0.13 μm CMOS standard cell technology required 0.015 mm^2 area per thread including the cost for the insertion sorter. This indicates that the cost implied by the ACE approach itself is very low compared to the cost for each thread in the processor core (e.g., register set), which is considerably higher. Due to the lower clock frequency in the ACE unit the power consumption is not an issue.

V. Conclusions

We have introduced a novel thread prioritization mechanism, called ACE, that combines three sources to influence a thread's priority in a distinct and unique way. The three sources are the application, a control unit, and external input that meet the requirements of their different natures, namely software, middleware, and hardware. The input of these sources is structured in such a way that differentiation between threads can be obtained in a relative way, which considerably simplifies the decision and input update process. The impact of the various sources on each thread's priority can be controlled by a control unit, e.g. operating system, without invoking the control program after every change of a source's contribution. This is done by assigning soft priority ranges per thread. The priority ranges are restricted by hard bounds to avoid thread starvation. The priorities used for instruction selection can be modified from the mentioned sources with minimal software or hardware effort. The ACE method is processor architecture independent and can be introduced to a core with low design effort. It can be used with a wide range of hardware thread numbers and a selected number of sources to accommodate different customer requirements.

We have presented initial experimental results. We have implemented a stochastic model of the architecture of [10] and a preliminary version of the ACE mechanism that incorporates only the input from the control unit. Our simulation results clearly illustrate that the ACE approach achieves stronger and guaranteed differentiation of threads with different priorities without impairing the processor's performance.

Our initial results are promising. We are currently working towards adding the other two sources in our simulator. We also intend to test the ACE method with a more accurate processor model. Finally, we aim to combine our approach with existing performance-boosting and cost-reducing techniques, such as those presented in [3] and [10], to evaluate their co-functioning.

VI. Acknowledgments

The authors thank Patricia Sagmeister for her valuable input on system modeling and performance evaluation.

VII. References

[1] Adiletta, M., Rosenbluth, M., Bernstein, D., Wolrich, G., and Wilkinson, H., "The Next Generation of Intel IXP Network Processors", Intel Technology Journal, Vol. 06, No. 03, August 15, 2002.

[2] Brinkschulte, U., Kreuzinger, J., Pfeffer, M., and Ungerer, T., "A Scheduling Technique Providing a Strict Isolation of Real-time Threads", *Proc. of Seventh IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'02)*, San Diego, CA, January 7-9, 2002.

[3] Dorai, G., K., Yeung, D., and Choi, S., "Optimizing SMT Processors for High Single-Thread Performance", *Journal of Instruction Level Paralellism*, Vol. 5, April 2003, pp. 1-35.

[4] IBM Corporation, "Network Processor 4GS3 Overview", *Application Note*, October 1999.

[5] Kimura, K., Kiyohara, T., and Yoshioka, K., "Multithreaded Processor for Processing Multiple Instruction Streams Independently of Each Other by Flexibly Controlling Throughput in Each Instruction Stream", US Patent 6105127, Aug., 1997.

[6] Luo, K., Mukherjee, S., S., and Senze, A., "Boosting SMT Performance by Speculation Control", *Proc. of 15th Int. Parallel and Distribited Processing Sysmposium (IPDPS'01)*, San Francisco, CA, April 23-27, 2001.

[7] Nussbaum, S., and Smith, J., E., "Modeling Superscalar Processors via Statistical Simulation", *Proc. of Int. Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, Barcelona, Spain, September 8-12, 2001.

[8] www.ibm.com/chips/techlib/techlib.nsf/productfamilies/ /PowerPC

[9] Snavely, A., Tullsen, D., M., and Voelker, G., "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor", *Proc. of Int. Conf. on Measurement and Modelling of Computer Systems (Sigmetrics 2002)*, Marina Del Rey, CA, June 15-19, 2002.

[10] Tullsen, D., M., Eggers, S., J., Emer, J., S., Levy, H., M., Lo, J., L., and Stamm, R., L., "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", *Proc. of the 23rd Int. Symposium on Computer Architecture (ISCA'96)*, Philadelphia, PA, May 22-24, 1996, pp. 191–202.

[11] Wolf, T., and Franklin, M., "CommBench - a telecommunications benchmark for network processors", *Proc. of IEEE Int. Symposium on Performance Analysis of Systems and Software*, Austin, TX, Apr. 2000, pp. 154-162.