# Static Timing Analysis of Embedded Software on Advanced Processor Architectures

A. Hergenhan and W. Rosenstiel

Forschungszentrum Informatik (FZI)
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany

Universität Tübingen
Sand 13
72076 Tübingen, Germany

{hergenhan, rosenstiel}@fzi.de

## Abstract

*This paper examines several techniques for static timing analysis. In detail, the first part of the paper analyzes the connection of prediction accuracy (worst case execution time) and applicability of a methodology for modeling and analysis of instruction as well as data cache behavior. The second part of the paper proposes a timing analysis technique for super-scalar processors. The objects of our studies are two processors of the* PowerPC *family, in particular the PPC403 and the MPC750.*

## 1. Introduction

The nature of an embedded system is given by its functional as well as its nonfunctional requirements of the specification. Consequently, the design of an embedded system entails several hardware/software codesign problems. Hereby, the first design step is the partitioning into a system architecture, splitting into software and hardware related parts. The partitioning step is of extreme importance, due to the fact that all following design steps are influenced by it.

In order to obtain reasonable results, tools that evaluate, estimate and judge the performance of all possible system architectures in a quick manner should be provided. Therefore, the timing performance analysis of software running on embedded microcontrollers has been of growing importance. If a system does not meet its timing constraints, it does not meet the functional requirements either, because this can have disastrous consequences.

Timing analysis results can be received by *simulation*, *emulation* or *static timing analysis*. Static timing analysis is a technique which collects information about the program structure as well as of the microcontroller at compile time rather than at runtime. Then, this analysis technique should provide a *worst case execution time (wcet)*. Generally, this value should be safe (therefore conservative) and accurate (in order to save resources). Unfortunately, the more complex the embedded microcontrollers the more challenging the task to tighten the analysis results. However, these results might be as exact enough to span a guideline for trade-off decisions for partitioning, as well as to provide initial information for software development within a real-time operating system.

Static timing analysis is a subject of research for some time. Here, the most general approach was presented by LI *et. al.* [4]. But in the end, all previous surveys were either limited to more basic *RISC*-processors [4][6] or to single architectural properties [3][2]. Moreover, most obtained results were validated by simulation and not by using a "real-world" environment. LIM *et. al.* [5] first introduced a analysis technique for multiple-issue processors for their analysis framework *ETS*. But, they restricted their experiments to this feature and did not have meaningful validation results.

That's why we moved systems in the center of our attention which contain processor implementations of the *PowerPC* family. They run on the base of the same (or at least similar) assembler or machine code, but have different architectural features. This made it possible to compare the processor implementations with the prospect to their timing behavior on the one hand, and enabled a true validation of the results of the static timing analysis on the other. For the moment we focused on prediction techniques for memory and bus behavior including instruction and data caching, for multiple-issue pipelines and for the interaction of these characteristics, as well. Further, we examined some of the prediction algorithms to see if they are practicable in any case due to their analysis complexity or if not, if there was a feasible solution in order to decrease the estimation cost.

Thus, the paper is organized as follows. In Section 2 we introduce our research environment. In particular, we explain a generalized view on static analysis which can integrate several approaches of previous work, and we describe the validation environment in detail. The subsequent sections are about our experiments which we mentioned above. Thus, section 3 analyzes the performance effects of instruction caches. In addition, here we surveyed analysis complexity and problems in modeling data cache behavior. Section 4 presents a technique which is applicable to multiple-issue pipelines. Finally, section 5 summarizes the paper and concludes with future works.

## 2. Static timing analysis

### 2.1. Methodology

Slightly contrary to other concepts, we would like to introduce static timing analysis as a process of modeling and analyzing. In detail, it is divided into three major tasks: *program path modeling*, *architectural modeling* and the *analysis composition* (Figure 1).

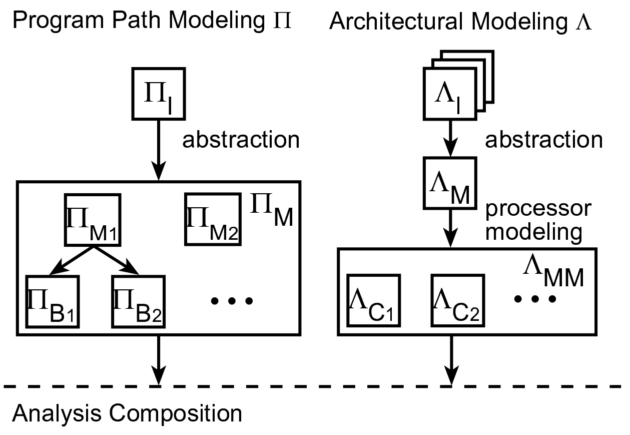

**Figure 1. Static timing analysis methodology**

A program implementation $\Pi_I$ (either machine code or high level language) to be analyzed is translated into an internal representation $\Pi_M$ which depicts the structure of the program. This is either an hierarchical control flow graph ($\Pi_{M1}$) [4] or another hierarchical organization of program constructs ($\Pi_{M2}$)[6]. From this, the base ($\Pi_{B1,2...}$) for timing analysis can be extracted. These are either code sequences formed by basic block segments, line block segments [4], high level program constructs [6], super-block segments or inter-block relations as well. Besides, a mixture of different analysis blocks for static timing analysis is conceivable. This complies with the hierarchical flow graph clustering and treatment of ERNST *et. al.* [1].

In general, static reasoning about worst case timing is an undecidable problem due to the facts, that the executed program trace depends on the initial program state and the applied input values. To make this problem tractable, the program path model $\Pi_M$ must not contain *unbounded loops, any direct or indirect recursion* or *dynamic function references* [9]. Ultimately, then, the program path model $\Pi_M$ must contain user provided runtime information of path execution by providing execution numbers or by providing information of path relation.

Considering the huge amount of different processor implementation $\Lambda_I$, the architectural modeling process is split into two stages: *abstraction* and *processor modeling*. In the first, a real processor $\Lambda_I$ with its several characteristics is drawn as a simplified structural and functional model $\Lambda_M$ with simplified characteristics. In the second stage, each characteristic of model $\Lambda_M$ is assigned to one or more algorithms $\Lambda_{C1,2,...}$ of a meta model $\Lambda_{MM}$ for the description of the behavior. Then, these algorithms can be applied to the analysis blocks mentioned above. It is worth stating at this point that algorithms and analysis blocks are interdependent.

From the above follows, that the analysis composition falls into two stages, too. In the first, the timing for the analysis blocks ($\Pi_{Mi}$) are computed according to the appropriate meta model algorithms ($\Lambda_{Ci}$). In the second, the worst-case execution time is the sum of execution times of all analysis blocks along the worst-case execution path.

This brings us to the question of the worst-case path. Here, for the first time, PARK [8] showed the necessity of consideration of path relations. However, this would lead to exponential path enumeration. Nowadays, it is well accepted that the implicit path enumeration approach [4] is the best way to bound the possible paths to the feasible. Implicit path enumeration takes the timing analysis as an optimization task, say maximize the sum-of-basic-block-timings.

### 2.2. The analysis environment

The methodology described above is implemented in a timing analysis tool called GROMIT. Thereby, the analysis composition is based on the implicit path enumeration approach [4]. The basic structure of GROMIT (Figure 2) can be seen as follows: Its inputs are a assembly program to be analyzed (*.s), a description of the functional path relation of that program (*.f) and a description of processor properties complying to the simplified processor model $\Lambda_M$ (*.ppc). Its output is a *integer linear programming (ILP)* description consisting of a cost function to be maximized (derived from the first stage of analysis composition) and structural (describing the behavior of architectural features $\Lambda_{Ci}$) and functional (describing path execution number and path relations) constraints. This can be given to an *ILP*-solver (e.g. lp_solve) in order to compute the worst-case execution time (wcet).

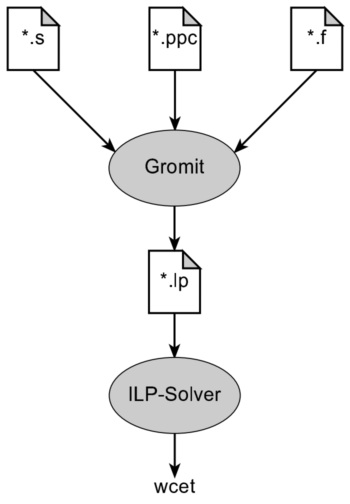

**Figure 2. The structure of the analysis tool**

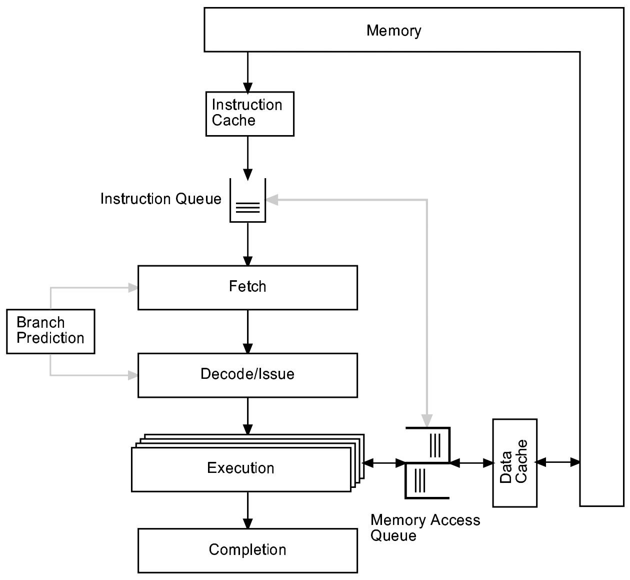

**Figure 3. The processor model $\Lambda_M$ of** GROMIT

The analysis system GROMIT is written in *Java* and is therefore portable to different platforms.

Figure 3 shows the processor model $\Lambda_M$ used to describe the several properties of the *PowerPC* processor family. It generally consists of a four-stage pipeline with the assigned functional stages *Fetch, Decode/Issue, Execution, Completion*. Added to that, there are modules like *Caches*, *Branch Prediction* or *Memory*.

The *fetch stage* is directly coupled to the memory hierarchy consisting of an internal instruction cache and an external, asynchronous memory of either type DRAM, SRAM or ROM. Due to different access times of cache and memory fetch latency times will also differ. Thus, we assume cache accesses takes one cycle. However, memory accesses have to be modeled as burst accesses controlled by different parameters. The *execution stage* can have multiple execution threads. Thereby, the instructions are issued and completed in order. Load/store instructions can access the data cache or main memory. Therefore, they can have a deep impact on the fetch stage. The *issue stage* and the *completion stage* have no further analysis semantics except for time consumption. *Branch prediction* is no matter of consideration yet, but will be an issue for future work.

## 2.3. The validation environment

An essential aspect of our work is the validation of the analysis results in order to draw conclusions from it. We set processors of the *PowerPC* family in the center of our experiments due to several reasons. Above all, they all comply to the *PowerPC* specification - the *User Instruction Set Architecture (UISA)*, the *Virtual Environment Architecture (VEA)* and/or the *Operating Environment Architecture (OEA)*, more or less. Sure, this eases the portability of programs. But, first of all, it offers a good opportunity to study effects of different architectural properties with respect to the actual or predicted execution timing. Hence, we have chosen two different *PowerPC* implementations - the processor PPC403 and the processor MPC750.

The embedded microcontroller PPC403 is fully compliant with specifications for 32-bit implementations of the *UISA*. It provides a fixed point CPU with single cycle execution for most instructions as well as instruction and data caches. This processor is used in an emulation environment called SPYDER[12] and runs with the real-time operating system *VxWorks/Tornado*. The execution timing was measured by triggering and grabbing external signals through a logic analyzer.

The MPC750 is a 32-bit implementation of the *PowerPC* architecture, as well. It provides five execution units in parallel, whereas only two instruction are issued together (see section 4). All instructions are issued and completed in the order of the fetched instruction stream. In addition, there is a branch processing unit which has not further been investigated, yet. The memory hierarchy supplies separate instruction and data cache units and a 64-bit access to memory. In our case, we do not have a look at either second-level caches nor memory management mechanism.

This processor is mounted on a *slot-0* board (MCP750) integrated in a CompactPCI system (CPX2108)[7]. There, the processors performance monitor registers provide the ability to monitor and count predefined events such as processor clocks, misses in the instruction cache, types of instructions, mispredicted branches etc. Our application *MakeMonitor* (written in *Java*) makes those features available to the user, and the program to be analyzed is instrumented accordingly. Thus, an easy and accurate determination of the programs runtime is ensured.

## 3. Cache modeling and analysis

Caches may have a great influence on the runtime which is illustrated in Figure 4. Thus, the benchmarks run about three times faster with an instruction cache than without any. One may conclude from that, that the prediction error is, at least, in this order of magnitude if there was no consideration of instruction cache modeling. That's why it is worth investigating the modeling and timing analysis concerning the accuracy and the cost especially since they are interdependent.

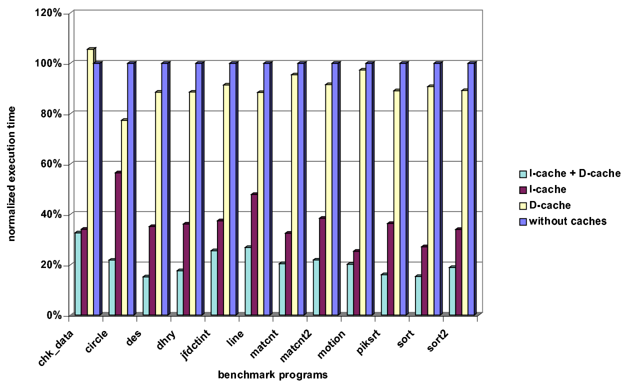

**Figure 4. Measured execution times of benchmark programs on the processor PPC403GA**

### 3.1. Methodology

From the introduction of instruction caches follows that there are different fetch latency times for cache access in case of a cache hit and for memory access in case of a cache miss. Furthermore, the behavior of the instruction cache depends on its structure (cache size, line size, level of associativity) and on its replacement mechanism. This has an implication for program path modeling, architectural modeling and analysis composition, as well. Here, we will give a short overview of the instruction cache modeling derived from [4].

First, the program has been read into an internal control flow graph (CFG) structure (Figure 5a) consisting of basic blocks ($BB_i$) connected by edges representing the possible program paths. Secondly, the basic blocks are subdivided into *line blocks* ($LB_i$). A line block contains all those instructions of a basic block which are mapped to the same cache line. Because each single instruction of a line block has the same or at least similar properties with respect to the fetch latency times and therefore, line blocks are suited for basic analysis.

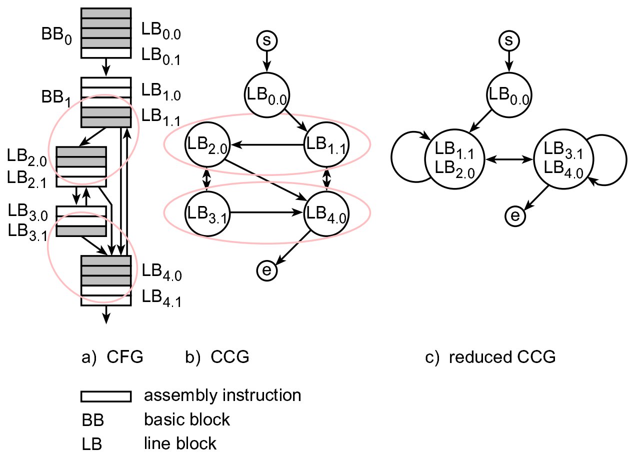

**Figure 5. The construction of the cache conflict graphs**

The architectural model describes two issues: Which line blocks compete for the same cache line and how does the cache replace competing line blocks? The first question is modeled by a so-called cache conflict graph (CCG, Figure 5b), the second by a cache state transition graph (CSTG). Cache conflict graphs depict the control flow through the competing line blocks. Figure 5a,b serves as an example: the line blocks with the same color (gray, white) are competing for the same cache line (5a) and thus, the cache conflict graph can be constructed (5b). The cache state transition graph (see [4] for details) results from the cache conflict graph as well as from the cache replacement mechanism (e.g. PLRU[1] of the MPC750) and associativity (e.g two-way associativity of the PPC403).

Figure 6 shows some prediction results of benchmark programs[2] running on processors with instruction caches at their different modeling variants. This demonstrates, that cache modeling is absolutely necessary in order to get proper prediction results.

### 3.2. Problems

The methodology of cache modeling and analysis described above raises a few questions with prospect to the computability and applicability.

The number of all possible states of the cache state transition graph depends of the used replacement strategy, the associativity $n$ and the competing line blocks $m$ and

[1]pseudo least recently used

[2]The processor PPC403 does not provide a floating point unit. Here, floating point benchmarks are not considered.

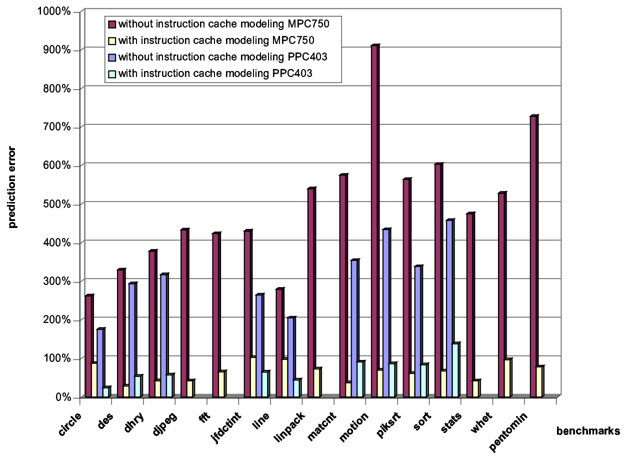

**Figure 6. The prediction results at different processors with instruction caches (PPC403 without floating point benchmarks)**

is given by $N_{LRU} = \sum_{i=0}^{n} \frac{m!}{(m-i)!}$ for LRU[3] and $N_{PLRU} = 2^{n-1} \sum_{i=0}^{n} \frac{m!}{(m-i)!}$ for PLRU replacement strategy. From this it follows that the maximal possible number of states of the cache state transition graph explodes quickly with the increasing number of conflicting line blocks and the cache associativity.

This situation has an implication to the *wcet*-analysis, as well. The more complex the cache model, the more complex the cache constraints needed and the more variables used by the ILP solver.

## 3.3. Solutions

There are several approaches to reduce the number of states of the cache conflict graph and cache state transition graph in order to reduce the complexity.

We think, that an proper reduction can be reached by

- *reducing the number of analysis blocks* in order to reduce the number of nodes in the cache conflict graph (CCG) and cache state transition graph (CSTG)
- *modeling a less complex cache replacement mechanism* in order to reduce the number of nodes in the cache state transition graph (CSTG)
- *modeling a lower level of cache associativity* in order to reduce the number of nodes in the cache state transition graph (CSTG)

[3]least recently used

**Reducing the number of analysis blocks** Line blocks which allocate the same cache line do not compete (Figure 5a). This enables the option of merging those line blocks together to a single node (Figure 5b). The reduced cache conflict graph contains much less nodes as well as edges (Figure 5c). Therefore, reducing the number of nodes in the graph reduces the number of cache constraints in the same order of magnitude. The number of edges which are synonym for variables of the ILP problem are reduced much more. The cache constraints are then going to be extracted as before.

The magnitude of the graph reduction depends on the size of the basic blocks and the size of the cache line. The larger the ratio of the cache line size and the average basic block size, the more likely the merging.

**Modeling a less complex replacement mechanism** As presented above, the different replacement strategies raise different computational complexity. Thus, the modeling of a LRU strategy instead of PLRU simplifies, but does not add a significant error.

**Modeling a lower level of cache associativity** The third approach does not encroach on the modeling methodology as the previous does. Conversely, it reduces complexity by modification of the cache parameter associativity. Hereby, the competing line blocks are constant, rather then the cache size. Consequently, if the magnitude of the associativity is divided by two, then the cache size is divided by two in order to ensure worst case conditions.

For all three techniques hold, that the reduction of computational complexity on the one hand means the reduction of accuracy on the other.

## 3.4. Results

In order to assess the effects of the proposed approaches on the computational complexity and the estimation accuracy, we carried out some experiments on benchmark programs running at the processor PPC403GA. Thereby, each benchmark program refers to a particular ratio of program size to cache size. The PPC403 provides a two-way set-associative instruction cache with a size of 2 KByte. Each of the two sets is organized as 64 lines of 16 bytes each, and the LRU replacement strategy is used.

Figure 7 shows the association of the time needed to compute the instruction cache constraints $t_{cm}$(y-axis), the ratio of program size to cache size (x-axis) and the model used for timing estimation (parameter). Figure 8 presents the same association of the time needed to solve the ILP problem. The different types *A* and *B* mentioned in Figure 7 and Figure 8 refer to the CCG (*A*, Figure 5b) and reduced CCG (*B*, Figure 5c).

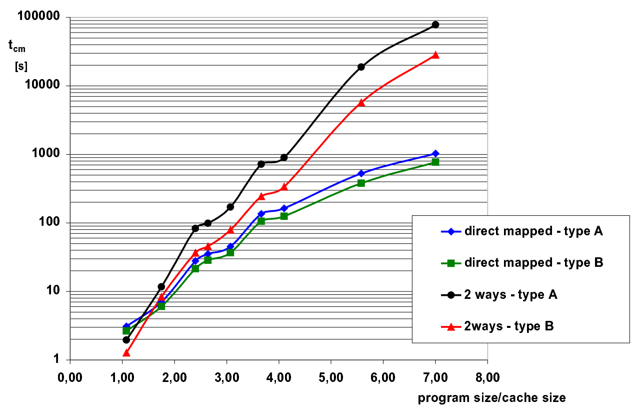

**Figure 7. The time needed to compute the cache constraints (*Sun Ultra2 300MHz, Sun JIT-Compiler*)**

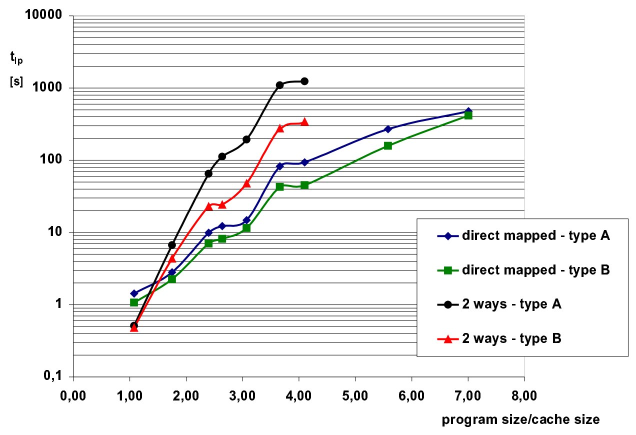

**Figure 8. The time needed to solve the ILP problem (*Sun Ultra2 300MHz, lp_solve Version 2.3*)**

As expected, modeling a two-way set-associative cache is much more expensive than modeling a direct mapped cache. Thus, at a ratio of 5 : 1 program size to cache size, the computation time needed to determine the cache constraints differs by two orders of magnitude (Figure 7). A similar behavior is given in Figure 8. The time needed to solve the ILP problem is about in the same order of magnitude as the modeling time. Unfortunately, some ILP results could not be gathered due to a bug in the program *lp_solve*.

Furthermore, it can be picked out from the Figures 7 and 8, that models using cache state transition graphs (here two-way set-associative caches) are more sensitive to the different analysis block types (*A* or *B*) than such without any. In fact, at those models both types seem to diverge from each other regarding modeling time or ILP time.

Figure 9 shows the additional prediction errors which arose from the discussed modeling variants. It can be seen from the figure that there is a more significant gain on accuracy by using the next level of cache associativity towards the actual two-way set-associativity than by using the more precise model of the cache conflict graph.

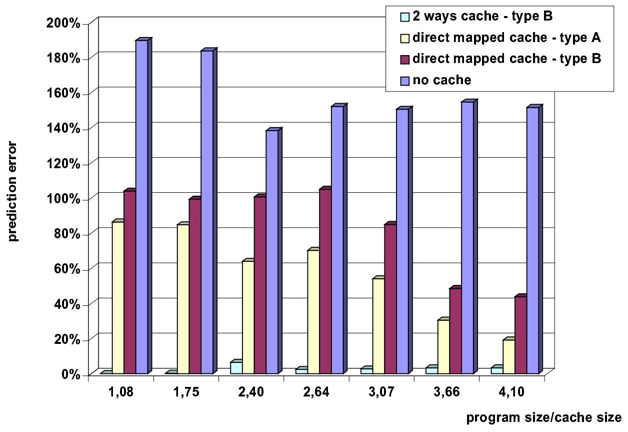

**Figure 9. The prediction errors of the cache models**

To summarize, Figures 7 and 8 show the significant reduction in computation time by using the reduced CCG instead of the CCG. Here, Figure 9 shows a relatively marginal loss of accuracy. The second option for reducing computation time, namely the modeling of a lower level of cache associativity, would result in a more significant reduction in computation time at larger prediction errors.

Basically, the introduced methodology on instruction cache modeling is transferable to the data cache modeling and analysis [4].

## 4. Analysis of multi-issue pipelines

The processor MPC750 provides a super-scalar architecture, whereas the instruction flow through the execution stage is shown in Figure 10. A maximum of two instructions can be dispatched per clock cycle, but only one instruction to each execution unit. The instructions are strongly dispatched as well as completed in the order of the instruction stream. But, different instructions mapped onto different execution units may consume multi-cycle latencies.

### 4.1. Methodology

Here, the dynamic schedule of the instruction sequence of the basic analysis blocks is required in order to determine the execution time. Basic analysis blocks might be basic blocks or line blocks or super-blocks, as well. The longer the instruction sequence, the more accurate the modeling of the actual schedule. In general, the required dynamic schedule has to consider all - dispatch, execution and completion.

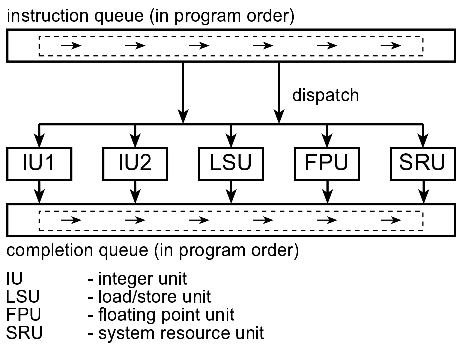
instruction queue (in program order)
dispatch
IU1
IU2
LSU
FPU
SRU
completion queue (in program order)
IU - integer unit
LSU - load/store unit
FPU - floating point unit
SRU - system resource unit

**Figure 10. The model of the processor MPC750**

The analysis input, the assembly program, has been statically scheduled by the compiler before. Thus, the order of the instructions within the analysis blocks is given. In the case of in-order-issue and in-order-completion this order will not be changed by the processor, anyhow. We would refer to it as the *implicitly* given resource constraints since the compiler used machine dependent information like number of registers etc., too. So, data dependencies, which are flow dependencies, anti dependencies and output dependencies, can be derived from the analysis blocks. Then, those data dependencies and the sequential order of instructions are the basis to specify a priority of each single instruction. If there were no data dependencies between adjacent instructions they get the same priority. If there were data dependencies they get different priority numbers in ascending order.

The processor model provides several execution units. Thereby, instructions can be mapped to one or more units. That's why we want to introduce the concepts of *assignment*, *binding* and *allocation*. Every instruction $i$ is attached to one instance $\gamma(i)$ of a resource type $r_k$. The maximal number of instances of $r_k$ is given by $\alpha(r_k)$. The assignment of instruction type onto a resource type is given by the so-called resource graph $G_R$. We want to refer to it as the *explicitly* given resource constraints.

The scheduling process has to find an appropriate binding with respect to the explicit given resource constraints. Therefore, we used a constraint driven list scheduling algorithm (Algorithm 1) [11].

The nominee set $K_{t,k}$ of $r_k$ consists of those instructions whose predecessors are finished at the time $t$. The set $G_{t,k}$ of $r_k$ consists of those instructions executed in $t$. Finally, through the function $p$ a set $S_t$ of maximal priority can be chosen from $K_{t,k}$ in order to get scheduled. The priority of the instructions is set by the implicitly given resource constraints, explained above. With all that, different schedules $\tau$ for different pipeline states (with/without caches) can be determined.

```
LIST($AnalysisBlock, G_R(V_R, E_R), \alpha, p$) {
   t:=0;
   REPEAT {
      FOR (k=1) to $|V_R|$ {
         Determine $K_{t,k}$;
         Determine $G_{t,k}$;
         Choose set of maximal priority
            $|S_t| \subseteq K_{t,k} : |S_t| + |G_{t,k}| \leq \alpha(r_k)$;
         FOREACH ($v_i \in S_t$) $\tau(v_i) := t$;
      }
      $t := t + 1$;
   }
   UNTIL (all nodes $v_i$ are scheduled);
   RETURN ($\tau$);
}
```

**Algorithm 1: The list scheduling algorithm**

## 4.2. Results

Figure 11 shows the gain of precision received by modeling of multi-issue pipelines in two cases. The first covers the case that the processor MPC750 and its model contain no caches. In the second, the processor and its model run with an instruction cache. The gain of prediction is much higher in case one than in case two. This can be explained by their different analysis blocks. In the first case basic blocks and in the second cache line blocks are used. In general, basic blocks contain more instructions which can be scheduled at the same time than cache line blocks.

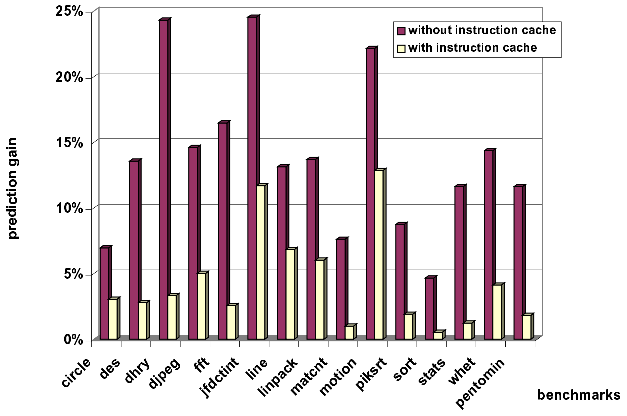

**Figure 11. The gain of precision received by multi-issue modeling at the MPC750**

Although, the gain received by modeling of multi-issue pipelines is strongly sensitive to the distribution of instructions within the analysis block, Figure 11 shows also the dependencies of the number of branches (`jfdctint` - 1%, `matcnt` - 20% of all instructions) and the possible prediction gain.

The impact of modeling multi-issue pipelines is much smaller than the impact of modeling the memory behavior regarding the prediction results.

## 5. Conclusions and Future Work

The accuracy of the prediction results and the applicability of the underlaying algorithms are influenced by several parameters. In addition, both issues are interdependent. Thus, the first part of this paper examined this connection regarding memory and cache modeling as well as analysis. In addition, here, we learned by experience about the difficulties of modeling more complex bus interfaces (MPC750). The second part introduced a methodology to predict the timing behavior for multi-issue machines.

Modern processor architecture are designed to increase the utilization of the parallelism which is inherent to programs. Beside pipelining, data and instruction caching or multi-issue processing, speculative branch processing is an advanced technique (e.g. provided by the MPC750) which is strongly coupled to the behavior of the memory hierarchy. This will be examined in future work.

## Acknowledgments

This work is supported in part with funds of the *Deutsche Forschungsgemeinschaft (DFG)* under reference number 322 1040 within the priority program “Design and design methodology of embedded systems”.

The results have been partialy obtained within the diploma thesis of Axel Siebenborn [10] supervised by the authors.

## References

[1] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proceedings of the ICCAD-97*, pages 598–604, Nov. 1997.

[2] C. Ferdinand and R. Wilhelm. Fast and Efficient Cache Behavior Prediction. 10-page-handout, Universität des Saarlandes, Sept. 1997.

[3] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. In *Proceedings of the 11th ACM International Conference on Supercomputing*, July 1997.

[4] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 254–263. IEEE, December 1996.

[5] S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 97–108, December 1998.

[6] S. Lim, B. Rhee, H. Shin, Y. Bae, S. Min, K. Park, and G. Jang. An Accurate Worst Case Timing Analysis Technique for RISC Processors. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 97–108, December 1995.

[7] Motorola Computer Group, Motorola Inc. *User Manual’s CPX2108 CompactPCI System*. http://www.mcg.mot.com/, 1999.

[8] C. Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, Seattle 98195, Aug. 1992.

[9] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 8(2):160–176, Sept. 1989.

[10] A. Siebenborn. *Cache-Modellierung für die statische Laufzeitanalyse von Software auf eingebetteten Mikrocontrollern*. Diploma Thesis, Universität Karlsruhe, 1999.

[11] J. Teich. *Digitale Hardware/Software-Systeme*. Springer-Verlag, Berlin, Heidelberg, New York, 1997.

[12] K. Weiss, T. Steckstor, C. Oetker, and R. Kistner. *User Manual SPYDER-CORE-P1*. http://www.fzi.de/sim/spyder.html, 1998.