Petri Net Analysis of Non-Redundant and Redundant Execution Schemes

Stefan Einer, Schweizerische Bundesbahnen, 3000 Bern, Switzerland, stefan.einer@sbb.ch
Bernhard Fechner, FernUniversität in Hagen, 58084 Hagen, Germany, bernhard.fechner@fernuni-hagen.de
Jörg Keller, FernUniversität in Hagen, 58084 Hagen, Germany, joerg.keller@fernuni-hagen.de

Abstract

The quest for high-performance has led to multi- and many-core systems. To push the performance of a single core to the limit, simultaneous multithreading (SMT) is used. SMT enables to fetch different instructions from different threads, hiding latencies in other threads. SMT also gives the opportunity to execute redundant threads (redundant multithreading, RMT) and thus to detect faults by comparing the results of both threads. The instruction fetch algorithm determines which instructions to fetch from which thread and therefore has great influence on processor performance. This work investigates the influence of different instruction fetch algorithms on the performance of an SMT processor by modeling it with Petri nets. Over the intrinsic results of a detailed processor simulation, our approach offers a generic evaluation. Furthermore, we distinguish between homogeneous (redundant execution, RMT) and inhomogeneous threads to determine the effects on the performance of each execution scheme with a dedicated instruction fetch algorithm. For inhomogeneous threads, the effect of instruction fetch algorithms can be confirmed, but not for homogeneous threads. Therefore, scheduling algorithms as simple as Round Robin can be recommended for redundant execution.

1. Introduction

Today's microprocessors execute programs in a parallel superscalar fashion. The degree of parallelization is limited by the available resources and data dependencies. A technique to push the performance of single-core processors to the limit is multithreading. Here, the instructions of multiple programs are partitioned into threads, being independent instruction streams. Dependencies and thus latencies within a thread can be hidden by switching the processor context between threads (Simultaneous Multithreading, SMT). SMT is able to feed instructions of different threads to different execution units. A major problem is to choose the appropriate instruction to execute from the I-Cache. It seems reasonable to consider the processor's inner state to decide what instruction to schedule onto the instruction window. This is done through instruction scheduling. The scheduling significantly determines the performance of an SMT processor.

In this work, we investigate the influence of different instruction fetch algorithms on the performance of an SMT processor by modeling it with Petri nets. Additionally, we examine homogenous and inhomogeneous threads.

Typically, the threads executed on an SMT processor differ from each other (inhomogeneous threads). The idea to execute homogeneous threads on an SMT-processor was first presented by [3]. There, a leading (active) and a trailing (redundant) thread execute the same process. The results are compared at the end of the execution. On a difference, an error is signaled. The work from [4] presents and evaluates different instruction fetch algorithms for SMT processors. Their experiments are based on a simulation of a specific processor. The behavior of the underlying processor is modeled in every detail. Thus, the model is complex and the experimental results intrinsic. A more generic approach is to evaluate the processor performance through simulation [6]. Here, an SMT processor is modeled by Petri nets. Our work is inspired by Zuberek's approach. A Petri net was developed to evaluate different instruction scheduling algorithms, including the behavioral model of an SMT processor. From this, data of unknown behavioral phenomena, including performance, can be derived. We distinguish between redundant execution and multi-programmed workloads.

The remainder of this article is organized as follows. In Section 2, we briefly describe instruction fetch algorithms and their evaluation criteria. In Section 3, we present our Petri net model of an SMT processor, and present and evaluate the simulation results in Section 4. Section 5 concludes the paper.

2. Instruction scheduling for SMT

The instruction fetch phase within a processor works optimal if the following criteria are fulfilled. A minimum number of instructions is fetched after a conditional branch. The waiting time of instructions in the instruction window is as small as possible to avoid congestions. As a consequence, the processing units can be provided continuously with instructions. To approximate these criteria, several strategies exist [4]. In the following, the four strategies to be analyzed are described.

Round Robin statically chooses the next thread cycle-by-cycle without considering the inner state of the processor. For the sake of simplicity it ignores the potential of dynamic scheduling strategies, but it was implemented in current processors and serves as a reference.

BRCOUNT counts the branch instructions in several pipeline stages per thread including the instruction window. The thread with the smallest number of branch instructions will be selected for the execution. The latency of control-flow conflicts is hidden.

MISSCOUNT tries to hide latencies caused by data cache misses. Data dependent instructions will have to wait in the instruction window until the data has been loaded. *MISSCOUNT* counts the number of data cache misses per thread. The thread which has the smallest number of cache misses has the highest priority.

ICOUNT counts the number of instructions in several stages. The thread with the smallest number of instructions is preferred to execute next.

Two additional parameters determine the performance of an instruction fetch algorithm in each cycle:

- P_1: The maximum number of threads that the fetch unit is able to fetch instructions from.
- P_2: The number of instructions that the fetch unit is able to extract from each thread.

Instruction fetch algorithms also differ in the following points:

- A constant set of instructions can be fetched within each cycle from a constant set of threads. Here, $P_1 \cdot P_2 = b$, where b equals the total bandwidth of the instruction fetch unit.
- The context is changed if no instructions can be fetched from the actual thread ($P_1 \cdot P_2 > b$).

The losses in efficiency can be determined through a comparison with an ideal (theoretical) parallelization. The values *I-Cache-V*, *IQ-Clog-V* and *Multipath-V* express this degradation. *I-Cache-V* occurs if no executable instructions are in the instruction cache. The execution will be prolonged in comparison with an ideal parallelization. *IQ-Clog-V* occurs if the instruction window is full. *Multipath-V* is caused by latencies of conditional branches. It comprises speculatively executed instructions, which are not relevant for the program's state, since the speculation lead to an irrelevant path.

3. Generation of Models

Petri nets are a widespread means for modeling, originally used for communicating automata [2]. Later, specializations and extensions evolved; cf. e.g. [1]. In the present investigation we use *Stochastic Colored Petri Nets (SCPN)* and the tool TimeNet [5]. Petri nets are bipartite graphs where nodes are either places or transitions. Places in a Petri net can contain tokens. Transitions can fire according to some rules and thus remove tokens from places and/or put them onto places. This token game allows changing the state of the Petri net. Thus Petri nets are useful to describe dynamic systems and simulate their behavior.

In our SCPN, we model the different stages during the processing of an instruction as a sequence of linked subnets. The instruction fetch algorithm to be used is activated before simulation. Tokens are used for the instruction and counter values. They also mark the availability of resources and the selection of threads (cf. Figure 1).

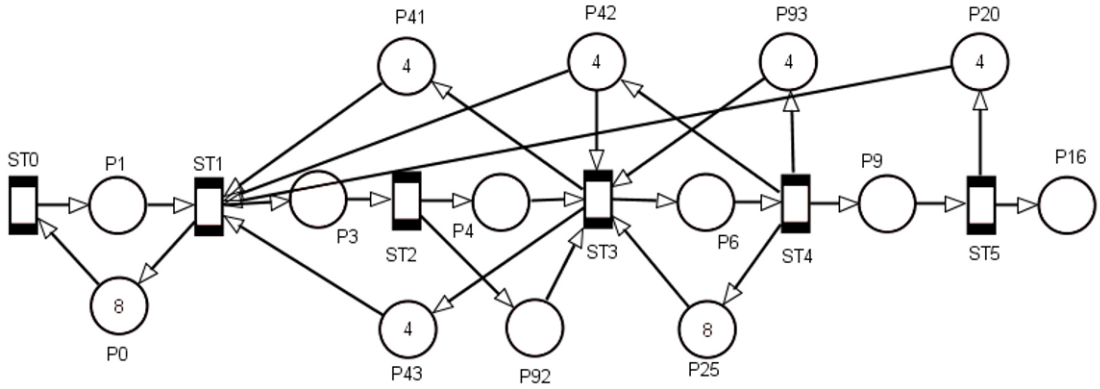

Figure 1: The Abstract Petri Net for Simulation

The subnets to describe the stages differ in complexity. They comprise the models for the instruction fetch algorithms, the instruction window and a generator for instructions which is specific for our model. The model assumes that multi-path execution is used to resolve control flow conflicts, instead of applying simple branch speculation. Moreover, we consider the scheduling strategy used to avoid dispatch of instructions from the instruction window that belong to a thread currently subject to a data cache miss.

The model has the following parameters:

1. Probabilities for conditional branches and misses in instruction and data caches
2. Latencies
3. Instruction window size

We model the instruction fetch algorithms from Section 2 as a 1.8 variant, i.e. in every cycle we try to take 8 instructions from one thread.

4. Simulation Results and Evaluation

The values for the model parameters were chosen in a straightforward manner. As the qualitative evaluation is mostly independent of the particular values, we refrain from discussing them. We only distinguish whether the parameter values (probabilities or time spans) are identical for all threads or can be different for each thread. Thus we obtain simulations with *homogeneous* or *inhomogeneous threads*, respectively. The distinction can be explained in a simplified manner as threads belonging to the redundant execution of the same program or threads belonging to different programs.

Interestingly, the simulation with homogeneous threads indicates no clear preference for a particular instruction fetch algorithm, i.e. losses and throughput vary from run to run in the same manner for all algorithms. The presence of variations indicates that the performance of the instruction fetch algorithms is dependent on the particular details and circumstances of each run. Only the simulations with inhomogeneous threads reveal preferences for particular instruction fetch algorithms, as indicated by the exemplary simulation result depicted in Figure 2. As expected by [4], the IQ-Clog loss is smallest with I-Count, followed by MISSCOUNT. On the other hand, these algorithms show more I-Cache loss than Round Robin or BRCOUNT. We conclude that for redundant execution, simple algorithms for the instruction fetch are sufficient.

5. Conclusions

We have presented how to model and simulate the behavior of an SMT processor with several instruction fetch algorithms, including redundant and non-redundant execution. Petri net modeling has the advantage that it avoids emulation of the SMT processor and already includes its known behavioral patterns as features. In contrast to a previous evaluation, our simulation distinguishes between homogeneous (redundant) [3] and inhomogeneous threads and different instruction fetch algorithms. For inhomogeneous threads, we could confirm the effect of instruction fetch algorithms, but not for homogeneous threads. Instead, our results indicate that the effect of an instruction fetch algorithm is strongly dependent from a particular program run.

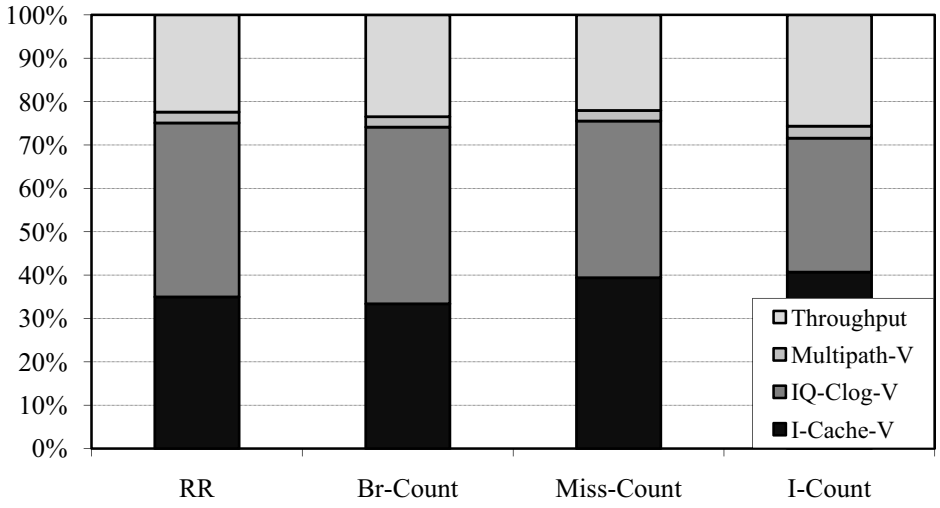

Figure 2: Comparison of Instruction Fetch Algorithms

As a consequence, a processor using SMT to provide fault tolerance by redundantly executing one thread can be less complex than a typical SMT processor, by choosing a simple scheduling algorithm such as Round Robin.

References

[1] Einer, S.: Petri netzbasierte Spezifikation und Analyse operationaler Prozesse am Beispiel Eisenbahnsicherung. Dissertation, TU Braunschweig, Fortschritt-Berichte, Reihe 20, Nr. 373, VDI Verlag, 2003.

[2] Petri, C. A.: Kommunikation mit Automaten. Dissertation im Fachbereich für Mathematik und Physik der TH Darmstadt, 1962.

[3] Rotenberg, E. 1999. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proceedings of the Twenty-Ninth Annual international Symposium on Fault-Tolerant Computing* (June 15 - 18, 1999). FTCS. IEEE Computer Society, Washington, DC, 84.

[4] Tullsen, D. M. et al.: Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. Proceedings of the 23rd Annual International Symposium on Computer Architecture, Philadelphia, 1996.

[5] Zimmermann, A.; Knoke, M.: TimeNet 4.0 – A Software Tool for the Performability Evaluation with Stochastic and Colored Petri Nets – User Manual. TU Berlin, Faculty of EE&CS Technical Report 2007-13, ISSN: 1436-9915, Berlin, 2007.

[6] Zuberek, W.M.: Modeling and Analysis of Simultaneous Multithreading. 14th Int'l. Conf. on Analytical and Stochastic Modeling Techniques and Applications (ASMTA'07), Prag, 2007.