# Towards a Statistical Model of a Microprocessor's Throughput by Analyzing Pipeline Stalls

Uwe Brinkschulte, Daniel Lohn, and Mathias Pacher

Institut für Informatik
Johann Wolfgang Goethe Universität Frankfurt, Germany
{brinks,lohn,pacher}@es.cs.uni-frankfurt.de

**Abstract.** In this paper we model a thread's throughput, the *instruction per cycle* rate (IPC rate), running on a general microprocessor as used in common embedded systems. Our model is not limited to a particular microprocessor because our aim is to develop a general model which can be adapted thus fitting to different microprocessor architectures. We include stalls caused by different pipeline obstacles like data dependencies, branch misprediction etc. These stalls involve latency clock cycles blocking the processor. We also describe each kind of stall in detail and develop a statistical model for the throughput including the entire processor pipeline.

## 1 Introduction

Nowadays, the development of embedded and ubiquitous systems is strongly advancing. We find microprocessors embedded and networked in all areas of life, e.g. in cell phones, cars, planes, and household aids. In many of these areas the microprocessors need special capabilities, e.g. guaranteeing execution time bounds for real-time applications like a control task on an autonomous guided vehicle. Therefore, we need models of the timing behavior of these microprocessors by which the execution time bounds can be computed.

In this paper we develop a statistical model for the IPC rate of a general purpose multi-threaded microprocessor to predict timing behavior thus improving the real-time capability. We consider both effects like data dependencies and processor speed-up techniques like branch- and branch target prediction, or caches. The model is a transfer function computing the IPC rate. By analyzing this model we obtain bounds for the IPC rate which can be used to compute bounds for the execution time of user applications. Another important use of a model like this is to control the IPC rate similar to [1,2,3]. Controlling the IPC rate in pipelined microprocessors is one of the long-term goals of our work: If we develop precise statistical models of the throughput we are able to adjust the controller parameters in a very fine-grained way. In addition, we can compute estimations for the applications' time bounds which is necessary for real-time systems.

The paper is structured as follows: Section 2 presents related work and similar approaches. In Section 3 we discuss modern scalar and multi-threaded microprocessors and in Section 4 we present our model which is validated by an example. Section 5 concludes our paper and gives an outlook to the future work.

## 2 State of the Art

Many approaches for Worst Case Execution Time (WCET) analysis are known. Most of them examine the semantics of the program code in respect to the pipeline used for the execution resulting in a cycle accurate analysis of the program code. One example is the work in [4]. The authors examine the WCET in the Motorola ColdFire 5307 and study i.e. cache interferences occurring while loop execution.

In [5], the authors discuss the WCET analysis in an out-of-order execution processor. They transform the WCET analysis problem by computing and examining the execution graph (whose nodes represent the tuple consisting of an instruction identifier and a pipeline stage) of the program code to be executed.

The authors of [6] consider the WCET analysis for processors with branch prediction. They classify each control transfer instruction with respect to branch prediction and use a timing analyzer to estimate the WCET according to the instruction classification.

A WCET analysis with respect to industrial requirements is discussed in [7].

A good review on existing WCET analysis techniques is given in [8]. The authors also present a generic framework for WCET analysis.

In [9], the author propose to split the cache into several independent caches to simplify the WCET analysis and get tighter upper bounds.

The authors of [10] design a model of a single-issue in-order pipeline for a static WCET analysis and consider time dependencies between instructions.

The papers presented above mostly provide cycle accurate techniques for WCET analysis of program codes. This is different to our approach as we use a probabilistic approach based on the pipeline structure. The characteristics of the program codes are generalized by statistical values like the probability of a misprediction etc. As a result, our model is not necessarily cycle accurate but we are able to use analytical techniques to examine the throughput behavior for individual programs as well as for program classes. Furthermore, as mentioned in the section 1, our long-term goal is to control the IPC rate. Using control theory as a basis, a model of a processor as proposed in this paper is necessary not only to analyze, but as well to improve and guarantee real-time behavior by a closed control loop.

## 3 Pipeline Stalls in Microprocessors

Techniques like long pipelines, branch prediction and caches were developed to improve the average performance of modern super-scalar microprocessors. But the worst case oriented real-time behavior suffers from various reasons like branch misprediction, cache misses, etc. Besides, there is only one set of registers in single-threaded processors, thus producing context switch costs of several clock cycles in case of a thread switch.

On application level thread synchronization also introduces latency clock cycles if one thread has to wait on a synchronization event. This problem depends on the programming model and is not affected by the architecture of the microprocessor used for its execution.

Multi-threaded processors suffer from the same problems considering real-time as single-threaded processors. However, there are several differences which makes them an interesting research platform to model the IPC rate: Contrary to single-threaded processors there are mostly separated internal resources in multi-threaded processors like program counters, status- and general purpose registers etc. for each thread. This decreases the interdependencies of different threads caused by the processor hardware. The remaining thread interdependencies only depend on the programming model, thus e.g. the context switching time between different threads is eliminated.

In addition, if a scheduling strategy like Guaranteed Percentage Scheduling (GP-Scheduling, see [11]) is used, the controller is able to to control each thread in a fine-grained way. In GP scheduling, a requested number of clock cycles is assigned to each thread. This assignment is guaranteed within a certain time period (e.g. 100 clock cycles).

Fig. 1 gives an example of three threads with a GP rate of 30% for Thread A, 20% for Thread B, and 40% for Thread C, and a time period of 100 clock cycles. This means, thread A gets 30 clock cycles, thread B gets 20 clock cycles, and thread C gets 40 clock cycles within the 100 clock cycle time period.

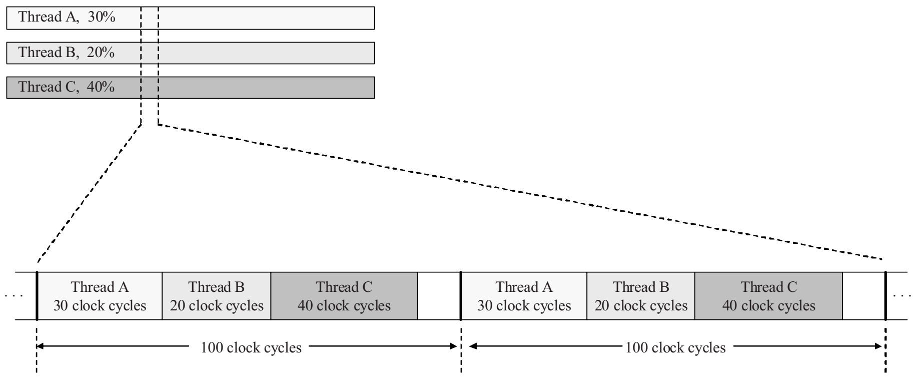

**Fig. 1.** Example for a GP schedule

## 4 Modeling

In this section we present a statistical model of a microprocessor evolving the parameters influencing the throughput. Our first approach considers a processor with one core and a simple scalar multi-threaded pipeline. Our analysis of throughput hazards starts with the lowest hardware level, the pipeline.

Every instruction to be executed passes the pipeline. Therefore, we have to consider the first pipeline stage, the instruction fetch unit. This stage exists in almost all modern microprocessors [11,12]. The instruction set of a processor can be divided into different classes: An instruction is either controlflow or data related. We can compute a probability for the occurrence of instructions from these classes. The probability of the occurrence of a controlflow class instruction in the interval $n$ is denoted by $p_a(n)$, while $p_b(n)$ represents the probability of a data related instruction in the interval $n$. We assume the probabilities to be time dependent, because they may change with the currently executed program code.

First, we consider controlflow related instructions like unconditional and conditional branches. These instructions may lead to a lower IPC rate, caused by delay cycles in the pipeline. Therefore, it is necessary to identify them as early as possible and handle them appropriately[1]. This is done with the help of a branch target buffer (BTB) [11]. The BTB contains the target addresses of unconditional branches, and some additional prediction bits for conditional branches to predict the branch direction. Whenever the target address of an instruction can't be found in the BTB, the pipeline has to be stalled until the target address has been computed. Therefore, we model these delay cycles by a penalty $D_{a^{target}}$, while $p_{a^{target}}(n)$ is the probability that such a stall event occurs in the time interval $n$.

If a conditional branch is fetched, the predictor may fail. In this case the pipeline has to be flushed and the instructions of the other branch direction have to be fed into the pipeline mostly leading to a long pipeline stall. The actual number of delay cycles depends on the length of the pipeline [11]. We call $p_{a^{mp}}(n)$ the probability a branch is mispredicted in the interval $n$ and $D_{a^{mp}}$ the penalty in delay cycles for pipeline flushing.

Now, we consider data related instructions because data dependencies also influence the IPC rate. There are three different kinds of dependencies [11]: The anti dependency or write-after-read-hazard (WAR) is the easiest one because it does not affect the execution in an in-order-pipeline at all. As an example let's assume an instruction writes to a register that was read earlier by another instruction. This does not influence the execution in any way. Output dependencies or write-after-write-hazards (WAW) can be solved by register renaming thus not affecting the IPC rate, too. True dependencies or read-after-write-hazards (RAW) are the worst kind of data dependencies. Their impact on the IPC rate can be reduced by hardware (forwarding techniques [12]) or by software (instruction reordering [12]). However, in several cases instructions have to wait in the reservation stations and several delay cycles have to be inserted into the pipeline until the dependency is solved. $p_{b^d}$ denotes the statistical probability for a pipeline stalling data dependency and $D_{b^d}$ denotes the average penalty in clock cycles.

[1] A modern microprocessor is able to detect controlflow related instructions yet in the instruction fetch stage of its pipeline.

The following formula (1) computes the IPC rate $I$ of a microprocessor including the above mentioned pipeline obstacles in the interval $n$:

$$\begin{aligned} I(n) =& \frac{G(n)}{1+X(n)} \\ X(n) =& p_a(n)(p_{a^{target}}(n)D_{a^{target}} + p_{a^{mp}}(n)D_{a^{mp}}) + p_b(n)p_{b^d}(n)D_{b^d} \end{aligned} \tag{1}$$

The IPC rate $I(n)$ of the executed thread in the interval $n$ is the Guaranteed Percentage rate $G(n)$ divided by one plus a penalty term $X(n)$, where $X(n)$ is the expected value of all inserted penalty delay cycles.

If we assume a perfect branch prediction and no pipeline stalls caused by data dependencies, then the probabilities for pipeline stalling events would be zero, turning the whole term $X(n)$ into zero. The resulting IPC rate would equal the Guaranteed Percentage rate, due to no latency cycles occurring. However, in case a data dependency could not be solved by forwarding, then $p_{b^d}(n)$ would not be zero and $X(n)$ would contain the penalty for the data dependency. Therefore, the IPC rate would suffer.

Figure 2 shows the impact of those pipeline hazards on the IPC rate.

The next step is to consider the effects of caches on the IPC rate, ignoring any delay cycles from other pipeline stages. Since we have no out-of-order execution, every cache miss leads to a pipeline stall, until the required data or instruction is available. The statistical probability of a cache miss occurring in the interval $n$ is denoted by $p_c(n)$ and $D_c$ is the average penalty in delay cycles.

So the resulting formula is quite similar to formula 1:

$$\begin{aligned} I(n) =& \frac{GP(n)}{1+Y(n)} \\ Y(n) =& p_c(n)D_c \end{aligned} \tag{2}$$

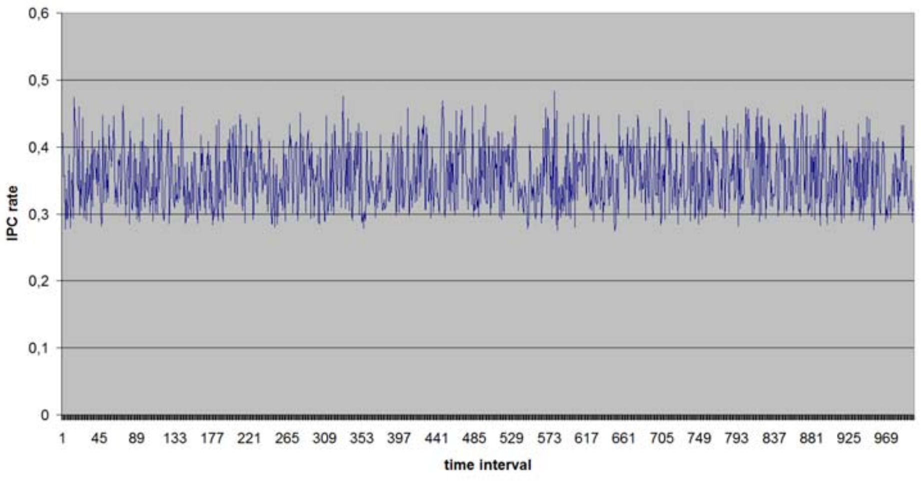

**Fig. 2.** Impact of pipeline hazards on the IPC rate

$Y(n)$ is the expected value of all delay cycles in the interval $n$, lowering the IPC rate $I(n)$. Figure 2 shows the effects of cache misses on the IPC rate.

Our final goal in this paper is to combine the pipeline hazard and the cache miss effects in one formula. As there is no dependency between cache misses and pipeline hazards, all the inserted delay cycles can simply be added, resulting in a final penalty of $Z(n)$.

Thus, we can bring together the effects of pipeline hazards and cache misses leading to the following formula 3:

$$\begin{aligned} I(n) &= \frac{GP(n)}{1+Z(n)} \\ Z(n) &= X(n) + Y(n) \\ X(n) &= p_a(n)(p_{a^{target}}(n)D_{a^{target}} + p_{a^{mp}}(n)D_{a^{mp}}) + p_b(n)p_{b^d}(n)D_{b^d} \\ Y(n) &= p_c(n)D_c \end{aligned} \tag{3}$$

Figure 3 shows the according IPC rate, taking into account all effects on hardware level.

Now, we show that formula 3 is an adequate model of a simple microprocessor. Therefore, we examine a short code fragment of ten instructions executed in the time interval $i$:

1. data instruction
2. controlflow instruction (jump target not known)
3. data instruction
4. data instruction (with dependency)
5. data instruction

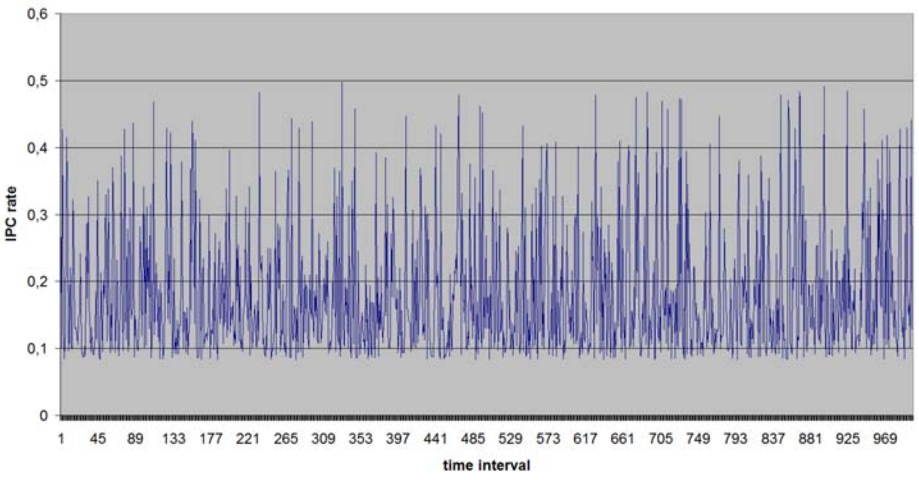

**Fig. 3.** Impact of cache misses on the IPC rate

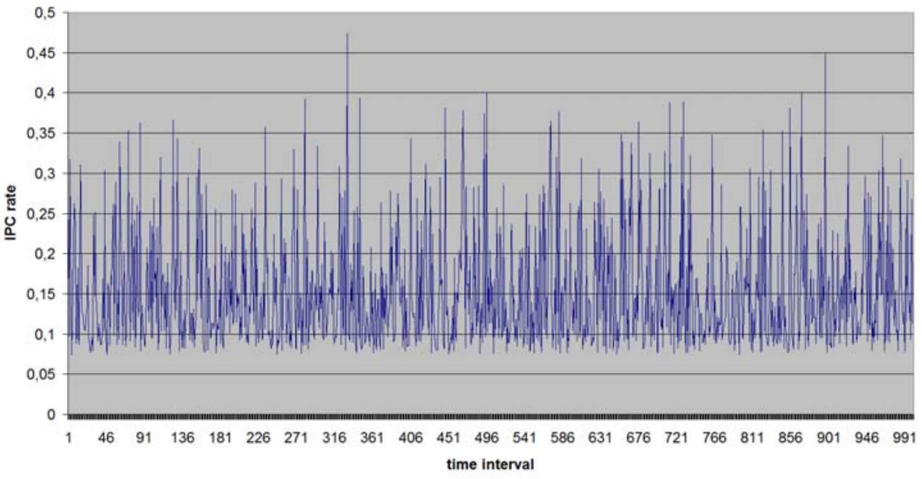

**Fig. 4.** The final IPC rate including pipeline hazards and cache misses

6. data instruction (with dependency)
7. controlflow instruction
8. data instruction (cache miss)
9. data instruction
10. data instruction

We assume our microprocessor has a five stage pipeline and runs two different threads and a Guaranteed Percentage value of 0.5 is granted to each of them . Furthermore, we assume a penalty of 2 clock cycles for an unknown branch target, 5 clock cycles for flushing the pipeline after a mispredicted branch, 1 clock cycle for an unresolved data dependency and 30 clock cycles for a cache miss.

Analyzing the code fragment produces the following probability values:

$p_a(i) = 0.2$
$p_{a^{target}}(i) = 0.1$
$p_{a^{mp}}(i) = 0$
$p_b(i) = 0.8$
$p_{b^d}(i) = 0.25$
$p_c = 0.1$

Having these values we are able to compute the IPC rate according to our model:

$$X(i) = 0.2 \cdot (0.1 \cdot 2 + 0) + 0.8 \cdot 0.25 \cdot 1 = 0.4$$
$$Y(i) = 0.1 \cdot 30 = 3$$
$$Z(i) = 0.4 + 3 = 3.4$$
$$I(i) = \frac{0.5}{1 + 3.4} \approx 0.114$$

To verify the model, we examine what happens on pipeline level. At the beginning of interval $i$ it takes five clock cycles to fill the pipeline. At the 6th clock cycle the first instruction is completed and then the pipeline is stalled for two cycles to compute the branch target of instruction 2. So instruction 2 finishes at the 9th clock cycle. Instruction 3 is completed at the 10th clock cycle and instruction 4 at the 12th clock cycle, because the unresolved data dependency of instruction 4 leads to a pipeline stall of one cycle. At the 13th clock cycle, instruction 5 is finished and at the 15th and 16th clock cycles the instructions 6 and 7, too. Because a cache miss happens during the execution of instruction 8, it finishes at the 47th clock cycle. The last two instructions finish at the 48th and 49th clock cycle.

Since the thread has a GP value of 0.5, we have to double the execution time. This means, the execution of the code fragment would take 98 clock cycles on the real processor. This is already very close to our model (about 10%). If we neglect the first cycles needed to fill the pipeline we even get exactly an IPC rate of $I(i) = \frac{10}{88} \approx 0.114$.

Since real programs consist of many instructions, the time for the first pipeline filling can be easily neglected, thus enabling our model to predict the correct value of the IPC rate.

## 5 Conclusion and Future Work

In this paper we developed a statistical model of a simple multi-threaded microprocessor to compute the throughput of a thread. We started to consider the influence of hardware effects like pipeline hazards or cache misses on the IPC rate. First, we considered each hardware effect on its own, then we combined all together to a single formula, see formula 3. We showed with the help of an example that our model adequately describes the IPC rate.

Future work will concern further improvements of the model, taking into account more advanced hardware techniques, like multicore or out-of-order execution.

As already mentioned above, our future work will not only concern to compute the throughput of a thread, but also to control and stabilize it to a given IPC rate by closed control loops. Therefore, we want to develop a model by what we are able to identify the most important parameters for the IPC rate.

## References

1. Brinkschulte, U., Pacher, M.: A Control Theory Approach to Improve the Real-Time Capability of Multi-Threaded Microprocessors. In: ISORC, pp. 399–404 (2008)
2. Pacher, M., Brinkschulte, U.: Implementing Control Algorithms Within a Multithreaded Java Microcontroller. In: Beigl, M., Lukowicz, P. (eds.) ARCS 2005. LNCS, vol. 3432, pp. 33–49. Springer, Heidelberg (2005)
3. Brinkschulte, U., Pacher, M.: Improving the Real-time Behaviour of a Multithreaded Java Microcontroller by Control Theory and Model Based Latency Prediction. In: WORDS 2005, Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems, Sedona, Arizona, USA (2005)

4. Langenbach, M., Thesing, S., Heckmann, R.: Pipeline modeling for timing analysis. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 294–309. Springer, Heidelberg (2002)
5. Li, X., Roychoudhury, A., Mitra, T.: Modeling out-of-order processors for software timing analysis. In: RTSS 2004: Proceedings of the 25th IEEE International Real-Time Systems Symposium, Washington, DC, USA, pp. 92–103. IEEE Computer Society, Los Alamitos (2004)
6. Colin, A., Puaut, I.: Worst case execution time analysis for a processor withbranch prediction, vol. 18(2/3), pp. 249–274. Kluwer Academic Publishers, Norwell (2000)
7. Ferdinand, C.: Worst case execution time prediction by static program analysis, vol. 3, p. 125a. IEEE Computer Society, Los Alamitos (2004)
8. Kirner, R., Puschner, P.: Classification of WCET analysis techniques. In: Proc. 8th IEEE International Symposium on Object-oriented Real-time distributed Computing, May 2005, pp. 190–199 (2005)
9. Schoeberl, M.: Time-predictable cache organization (2009), `http://www.jopdesign.com/doc/tpcache.pdf`
10. Engblom, J., Jonsson, B.: Processor pipelines and their properties for static wcet analysis. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 334–348. Springer, Heidelberg (2002)
11. Brinkschulte, U., Ungerer, T.: Mikrocontroller und Mikroprozessoren, 2nd edn. Springer, Heidelberg (2007)
12. Hennessy, J.L., Patterson, D.A.: Computer architecture: a quantitative approach, 4th edn. Elsevier [u.a.], Amsterdam (2007); Includes bibliographical references and index