A Time-Predictable Execution Mode for Superscalar Pipelines with Instruction Prescheduling

Christine Rochange[b] and Pascal Sainrat[a,b]
[a] HiPEAC Network
[b] Institut de Recherche en Informatique de Toulouse
118, route de Narbonne
31 062 Toulouse cedex 4, France
(33) 561 55 63 60
{rochange, sainrat}@irit.fr

ABSTRACT
The time predictability of the components of a real-time system is required whenever it must be guaranteed that deadlines will be met. Research on techniques to evaluate the Worst-Case Execution Time (WCET) of programs has received much attention these last years but current high-performance processors prove to be hard to model both safely and tightly. We acknowledge the difficulty of taking into account more and more dynamic mechanisms within static analysis and this motivates our approach that consists in making the processor fit WCET estimation techniques. We focus on out-of-order superscalar pipelines and we propose to regulate the instruction flow so that subsequent basic blocks execute independently one of each other. This would allow any WCET estimation tool to limit the measurement to individual basic blocks.

Categories and Subject Descriptors
C.1 [**Computer Systems Organization**]: Processor Architectures; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems: Real-Time and Embedded Systems

General Terms
Measurement, Performance, Design.

Keywords
Processor architecture, Pipeline, Real-time, WCET.

1. INTRODUCTION
As part of the design of a hard real-time system, it is necessary to be able to guarantee that deadlines will always be met. This means that the temporal behaviour of every component of the system should be predictable.

While hardware simplicity is often presented as a rule to design safe real-time systems, our purpose is to reconcile high-performance with time predictability. In this paper, we focus on the processor architecture and we ignore peripheral components (cache memories, TLBs) as well as external events (interrupts) and interactions with the operating system (process scheduling, virtual memory, etc.).

The Worst-Case Execution Time (WCET) of a program to be run on a given processor is the maximum execution time of the task whatever the input data is. It is often difficult to derive the real value of the WCET and generally an upper bound of the WCET is estimated. To be useful, this bound should be both tight and safe.

The time predictability of a processor relies on the availability of techniques that can model its behaviour to calculate an upper bound of the WCET. The complexity of these techniques must be taken into account and a processor might be declared unpredictable if computation and/or memory requirements to analyse the WCET are prohibitive. Also the accuracy of the WCET bound can moderate the notion of time predictability: if the analysability requires making very pessimistic assumptions about the hardware (e.g. all of the branches are mispredicted), then the system cannot be considered as predictable. In the rest of the paper, we say that a processor is predictable if it can be analysed at a reasonable cost and if the estimated WCET is not too far from the real WCET.

These last 15 years, tremendous work has been done to develop analysis methods to estimate the WCET for high-performance processors. Target features include instruction and data caches [19] [1] [23] [13], simple in-order pipelines [14], simple branch predictors [6]. Some solutions have been proposed to analyse more complex hardware (superscalar and dynamically-scheduled pipelines [12], [18], advanced branch predictors [22]) but their cost (in terms of computation time and/or memory footprint) leave unclear their contribution to the time predictability of advanced architectures.

As said before, hardware simplicity is often presented as the key requirement for time predictability. It is argued that the choice of a processor should be restricted to scalar pipelines with nice properties [9], that cache memories should be locked [20] or replaced by scratchpad memories [3], that branch prediction should be static or, at worst, limited to one-level schemes [10]. However, some real-time systems have high computation requirements and the use of high-performance processors might be unavoidable. This is why we propose to fit out advanced processors with the possibility of executing in a real-time oriented mode so that they can be accurately modelled by state-of-the-art WCET analysis tools while keeping most of their performance.

In this paper, we focus on processors featuring dynamically-scheduled pipelines which have been shown to lack in predictability with respect to WCET static analysis [9]. Section 2 gives an overview of WCET calculation techniques. Section 3 discusses the time unpredictability of dynamically-scheduled pipelines and shows how they could be adapted to WCET analysis methods. Section 4 describes an implementation of the proposed approach that consists in regulating the instruction flow to make the executions of subsequent basic blocks completely independent one of each other. Simulation results that show the relevance of this scheme are given in section 5. Section 6 discusses related work. We draw conclusions in section 7. The appendix at the end of the paper describes the evaluation framework used to produce all the results given in the paper.

2. WCET ANALYSIS: STATE OF THE ART

Dynamic WCET evaluation consists in measuring the execution time of the program either on the real hardware or on a simulator. To obtain a safe upper bound of the WCET, it is necessary either to determine an input that guides the execution along the longest path (but, in the general case, the length of the different execution paths cannot be statically predicted and thus the corresponding input cannot be determined) or to measure every possible path, which requires to define a set of inputs that cover all the possible paths (which might not be easier). If measurements are done using a simulator, the search of those inputs can be avoided by using the symbolic execution technique [18] that initializes any input data with the unknown value, propagates this value through the program execution and, whenever a conditional branch with an unknown condition is encountered, explores both branches. In all cases, if the number of possible paths is large, the measurement time is prohibitive.

The main goal of static WCET analysis is to limit measurements to small pieces of code (instead of full paths). Often, the granularity is that of a basic block (i.e. a piece of code which is always entered by its first instruction and exited by its last instruction) or of short sequences of basic blocks. Splitting each path into basic blocks allows a limitation of the measurement time because one block potentially belongs to several paths (but is only measured once).

In broad outline, static WCET analysis is done in three stages [8]. The flow analysis determines all the possible paths and represents them with a control flow graph and/or a syntax tree annotated with loop bounds, infeasible paths, etc. These annotations can be provided by the user or, in some cases, derived automatically. The timing analysis finds out the execution times of basic blocks. This generally requires two steps. First, history-dependent mechanisms (cache memories and the branch predictor) are analysed. This first step, referred to as global timing analysis, has to consider full execution paths and often uses algorithms derived from compiler techniques. In a second step, known as local analysis, the execution times of basic blocks are estimated (often using a cycle-level simulator) taking into account the results of the global step. Then, the computation of the WCET consists in adding individual execution times of basic blocks along the syntax tree [25] [16] or the control flow graph [11]. The IPET method [15] expresses the search of an upper bound of the WCET as an optimization problem where the execution time of the full program is to be maximized under some constraints derived from the control flow graph.

In the next section, we will explain why static analysis methods are not adequate to model current advanced processor architectures).

3. LONG TIMING EFFETS IN PIPELINES

In a pipelined processor, there are some temporal interactions between basic blocks that should be taken into account if the upper bound of the WCET has to be tight. Interactions between two successive basic blocks are generally easy to model. For example, if we consider the IPET calculation technique, the edge that links these two blocks in the control flow graph can be assigned a negative execution time that represents the gain due to the overlap of the blocks in the pipeline. However, interactions between distant blocks are more difficult both to identify and to model for WCET estimation.

Before explaining how the hardware could be adapted to prevent this kind of undesirable interactions from occurring, we give an overview of previous work by Jakob Engblom [9] who revealed the possible existence of such effects.

3.1 Engblom's Timing Model

Engblom has proposed a general temporal model that relates the execution times of blocks and those of sequences of blocks. This model is expressed by the equations and diagram given in Figure 1.

$$t_{B_1 \ldots B_n} = \sum_{j=1}^{n} t_{B_j} + \sum_{1 \le i < k \le n} \delta_{N_i \ldots B_k}$$

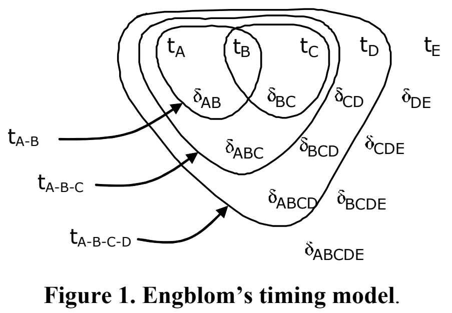

Figure 1. Engblom's timing model.

The execution time of the sequence of blocks $B_1 \ldots Bn$ is the sum of the execution times of the individual basic blocks and of some terms, called long timing effects, that represent the impact of the execution of one block B_i onto the execution of the following sequence $B_{i+1} \ldots B_k$. Engblom has shown that a long timing effect can only occur if Ni stalls the execution of a later instruction belonging to the sequence $B_{i+1}…B_k$ or if B_i is finally parallel to $B_{i+1}…B_k$, that is if B_i retires after the sequence. Since our processor model assumes that instructions complete in the program order, we will not consider the second possibility and we will only retain instruction stalling as a source of long timing effects.

Engblom has analysed many examples and has found that long timing effects can be unbounded, i.e. that they can occur for block sequences of any length (potentially full execution paths). This means that identifying all the possible long timing effects might require examining all the possible complete execution paths, which obviously goes against the basic principle of static WCET analysis.

Unfortunately, Engblom has shown that long timing effects can either be positive, negative or null. Ignoring the negative effects can make the estimated WCET longer than the real WCET. Over-estimating the WCET is generally undesirable, first because it can lead to budget components too much powerful compared to the real requirements (and then these components will be under-used) and secondly because it can lead to the wrong conclusion that the system cannot be scheduled to meet the deadlines. However, positive long timing effects are far more dangerous because ignoring them during the WCET analysis can engender WCET under-estimation which is not acceptable in a hard real-time context where human lives might be concerned. Thus, positive long timing effects must be prohibited if the estimated WCET has to be safe.

3.2 Measuring Long Timing Effects

Engblom has measured the frequency of long timing effects in some benchmark codes, considering a NEC V850 processor. He has recorded many long timing effects that were all negative (while hand-built examples show that positive effects are possible). We extend his work by making some measurements for an out-of-order superscalar processor.

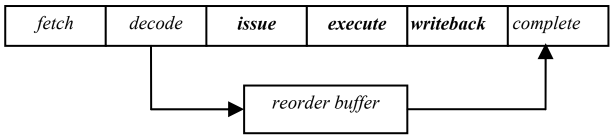

Figure 2. Baseline processor model.

We used a cycle-level simulator for a 6-stage superscalar out of-order processor as the one shown in Figure 2. More details on our simulation model and on the simulated benchmarks are given in the Appendix at the end of the paper. Our simulator is able to extract all the possible sequences of basic blocks shorter than a given length from a CFG and to simulate all these sequences (the processor state is reinitialized before simulating a new sequence).

Table 1 gives the distribution of the values of the positive effects we measured among all possible sequences of less than 10 basic blocks for a 4-way superscalar processor. It appears that some benchmarks, like `lms`, exhibit a large number of positive long timing effects, with values up to +6 (recall that in the absence of balancing negative effects, ignoring an effect of value +6 leads to underestimating the WCET by 6 cycles).

Table 1. Values of positive long timing effects in 10-basic-block sequences

LTE value / *program*	1	2	3	4	5	6
`crc`	24	17	14	10	2	0
`jfdctint`	7	7	1	0	0	0
`ludcmp`	20	19	2	3	1	0
`fibcall`	0	1	0	0	0	0
`matmul`	34	25	3	0	0	0
`fft1`	111	120	24	7	12	0
`lms`	7	91	11	5	1	1

Table 2 shows the distributions of the lengths of the sequences that generate long timing effects. Most of the benchmarks exhibit timing effects over long block sequences, while it should be noted that sequences longer than 11 blocks (not simulated in this experiment) might also generate long timing effects. This means that, if we wanted to catch up all the possible timing effects, we should simulate all the possible sequences of any length. This would obviously be much more costly than limiting measurements to individual basic blocks and two-block sequences (thus ignoring the possible long timing effects). For example, we measured that the `lms` benchmark contains 6,399 possible sequences of 11 basic blocks or less, which represents 501,500 instructions to simulate, while measuring only sequences of one or two basic blocks limits the total number of instructions to simulate to 2,761.

Table 2. Lengths of sequences generating long timing effects

LTE length / *program*	3	4	5	6	7	8	9	10	11
`crc`	0	26	26	0	0	12	3	0	0
`jfdctint`	0	4	2	4	1	1	1	2	0
`ludcmp`	0	15	15	5	2	4	3	1	0
`fibcall`	0	1	0	0	0	0	0	0	0
`matmul`	0	8	9	13	4	8	10	7	3
`fft1`	1	86	73	19	16	30	34	6	9
`lms`	3	31	19	3	18	12	0	18	12

3.3 Illustrative Example

To illustrate the potential WCET under-estimate due to long timing effects, let us consider the sample code shown in Figure 3. We compiled this code with the `gcc` cross-compiler targeting the PowerPC 750 ISA, with all optimizations disabled (which is generally the case in real-time systems). The control flow graph that we extracted from the object code is given in Figure 4. As in the previous section, we measured all shorter-than-10-block sequences and Table 3 gives the positive long timing effects that were observed.

As said before, ignoring positive effects could lead to WCET underestimation if they are not completely balanced by negative effects. We verified this statement for our sample code by making the following experiment. First we simulated all the possible execution paths (our simulator implements non-optimized symbolic execution [18]) and we found that the real WCET was equal to 157 cycles. Then we computed the WCET using the

IPET technique (we wrote manually the set of flow and structural constraints, and then used the lp_solve tool [4]) and we found it equal to 155 cycles, which is an unsafe estimate, even if the error is small here. The error comes from ignored long timing effects and, in the general case, it cannot be bounded.

```
#define N 4

int main()
{
   int i, tmp, max, nb=0;
   char done=false;

   while ( !done && (nb < N-1))
   {
      done = true;
         for (i=0 ; i<N-1 ; i++)
            if (array[i] < array[i+1])
            {
               done = false;
               tmp = array[i];
               array[i]=array[i+1];
               array[i+1] = tmp;
            }
         nb++;
     }
}
```

Figure 3. Sample source code

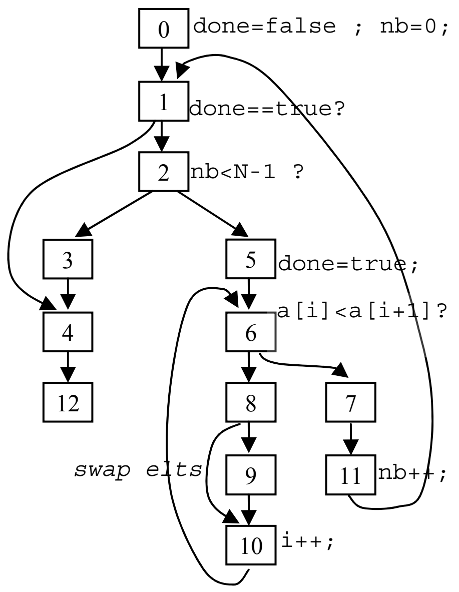

Figure 4. Control Flow Graph of the sample code

A detailed study of the sources of the long timing effects observed in the example can be found in [26]: execution traces are analysed and the way some basic blocks impact the execution time of subsequent and distant blocks is highlighted. Many hardware features are likely to generate long timing effects: superscalar pipelines, long latency functional units, limited capacity of all kind of resources (functional units, reorder buffer, checkpoint structures for recovering from branch misprediction, in-order completion, etc). Since high-performance processors include most of these features, and acknowledging the fact that long timing effects are difficult to take into account during static WCET analysis, it seems desirable to find a way to eliminate these effects as exposed in the next section.

Table 3. Positive LTEs in our sample code

sequence of blocks	LTE	sequence of blocks	LTE
1-2-3-4	3	2-5-6-7-11	3
1-2-5-6-7	1	2-5-6-7-11-1-2-5-6	1
1-2-5-6-8	3	2-5-6-7-11-1-4-12	2
10-6-7-11	2	6-7-11-1	1
10-6-7-11-1-2-5-6	1	6-7-11-1-2-5-6-7	1
10-6-7-11-1-4-12	2	6-7-11-1-2-5-6-8	1
11-1-2-5-6	1	8-10-6-7	1
11-1-4-12	2	8-10-6-8	4
2-3-4-12	3		

4. TOWARDS REAL-TIME ORIENTED EXECUTION

4.1 General Principle

The baseline idea of our approach to make hardware predictable is that the processor should include some additional logic to dynamically decide when a new basic block can enter the pipeline without running the risk of being stalled by a previous instruction or of itself stalling a later instruction (an instruction stall can be caused by data or resource dependencies). If the fetch of basic blocks can be regulated this way, every block is executed as if it was alone in the pipeline and then no long timing effect can arise. Such a mechanism can be compared to *pipeline gating* [21], a technique proposed to reduce energy consumption, that consists in controlling the speculative execution to prevent from doing too much work on the wrong path.

4.2 Instruction Flow Regulation Policy

Our regulation strategy determines the earliest time when a block can enter in the pipeline without the risk of any inter-block temporal effect. We suggest an implementation of such a policy based on the following data:

- the earliest time at which resource might be needed by any instruction entering the pipeline at the next cycle. All kinds of resources are considered: pipeline stages, functional units, operand registers, etc.
- the worst-case availability times of these resources.

Fetching a new basic block should be prevented until those data match. The feasibility of calculating this information mostly depends on the instruction scheduling strategy.

In a conventional out-of-order processor with a deep instruction window, the pipeline might contain a large number of instructions that still have not been issued at the time a new basic block is about to be fetched. Determining the availability times of the resources would require to anticipate the way all these instructions will be scheduled (issue, execution and completion times), which means computing in a single cycle what the issue mechanism computes in several cycles (it only schedules a few instructions at each cycle). Moreover, due to the dynamic instruction scheduling policy, any instruction entering the pipeline might undermine the anticipated scheduling and then the availability times should be re-computed at each cycle. This would obviously be far more costly than the conventional issue process (waking up ready instructions and selecting the ones to be executed by functional units at the next cycle) which is already very complex [24].

Several mechanisms have been proposed these last few years to simplify the issuing scheme. Among them, the instruction prescheduling strategy [5] computes the issue time of decoded instructions on the basis of the latencies and of the expected issue times of the instructions they depend on (instructions that cannot be prescheduled because the latency of one of their producer instructions is still unknown are kept in a wait buffer). Instructions are stored into a prescheduling buffer in the order of their anticipated issue times, and the issue process then only consists in selecting instructions from the head of the prescheduling buffer.

The instruction prescheduling strategy completely fits our requirements for making pipelines predictable since it makes it possible to maintain the resource availability times needed to implement the aggressive instruction flow regulation policy.

4.3 Implementing instruction flow regulation for time-predictability

Instruction flow regulation can be implemented in a processor with dynamic instruction prescheduling. Fetching a new basic block is gated until all the instructions in the pipeline have been prescheduled and the availability times of all resources match the earliest requirements of any future instruction. This way, an instruction can never be stalled by an instruction belonging to a previous basic block. For our pipeline model, the earliest requirements for any future instruction that would be fetched at cycle c are the following: it would like to be prescheduled at cycle c+1 and to be issued at cycle c+2. If the shortest functional unit latency is one cycle, the instruction might execute during cycle c+3, its result might be written back at cycle c+4 and the instruction might complete at cycle c+5.

The regulation mechanism has to make sure that the different resources (pipeline stages, functional units and physical register values) will be available at the time they might be required. Since we consider a superscalar processor, the term “available” should be understood as “empty” for pipeline stages. Before considering another basic block for admission into the pipeline, the regulation scheme has to wait until all instructions present in the pipeline have been prescheduled and until all the memory operations have been disambiguated.

Note that this scheme requires that the hardware be able to detect the end of basic blocks. Most of the basic blocks end with a branch instruction which can easily be identified. The last instruction of the other ones could be marked with an unused bit in the instruction code, if any. Otherwise, the compiler could be forced to terminate each basic block with a branch (targeting the next instruction whenever no branch is semantically required). Note that the number of branches to add would be very limited since most of the blocks already end with a branch. Also, the added branches would be unconditional and direct branches, which means that they would be very easy to predict. In all cases, the end of a basic block can only be detected at the decode stage and thus the instruction flow regulation actually takes place at this point. Note that this does not have any impact on the temporal independency of block execution.

While we intuitively feel that our approach guarantees the absence of any long timing effect (because the schedule of a basic block in the pipeline cannot be distorted by other blocks), we are currently working on formally proving it. Even if it cannot be considered as a proof, we report that we have made some measurement on all our benchmark codes considering up-to-11-block sequences and have not found any long timing effect.

5. PERFORMANCE COST OF INSTRUCTION FLOW REGULATION IN THE AVERAGE CASE

All the results given in this section were obtained under the conditions described in the Appendix. Figure 5 shows the cost of dynamic instruction prescheduling. The average slowdown (against conventional out-of-order scheduling) ranges from 5% (2-way superscalar pipeline) to 10.1% (8-way). These figures are close to those reported in [5].

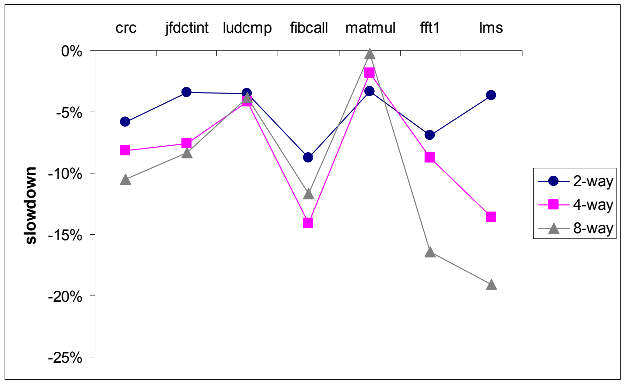

Figure 5. Slowdown due to prescheduling

Figure 6 gives the slowdown observed when the instruction flow regulation mechanism is enabled (compared to prescheduled out-of-order scheduling without any regulation). On a mean over all the benchmarks, the performance loss ranges from 19.1% (2-way) to 37.1% (8-way). The slowdown is due to the fact that the admission of any basic block in the pipeline is delayed because the resource availability times do not match potential earliest resource requirements (at least, there are some unprescheduled instructions in the fetch and decode stages). The mean slowdown against a conventional out-of-order processor ranges from 21.9% (2-way) to 45.4% (8-way).

While the performance slowdown may seem important, one should not forget that usual static WCET analysis is only safe for very simple, scalar and in-order scheduled pipelines. A superscalar and dynamically-scheduled processor that includes our regulation mechanism can execute in a safe mode compatible with WCET static analysis, and largely outperfoms a scalar processor, as shown in Figure 7. The average speedup reaches 130% for an 8-way pipeline.

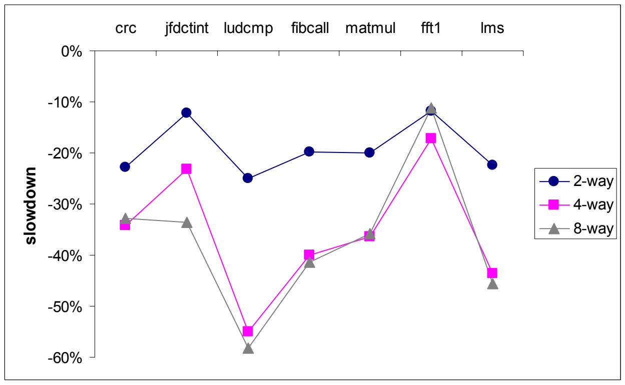

Figure 6. Slowdown due to instruction flow regulation

Note that it is probably not provable that a program would execute faster on a pure out-of-order processor than on a prescheduled and regulated one on every possible execution path. This is why the average-case results should not be understood as an argument for using processors with standard out-of-order scheduling on the pretext that it performs better than the scheme described in this paper.

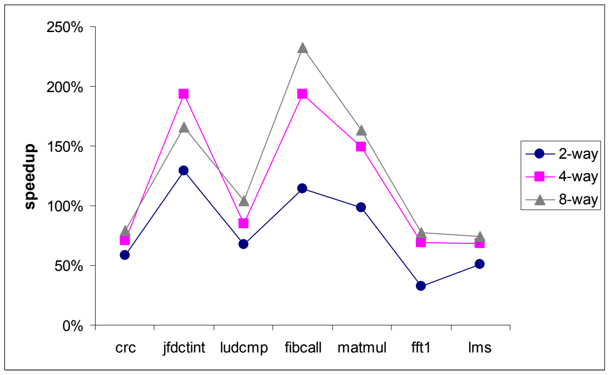

Figure 7. Speedup vs. a scalar in-order pipeline

6. RELATED WORK

The tool described in [12] can handle some superscalar processors with out-of-order execution. However, since the algorithm implemented to estimate the WCET has to consider the effects of all possible interleavings of instructions, the authors note that these interleavings should not be too numerous so that the complexity is kept reasonable. Today and future high-performance processors, with wide pipelines and aggressive out-of-order instruction scheduling might not be tractable. We feel that the scheme we have presented in this paper could largely reduce the number of possible instruction interleavings and then could greatly help in the applicability of this analysis tool.

A recent paper proposes a model to compute the execution time of a basic block in a superscalar, dynamically-scheduled pipeline [17]. This model represents the execution of a basic block by a dependence graph and determines its timing schedule according to the state of the pipeline at the beginning of the block, possible cache miss penalties, etc. It computes the earliest and latest finish times for each instruction of each basic block, and iterates over the control flow graph until it reaches a stable state. Reported results were obtained for a very small processor (4-entry instruction buffer, 8-entry reorder-buffer) which makes it difficult to appreciate the complexity of the model computation. Moreover, while this model seems to take into account most of inter-block effects, its ability to capture any possible long timing effect has not been proved. In Engblom's thesis [9], some examples show that it is possible that none of the sub-sequences of the sequence $B_1 \ldots B_n$ generates a long timing effect, while a LTE is found for the sequence $B_1 \ldots B_{n+1}$ (i.e. block B_1 has no impact on the execution of any block B_i, $_{1<i<n}$ but impacts the execution time of block B_{n+1}). We feel that the model presented in [17] might miss this kind of effect.

Delvai et al. [7] described the SPEAR processor that supports the single-path programming paradigm. Their objective is to reduce the worst-case execution time, even if the average-case performance is degraded. This approach is clearly innovative and is compatible with reliable WCET analysis: since the program has a single execution path, evaluating the WCET simply consists in measuring this path. However, the single-path paradigm leads to severe performance degradation compared to the normal multiple-path programming model, at least on the average case (the authors claim that the worst-case performance might not be impacted). Then this kind of processor is clearly dedicated to real-time applications and would not fit performance requirements for other applications, and thus we wonder whether it would be viable from an industrial point-of-view.

Anantaraman et al. [2] proposed to add to a high-performance processor a scalar in-order execution mode which has been proved predictable. A static analysis of the WCET is performed considering this safe execution mode and checkpoints are defined in the executable code. The program is executed in the fast mode and, whenever a checkpoint is encountered, it is checked whether the effective execution time until the checkpoint has been lower than estimated by the WCET analysis. If it is not the case, the processor switches to the slow but safe execution mode to terminate the execution so that the total execution time matches the estimation. The scheme we described in this paper could be used as the "slow and safe" execution mode, which would help in getting much tighter estimates of the real WCET.

7. CONCLUSION

Designing safe hard real-time systems requires being able to estimate the worst-case execution time of programs. While much research work has been carried out these last years to propose and improve WCET static analysis techniques so that they can apply to more and more sophisticated architectures, current tools cannot tightly model current off-the-shelf high-performance processors. The main difficulties reside in the highly dynamic features implemented to achieve a high-performance level in the average case.

Since we believe that the use of high-performance in some real-time embedded systems is unavoidable, we feel that solutions to make them predictable should be found. In this paper, we have presented an approach that consists in modifying the pipeline so that it fits usual WCET analysis methods. The proposed scheme assumes that the processor features dynamic instruction prescheduling (a recently proposed scheme that aims to reduce the issuing complexity). Time predictability is then achieved through fetch gating that regulates the instruction flow through the pipeline, so that basic blocks do not interfere.

Simulation results show that this scheme has a cost, with an average slowdown of 35% for a 4-way superscalar processor while it largely outperforms a scalar in-order pipeline (proved safe for WCET static analysis) by 120% on a mean. The performance cost should be balanced against the fact that the scheme makes it possible to compute the Worst-Case Execution Time both easily and safely. As we said before, while some recent studies have proposed models to take into account advanced processor architectures, their requirements in terms of computation time may make them difficult to use in practice. Moreover, their ability to capture all the possible long timing effects has not been proved yet.

Another case for our approach is that the regulation mechanism could easily be disabled which would allow the processor to execute in a high-performance mode without any impact on the average-case performance. This possibility might be important from an economical point of view.

As explained in the paper, the instruction flow regulation requires that the processor implements an instruction prescheduling policy. This is linked to the fact that we want to be sure that a basic block that enters the pipeline will *not* be delayed by previous (or subsequent) instructions. This requirement has a cost since it noticeably reduces the overlapping of basic blocks in the pipeline and limits the opportunities to exploit instruction parallelism. Sometimes, the regulation mechanism delays the fetching of a new basic block while it would not have generated any positive long timing effect and then some part of the performance degradation could be avoided. This motivates our ongoing work on *speculative* instruction flow regulation where the processor fetches basic blocks normally, predicting that no long timing effect will occur, and initiates recovery actions whenever it detects a temporal interaction between distant blocks.

We also plan to address the predictability of advanced features of modern processors (branch prediction and speculative execution, non-perfect memory hierarchy, etc). These mechanisms seriously complicate the evaluation of the WCET because they rely on history information which is difficult to analyse without measuring complete execution paths. As mentioned in sections 1 and 2, some work has been done to try to model cache memories and branch predictors as precisely as possible, but it implies some approximations that might not fit with precise pipeline analysis. Instead of attempting to model things that prove to be hard to be modelled, we intend to design more predictable mechanisms.

8. REFERENCES

[1] M. Alt, C. Ferdinand, F. Martin, R. Wilhelm, “Cache Behaviour Prediction by Abstract Interpretation”, Static Analysis Symposium, 1996.

[2] Anantaraman, K. Seth, K. Patil, E. Rotenberg, F. Mueller, “Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-time Systems”, 30^{th} International Symposium on Computer Architecture, 2003.

[3] O. Avissar, R. Barua, D. Stewart, “Heterogeneous Memory Management for Embedded Systems”, International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2001.

[4] M. Berkelaar, “lp_solve: (Mixed Integer) Linear Programming Problem Solver”, 2003. (`ftp://ftp.es.ele.tue.nl/pub/lp_solve`)

[5] R. Canal, A. Gonzalez, “A Low-Complexity Issue Logic”, International Conference on Supercomputing, 2000.

[6] A. Colin, I. Puaut, “Worst-Case Execution Time Analysis for a Processor with Branch Prediction”, Real-Time Systems, vol. 18, n°2-3, Kluwer, 2000.

[7] M. Delvai, W. Huber, P. Puschner, A. Steininger, “Processor Support for Temporal Predictability — The SPEAR Design Example”, 15th Euromicro Conference on Real-Time Systems, 2003.

[8] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, H. Hansson, “Towards Industry-Strength Worst-Case Execution Time Analysis”, Technical Report ASTEC 99/02, 1999.

[9] J. Engblom, “Processor Pipelines and Static Worst-Case Execution Time Analysis”, PhD thesis, University of Uppsala, 2002.

[10] J. Engblom, “Analysis of the Execution Time Unpredictability Caused by Dynamic Branch Prediction”, 9th Real-Time/Embedded Technology and Application Sympo-sium, 2003.

[11] M. Harmon, T. Baker, D. Whalley, “A Retargetable Technique for Predicting Execution Time”, Real Time Systems Symposium, 1992.

[12] R. Heckmann, M. Langenbach, S. Thesing, R. Wilhelm “The Influence of Processor Architecture on the Design and the Results of WCET tools”, Proceedings of the IEEE, vol.9, n°7, 2003.

[13] S. Kim, S. Min, R. Ha, “Efficient Worst Case Timing Analysis of Data Caching”, International Real-Time Technology and Applications Symposium, 1996.

[14] S.-S. Lim, Y. Bae, G. Jang, B.-D. Rhee, S. Min, C. Park, H. Shin, C. Kim, “An Accurate Worst Case Timing Analysis for RISC Processors”, Real-Time Systems Symposium, 1994.

[15] Y.-T. Li, S. Malik, “Performance Analysis of Embedded Software using Implicit Path Enumeration”, ACM SIGPLAN Notices, vol. 30, n°11, 1995

[16] S.-S. Lim, S. Min, M. Lee, C. Park, H. Shin, C. S. Kim, “An Accurate Instruction Cache Analysis Technique for Real-Time Systems”, Workshop on Architectures for Real-Time Applications, 1994.

[17] X. Li, A. Roychoudhury, T. Mitra, “Modeling Out-Of-Order Processors for Software Timing Analysis”, 25th IEEE Conference on Real-Time Systems, 2004.

[18] T. Lundqvist, P. Stenström, “An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution”, Real-Time Systems, Vol. 17, n°2, 1999.

[19] Y.-T. Li, S. Malik, A. Wolfe, “Performance Estimation of Embedded Software with Instruction Cache Modeling”, International Conference on Computer Aided Design, 1995.

[20] Marti-Campoy, A. Ivars, J. Busquets-Martaix, "Static Use of Locking Caches in Multitask Preemptive Real-Time Systems", Real-Time Embedded Systems Workshop, 2001.

[21] S. Manne, A. Klauser, D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction", 25th International Symposium on Computer Architecture, 1998.

[22] T. Mitra, A. Roychoudhury, "A Framework to Model Branch Prediction for WCET Analysis", 2nd Workshop on WCET Analysis, 2002.

[23] F. Mueller, "Timing Analysis for Instruction Caches", Real-Time Systems, 18(2), Kluwer, 2000.

[24] S. Palacharla, N. P. Jouppi, J. E. Smith, "Complexity-Effective Superscalar Processors", 24th International Symposium on Computer Architecture, 1997.

[25] P. Puschner, C. Koza, "Calculating the Maximum Execution Time of Real-Time Programs", Real-Time Systems, vol. 1, n°2, 1989.

[26] C.Rochange, P. Sainrat, "Dissecting Execution Traces to Understand Long Timing Effects", Technical Report IRIT-2005-6-R, 2005.

9. APPENDIX: EVALUATION METHODOLOGY

9.1 Processor Model and Simulator

The measurements presented in this paper were done using a cycle-level simulator that we developed and that consists of three modules:

- an *instruction-set simulator* able to fetch an executable code into the simulated memory, and to decode and execute instructions (GLISS: www.irit.fr/recherches/ARCHI/MARCH/)
- a *timing simulator*, that models the processor architecture, and was developed on top of SystemC (www.systemc.org). The parameters that we used for the work presented here are given in Table 4.
- a *simulation controller* that controls the execution path and enables three simulation modes:
    - *full-path* simulation: the program is executed from the beginning to the end, with a given input data set.
    - *graph* simulation: the controller extracts the Control Flow Graph from the program binary code and builds the list of all the possible sequences of blocks shorter than a fixed limit. Then, it guides the simulation along these sequences to measure their individual execution times, reinitializing the processor between two sequences. These times can then be analysed to compute inter-block timing effects.
    - *symbolic* simulation: it implements a technique proposed by Lundqvist and Stenström [18] that consists in assuming that the input data is unknown and in propagating this "unknown" value through the computations. Whenever a conditional branch with an "unknown" condition is encountered, the simulation controller first enforces the exploration of one of the possible paths and later guides the execution onto the second possible path. This mode allows simulating all the possible paths in a program without having to determine input data sets that guarantee a total coverage of these paths.

Table 4. Processor Model

Pipeline width	**2-way**	**4-way**	**8-way**
fetch queue size	16	32	64
instruction cache	perfect (100% hit rate)		
branch predictor	2048 2-bit counters		
number of pending branches	4	8	32
re-order buffer size	16	64	256
prescheduling buffer depth (width equals the issue width)	4	4	8
wait buffer size	8	16	32
# of functional units (*latency*)			
integer add (*1 cycle*)	2	4	6
integer mul/div (*6 cycles*)	1	1	2
floating-point add (*3 cycles*)	1	1	2
fp mul/div (*6/15 cycles*)	1	1	1
load/store (*2 cycles*)	1	2	2
data cache	perfect		

9.2 Benchmarks

The benchmarks, listed in Table 5, were taken from the SNU (Singapour National University) suite:

`archi.snu.ac.kr/realtime/benchmark/`

Table 5. Benchmarks

benchmark	**description**
`crc`	Cyclic Redundancy Check
`jfdctint`	JPEG slow-but-accurate integer implementation of the forward DCT
`ludcmp`	LU decomposition
`fibcall`	Fibonacci series
`matmul`	Matrix multiplication
`fft1`	FFT (Fast Fourier Transform) using Cooly-Turkey algorithm
`lms`	LMS adaptive signal enhancement