Avoiding Timing Anomalies using Code Transformations

Albrecht Kadlec, Raimund Kirner, Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Wien, Austria
{*albrecht, raimund, peter*}*@vmars.tuwien.ac.at*

***Abstract*— *Divide-and-conquer* approaches to *worst-case execution-time analysis* (WCET analysis) pose a safety risk when applied to code for complex modern processors: Interferences between the hardware acceleration mechanisms of these processors lead to timing anomalies, i.e., a local timing change causes an either larger or inverse change of the global timing. This phenomenon may result in dangerous WCET underestimation.**

This paper presents intermediate results of our work on strategies for eliminating timing anomalies. These strategies are purely based on the modification of software, i.e., they do not require any changes to hardware. In an effort to eliminate the timing anomalies originating from the processor's out-of-order instruction pipeline, we explored different methods of inserting instructions in the program code that render the dynamic instruction scheduler inoperative. We explain how the proposed strategies remove the timing anomalies caused by the pipeline. In the absence of working solutions for timing analysis for these complex processors, we chose portable metrics from compiler construction to assess the properties of our algorithms.

***Keywords*-timing anomalies; worst case execution time (WCET); worst case execution time (WCET) analysis; hard real time; code transformations; compilers**

I. Introduction

Safety-critical hard real-time systems have to guarantee the timeliness as well as the correctness of their results under allconditions. Hence *worst-case execution time analysis* is employed to extract a *safe upper bound* of the program's WCET. Its quality is determined by its proximity to the real WCET, the *tightness*.

To fight the complexity in the number of possible execution paths, it is common for timing analysis to use a *divide and conquer* strategy, analyzing parts of the *control flow graph* (CFG) separately and combining the results. This strategy directly translates into the *WCET compositionality* requirement [1], which states that it must be possible to combine the results for subcomponents with simple computation recipes without sacrificing too much of the timing analysis' *tightness*.

Timing anomalies threaten the WCET compositionality. For timing analysis, the term *timing anomaly* was coined by Lundqvist and Stenström [2]. They describe timing anomalies as *counterintuitive timing behavior*, where a local timing change causes an either larger or inverse change of the global timing.

Timing anomalies result from the timing interference of functionally independent processor components and can be accommodated for by overapproximation or by exhaustive analysis of the state space [3]. The first leads to reduced tightness and poor resource utilization, while the latter causes problems in analysis complexity. By stressing one or the other, the analysis can trade analysis time for tightness.

Timing anomalies eventually manifest as *delay* or *early-out* of individual instructions within the processor pipeline. This effect is more dramatic for out-of-order pipelines, but has also been shown for in-order pipelines [4]. If this effect leads to rearrangements in the issuing of succeeding instructions, the size of the search space for timing analysis is multiplied.

To disburden the task of the real-time systems designer, who needs *safe*, *tight* and *fast* timing analysis, faster timing analysis or total avoidance of timing anomalies can be achieved in one of the following ways:

- *Simplified hardware* avoids dynamic allocation but moves the resource allocation problem to compile time, making WCET analysis simpler.
- *Hybrid hardware* provides a 'disable switch' for the dynamic optimizations and allows for a pragmatic hardware implementation.
- *Compiler transformations* can transform the program in a way, so that it is not sensitive to the dynamic optimizations. This avoids time-consuming and costly changes to hardware, allowing the use of well established, thoroughly tested and hence trusted hardware.

There is a lot of work on hardware modifications [5], [6], [7], and also a lot of work on thorough analysis of existing hardware [8], [9], which trades analysis time to gain safety and tightness. Our basic research is aimed to evaluate the less-researched third option, especially compiler backend transformations for off-the-shelf hardware. This approach was so far only briefly covered by the original work of Lundqvist and Stenström [2] and by Rochange and Sainrath as an adaptation of a proposed hardware modification [10]. A further motivation is, that the RAD750 [11], a space-hardened implementation of the PowerPC 750, is already used for satellite applications, while the direct descendant, the PowerPC 755 has been shown to expose timing anomalies [12]. Until suitable hardware is widely available, a pure software solution could bridge the gap.

In this paper we explain how rescheduling and insertion of non-functional code can be used to avoid pipeline-based

timing anomalies. We present three compiler transformations that achieve this goal of a pure software solution. We conduct measurements and describe the relative weaknesses and strengths of the three approaches.

Together with implementation details, we describe an interface to our code scheduler that allows the integration of analysis results from separate analyses like e.g. data cache analysis to improve the results.

II. Timing Anomalies

Timing anomalies in the context of WCET analysis were first described by Lundqvist et al. [2]. The general problem with timing anomalies is that they do not permit efficient timing analysis, since the local worst-case is not guaranteed to be part of the global worst-case [13]. This is because timing anomalies represent a kind of discontinuity of the hardware behavior. Following the terminology in [14], we call it *amplification timing anomaly*, if there exist two different initial states s_1 and s_2 with the same functional part but a different non-functional part, such that between s_1 and s_2 the change of the global timing is larger than the change of the local timing. We call it *inversion timing anomaly* if between s_1 and s_2 the change of the global timing has a different sign than the change of the local timing.

Timing anomalies were originally known in the context of timing analysis for composed instruction sequences, which are called *series timing anomalies*. Recently, Kirner et al. have shown that timing anomalies can also occur when combining the timing results of multiple hardware analysis phases, which they call *parallel timing anomalies* [14].

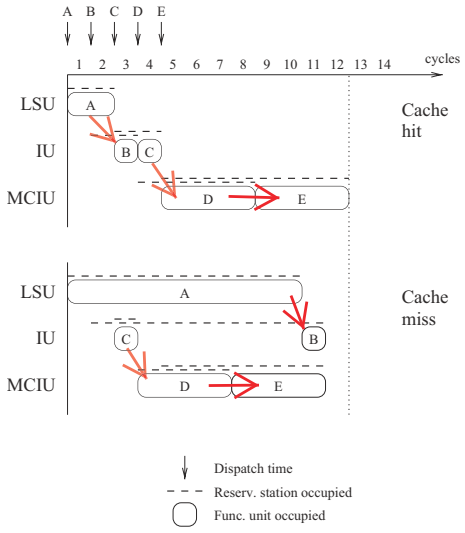
Performing efficient WCET analysis by composing timing results is based on reasoning from the local worst case to the global worst case. As such a reasoning is not valid in case of timing anomalies, we work on code modifications to explicitly avoid timing anomalies.

A. Motivating Example

To illustrate our approach of avoiding timing anomalies arising from dynamic scheduling including out-of-order execution and data caches, we reuse the original timing anomaly example given by Lundqvist, shown in Figure 1(a): Two separate dependence chains are emitted sequentially by the code generator. The data dependences have been added as red arrows.

If instruction A of the first sequence finishes early, instruction B is executed immediately afterwards, delaying the second instruction sequence C-D-E because of resource contention in the integer unit. If, however, A is delayed by a cache miss, B cannot be executed because of the true data dependency and the hardware instruction scheduler schedules C early, which leads to more parallelism (B parallel to E) and thus a shorter overall execution sequence.

This is a nice example of the interaction of dependences with resource restrictions which is the overall cause of scheduling anomalies with greedy scheduling algorithms.



Label	Disp. cycle	Instruction	
A	1	LD	r4, 0(r3)
B	2	ADD	r5, r4, r4
C	3	ADD	r11, r10, r10
D	4	MUL	r12, r11, r11
E	5	MUL	r13, r12, r12

(a) Lundqvist's original example from [2] with dependencies added as arrows

```
A    1    LD   r4, 0(r3)
C    2    ADD  r11,r10,r10
D    3    MUL  r12, r11, r11
E    4    MUL  r13, r12, r12
B    5    ADD  r5,r4,r4
```

(b) The equivalent code after instruction scheduling

Figure 1: Lundqvist's original example revisited.

The timing anomaly can be avoided by rearranging the original instruction sequence: An instruction scheduler that honors the potential latency due to the cache miss will rearrange the code as shown in Figure 1(b). A simple, also greedy compile-time scheduler balancing the dependence-distance of each instruction to the end of the sequence already does the trick for this simple example: The look-ahead that is impossible for the hardware scheduler delays the issuing of B to the end of the sequence. This is sufficient to avoid the bad behaviour in case of a cache hit, as shown at the top of Lundqvist's example in Figure 1(a). The general case is more difficult to handle, as the processor state at the beginning of the sequence must be considered.

B. Problem Setting

Timing anomalies result from the interference of independent hardware accelerators that each try to improve the timing behavior of the program while preserving the functional semantics. The accelerators work with their individual local *state* not exposed on the architectural level (hence: *non-functional state*) and *allocate resources* according to *predictions* that are based on the *execution history*. Thus they cannot avoid

unfortunate cases: In the best case, the program is accelerated, in the worst case, the program is actually slowed down by going for a local improvement that has adverse effects in the future. The interactions and the use of runtime knowledge makes the timing behavior difficult to predict.

The timing effects are exposed in the pipeline as *delays*, i.e: delayed instructions or *early-outs* i.e: instructions finishing early. For out-of-order pipelines, this leads to secondary effects, as the instruction execution order can change: The out-of-order pipeline is only bound by data and control dependences to preserve functional semantics of the program. Apart from these dependences it can freely reorder instructions to better utilize the execution resources, which it will eagerly do if an instruction is delayed or finishes early.

Thus timing analysis can no longer only analyze the instruction sequence given. It must instead analyze all possible schedules of the corresponding dependence graph, i.e., all valid reorderings of the original instruction sequence, or at least all schedules possible for the given input states. The maximum of the analysis results for the different schedules is taken as the WCET result. Still, the schedule that is part of the global WCET path does not need to be the same as the local worst case schedule. Furthermore, abstractions on the input states are involved, potentially introducing infeasible combinations. Thus the analyzed WCET result is less tight than it could be.

The sensitivity to the exact history and the actual algorithm used also make up the difficulty of predicting them at compile time. For a more detailed discussion of predictability see [3], [15].

III. Terminology

All processors since about 1985 use pipelining according to Hennessy and Patterson [16].

Our out-of-order pipeline model is depicted in Figure 2: The *fetch* stage feeds instructions from the cache or main memory into the *prefetch window*, which effectively decouples the fetch stage from the following pipelines.

The *issue* stage may select any ready candidate instructions from the prefetch window for issuing to the individual execution unit pipelines. At the end of the pipelines, a common reorder buffer (omitted) makes sure that the results are written back in the right order. For *in-order* architectures, the fetch and dispatch mechanisms directly feed the individual pipelines – the prefetch window and reorder buffer are not present.

This issuing of instructions is also termed dynamic, runtime, pipeline or hardware instruction scheduling. In contrast, compile-time, static, software or compiler instruction scheduling denotes the reordering of instructions at compile time during code generation. We will use the terms *compile-time* resp. *hardware scheduling* to distinguish the pre-runtime instruction scheduling mechanisms from the runtime instruction scheduler.

Both types of schedulers have the job of allocating instructions to execution resources, while honoring the program-

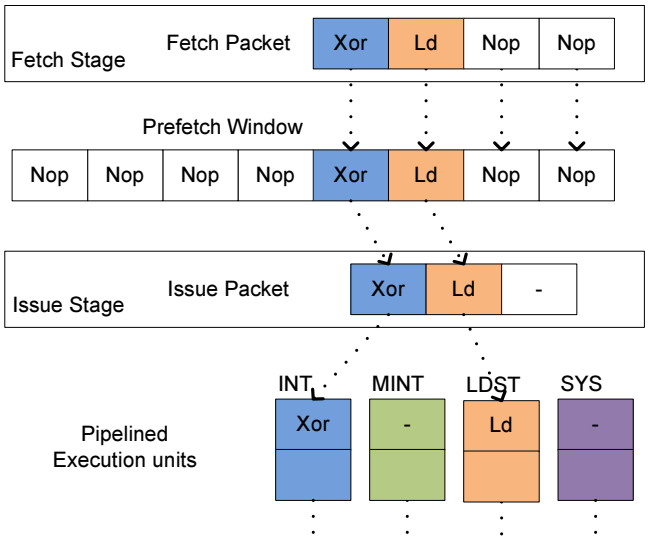


Figure 2: Our out-of-order pipeline model, showing the path of the instructions through the out-of-order pipeline and the relevant storage.

inherent dependences. *Data dependences*, also called *true dependences*, denote dependences, where a definition is used by a later instruction. Figure 1(a) shows the true dependences as arrows for Lundqvist's original timing anomaly example. *Anti dependences* denote restrictions, where a data item must be read by a previous instruction, before it may be overwritten by a later one. *Output dependences* denote cases, where one write must precede another, so that only the second value survives.

These individual dependences form dependence chains that serialize the instructions within a chain, while the instructions from different dependence chains can move freely against each other. Dependences are represented in the *data dependence graph* (DDG), which is a weighted unidirectional graph, where the weight of each edge represents the (min/avg/max) dependence distance. The dependences represent the fundamental limitations on the execution order of the program's instructions. The interaction of these dependences with the allocation of the finite hardware resources is to a big part responsible for timing anomalies.

Compile-time instruction scheduling takes place before timing analysis, so it is automatically covered by the analysis. Hardware instruction scheduling, on the other hand, must be anticipated by timing analysis.

IV. Compiler Countermeasures

As stated in Section I we focus on compiler backend transformations as a countermeasure to timing anomalies. For a given program that suffers from timing anomalies, the goal is to construct a functionally equivalent program that is not susceptible to timing anomalies and does not show the described followup effects. Thus we have to move all scheduling decisions from the hardware instruction scheduler to the software instruction scheduler. As for every valid code transformation, the functional semantics of the original code must not be

altered. Without assumptions about the input code sequence, this only leaves insertion of *functionally useless code* as an option. We will call this code *non-functional*. Even if that does not seem much of an improvement, it still opens up a way to exploit certain hardware limits: In the following, we will describe three different approaches that insert *non-functional code* to avoid timing anomalies in the out-of-order pipeline, exploiting three different hardware limits. All three approaches employ compile-time instruction scheduling to rearrange the code according to a model of the hardware pipeline and minimize the non-functional code needed.

A. Rate NOP Insertion

The first algorithm exploits the limited fetch width of the pipeline: When implementing a CPU, this fetch width must be selected as narrow as possible for memory bandwidth cost reasons. On the other hand, it must be wide enough to sustain the average issue rate and wide enough to recover from pipe break instructions that reassign the fetch pointer in a late pipeline stage, such as conditional jumps, calls, returns or interrupts.

These restrictions on the design decisions lead to fetch widths equal or just slightly larger than the average issue width – rounded up to the next integer multiple of the basic instruction size.

Thus there is a relatively small excess fetch width. If we now – during compile-time instruction scheduling – add just enough NOP instructions to fill the excess fetch width, then there is never an excess instruction waiting to be executed, provided that we start with an empty or flushed prefetch window. This effectively disables the prefetch window, as it at any time only holds the instructions that are ready to be issued immediately. Since no spare instructions are waiting to be executed, the hardware instruction scheduler does not have any potential to issue a pending instruction, if any of the previously issued instructions finishes early. Thus no reorderings of instructions can happen.

This approach very much resembles the "simplified hardware" approach. By scheduling for the underlying VLIW pipeline and by disabling the prefetch window, we essentially simulate the "simplified hardware". The main disadvantage is the lack of architectural support for horizontal and vertical NOP compression [5].

Figure 2 shows the path of the instructions through the pipeline: The instructions for a cycle are padded with NOPs to the size of a fetch packet. This packet is fetched to the prefetch window and the instructions are issued immediately in the next cycle, as no other instructions are available in the prefetch window. As all useful instructions that entered the prefetch window have been issued, the next packet is handled exactly the same way.

Figure 3 shows Lundqvist's example after rate NOP insertion. However, the architectural parameters are consistent with the evaluation for the second architecture in Section V and thus present a wider architecture, but with shorter miss and multiplication latencies to keep the code examples short.

ALU	MINT	LD/ST	SYS
ADD r11,r10,r10	NOP	LD r4, 0(r3)	NOP
NOP	MUL r12, r11, r11	NOP	NOP
NOP	NOP	NOP	NOP
NOP	MUL r13, r12, r12	NOP	NOP
ADD r5,r4,r4	NOP	NOP	NOP

Figure 3: Lundqvist's example with rate NOP insertion

The lines represent issue cycles in our model, while the columns represent the different execution units, i.e., the instructions in the same line are issued together to the individual execution units in the same cycle.

Overhead: Formula 1 presents the instruction overhead O_{instr} for *rate NOP insertion*. It is the difference between fetch width W_F and average issue width of the concrete program $W_{I,avg}$, multiplied with the number of cycles of the code sequence N_{cycles} and divided by the size of a NOP S_{Nop} to obtain the necessary number of NOP instructions.

$$O_{instr} = \frac{W_F - W_{I,avg}}{S_{Nop}} \cdot N_{cycles} \quad (1)$$

This can be subdivided further by separating the constant algorithm-inherent *rate* overhead per cycle from the program dependent overhead. In Formula 2, the first term denotes the impact of filling the difference of fetch bandwidth and maximum issue bandwidth $W_{I,max}$ with NOPs, thus giving the architecture-dependent overhead. The second term denotes the *instruction level parallelism* (ILP) penalty of the program for that architecture: the number of NOPs that are needed to fill the issue slots not used by the program.

$$O_{instr} = \left(\frac{W_F - W_{I,max}}{S_{Nop}} + \frac{W_{I,max} - W_{I,avg}}{S_{Nop}}\right) \cdot N_{cycles} \quad (2)$$

The overhead is a tight bound and only the low-ILP penalty part can be improved by ILP-enhancing compiler-optimizations. It is sensitive to changes in fetch width, issue width and to low ILP, but not to changes in the prefetch window width.

B. Sparse NOP Insertion

This algorithm exploits the limited size of the prefetch window. The algorithm does not always keep the prefetch window in a *mostly-empty* state, but rather works in a *sparse* way: After scheduling a variable-latency instruction for a specific execution unit, it prevents successor instructions for that execution unit and dependent instructions that execute in any unit from entering the prefetch window until the worst case execution time for that instruction has passed. This way the succeeding instructions cannot be executed too early, should the first instruction finish early, but can only execute in their intended place and with their intended parallel instructions.

This algorithm has advantages for architectures that do not expose many variable-latency instructions, as they do not have to continuously pay the code size penalty of maintaining the *mostly-empty state*, but rather only when actually using those features.

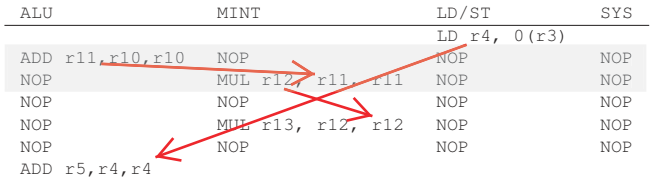


Figure 4: Lundqvist's example with sparse NOP insertion

Figure 4 shows the code for Lundqvist's example, after sparse NOP insertion for the second architecture in Section V: Triggered by the variable-latency load, the worst case is, that the load is at the start of the prefetch window, when it is issued: The prefetch window (grey area) and the fetch packets loaded during the execution cycles of the variable-latency instruction (following 3x4 instructions) must be kept free of instructions issuing to the same unit or dependent from the variable-latencyinstruction - in this case the final ADD. Note how the add and multipies are used first to fill the distance.

Overhead: Formula 3 shows the overhead for *sparse* NOP insertion, considering the described worst case: The prefetch window (size=S_{PW}) and the fetch packets (width=W_F) corresponding to the execution cycles of the variable-latency instruction must be kept free of instructions issuing to the same unit. $L_{i,max}$ denotes the maximum number of execution cycles for the individual variable-latency instruction. The sum over all instructions with variable latency (N_{IVL}) then yields the overall overhead.

$$O_{instr} = \sum_{i=0}^{N_{IVL}} \frac{S_{PW} + L_{i,max} \cdot W_F}{S_{Nop}} \tag{3}$$

In reality the overhead is lower because independent instructions for other units are used by the instruction scheduler to fill the prefetch window before NOPs are inserted as shown in Figure 4. So the bound is not tight, but conservative and also sensitive to low ILP as well as to large prefetch window sizes and fetch widths.

C. Dependence Insertion

The third algorithm builds on the dependence checking done in the issue stage of the pipeline: During compile-time instruction scheduling new artificial dependences are added after each variable-latency instruction. These artificial dependences eliminate any freedom to reorder successive instructions, thus leaving no alternative choices to the hardware instruction scheduler.

The list scheduling algorithm is extended to insert additional dependences to all instructions in the ready set that could be issued to the same execution unit as the just scheduled variable-latency instruction. The same is done for all instructions that are unblocked by the execution of the variable-latency instruction, regardless of the unit they execute in.

These dependences have to use an anti-dependence to a source operand of the problematic instruction, so that they can at the earliest execute in parallel. Then they have to supply a data dependence to the instruction following the problematic instruction in the same execution unit to keep this instruction from getting *ready*. This creates a worst-case fixed-latency dependence path to the followup instructions, thus hiding the variable latency.

The requirements for the dependence code to be inserted are:

1) anti dependence to variable-latency instruction
2) true dependence to successor instruction
3) executed in a different execution unit
4) constant execution time
5) no modification to any register

Requirement 5 can be fulfilled with identity operations like moves or binary operations with the neutral element of the operation as one argument. To fulfill requirement 4, we just need to look at the shortest instructions of the architecture: The simpler ALU instructions like moves and bit operations execute in a single cycle on every architecture. This also fits requirement 3 because the variable-latency instructions are usually not ALU instructions.

Requirements 1 and 2 can only be fulfilled by a single simple instruction, if the registers that carry the dependence – let's name them ra and rb – are the same. In this case even a move instruction would do. However, as these registers typically are not identical, we have to connect them via instructions that form the dependence, but without modifying their data. This can be achieved by using two instructions, where the second one does the inverse operation of the first. We chose the XOR operation for dependence insertion: the neutral element is zero and the inverse operation is an XOR with the same argument. Furthermore the execution time of XOR does not depend on its arguments. Thus it will be single-cycle on every architecture.

The dependence is built using the following code pattern (C-style assembler notation):

```
..= ra ....;    # the variable latency instruction
ra= ra ^ 0;     # anti- & true- dependence on ra
rb= rb ^ ra;    # connect ra dependence chain to rb
rb= rb ^ ra;    # undo rb- modification
..= rb ...      # instruction using rb
```

As can be seen, the minimum latency of our inserted code is three cycles, so two-cycle variable latency instructions will be lengthened to three cycles. Longer latencies can be produced by repeating the first XOR instruction.

In Lundqvist's original example, there is no subsequent load, but only an ADD with a true dependence, thus Figure 5 shows this special case. Note that an additional cycle is required, as there is a resource contention for the ALU, so the first XOR follows the load, while it could actually be parallel - the respective anti-dependence is shown as blue arrow. Alternatives in constructing the dependence could improve this.

Overhead: Formula 4 gives the overhead of the dependence insertion algorithm: D is the set of variable-cycle instructions and $C(d)$ is the set of ready candidates unblocked by scheduling d, unioned with the instructions available after scheduling d, which can be scheduled to the same unit as d. L_d is the maximum latency of d, but a minimum of 3 cycles,

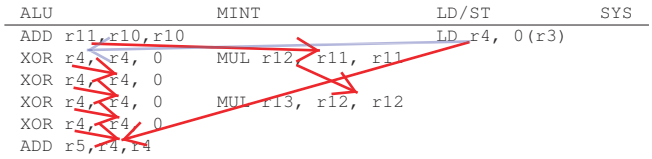


Figure 5: Lundqvist's example with dependence insertion (the XORs are the inserted dependence instructions)

due to the minimum number of instructions to generate the dependence.

$$O_{instr} = \sum_{d \in D} L_d \cdot |C(d)| \tag{4}$$

This is a conservative upper bound for the overhead: the actual overhead is improved by reusing the first part of the dependence chain, which only depends on the source instruction. If more than one successor must be handled, a separate search for the minimum set of source registers within these instructions also enables reuse of the second part of the dependence chain.

Note that this approach also works for duplicated execution units without inhibiting parallel execution in these. If every instruction can only be executed in one type of execution unit, which is not duplicated, then each variable-latency instruction only needs to have a dependence to the next one, effectively linearizing the variable-latency instructions for each execution unit.

D. Implementation

As a test bed we used the LLVM compiler framework, which supplies a configurable framework supporting several backends. We implemented a meld list scheduler that tracks resource usage and latencies over basic block boundaries [17]. This scheduler does a sweep over the Data Dependence Graph (DDG), checking each ready instruction, whether it can be executed with the currently available resources. If so, it is issued. Then all its successors are checked whether they are now ready and if so, are added to the ready set. In addition to the DDG, ready set and the resource tables, a fourth data structure was added to the scheduler, that keeps track of the *code distance* for each execution unit to model the code distance restrictions for the instruction fetch. The standard list scheduling algorithm was extended to track the program counter and distinguish between the cases where *no resources* are available (checkResources) vs. *all ready instructions are disallowed* (checkDistances). In the former case stall cycles are inserted just like in traditional list scheduling, while in the latter case size consuming NOP instructions are issued without advancing to the next cycle immediately. The *rate* algorithm always works in NOP-insertion mode, while the *sparse* algorithm is triggered by variable-latency instructions. For the *dependence* approach the NOP insertion is turned off, and XOR dependence chains are inserted into the DDG for each variable-latency instruction. The additions to the main loop of the meld list scheduling algorithm are depicted in line 6 and 8-12 in Figure 6.

```
1  // Build Data Dependence Graph (DDG)
2  // Sweep DDG, keeping track of ready
3  // instructions, resources & size restrictions
4  while (unissued instructions left)
5    if exist ready instruction i so that
6      checkResources(i) & checkDistances(i)
7        issue(i)
8        if (isVariableLatency(i))
9          insertDependence2ReadySuccessors(i)
10   else if exist ready instruction i so that
11     checkResources(i) & !checkDistances(i)
12       issue(NOP)
13   else // no candidate
14     nextCycle()
```

Figure 6: The modified meld list scheduling algorithm, extensions highlighted

The algorithm is adaptable to different architectures by providing data about the variable cycle instructions. This processor-implementation specific instruction data can be overridden on an instruction-by-instruction basis by supplementing data from an external analysis: Data cache analysis, for example, can thus mark the always-hit and always-miss cases with the minimum and maximum latency respectively, which reduces the amount of inserted code.

The scheduler is freely configurable for the relevant architectural implementation parameters like fetch width, prefetch window size and number of instances of each execution unit. Instruction insertion is done by virtual call interfaces into the concrete architecture-specific code selector, so the scheduler is highly portable.

Our scheduler is executed as the last stage of the compiler, so that the output is exactly the code seen by the timing analysis and the hardware.

V. Evaluation

In our experiments we want to compare the timing-anomaly-free code generated using the three different transformations to the originally scheduled code that might have timing anomalies, assessing the costs in size and execution overhead.

Our current analysis tool so far cannot handle the combinations of branch prediction, speculation and caches that typically come with every out-of-order pipelined processor. In fact this work was started to make such an analysis computationally feasible for complex processors by eliminating as many timing anomalies as possible using code transformations prior to timing analysis. The reduction in analysis complexity (state space) due to the elimination of timing anomalies should be observable in the final tool set either as increase in precision, as less abstraction and overestimation is required, or as faster overall compile/analysis time. In the current work, the more-than-linear reduction in analysis complexity is paid for by an O(n) list scheduling pass during compilation plus any runtime overhead of the modified code. The reduction in state space will also result in less variations in execution time - remember: the extreme case is single path code on a VLIW processor without caches, where only one execution time exists.

Existing hardware with out-of-order pipelines of appropriate complexity also has features like jump prediction, not allowing to investigate pipeline timing anomalies alone. The currently available configurable FPGA cores (NIOS II, Microblaze) do not have out-of-order pipelines, so cannot be used. Thus we resorted to simulation and selected an architecture resembling typical RISC architectures with a low amount of idiosyncrasies and an easily extensible compiler: We chose the LLVM compiler for the ARM architecture, which leaves us the option to switch to the PowerPC backend, as soon as jump prediction is supported.

Short of an end-to-end timing analysis we are more interested in evaluating different settings for the current algorithm, investigating the relevant architectural parameters like fetch width, issue width, prefetch window size and number of instances of each execution unit, evaluating out-of-order-pipelines in separation.

For evaluation we chose evaluation metrics from portable compiler backend construction, which are insensitive to hardware implementation details, so that the results apply to a broader range of typical RISC architectures with the selected parameters: We use the number of *statically inserted instructions* (*inserted_instructions*), i.e. the number of NOPs or XORs inserted, to characterize the code size overhead of our algorithms independently of instruction encoding issues. We use the number of *dynamically executed instructions* (*executed_instructions*) to capture the runtime instruction traffic as seen by the cache or memory interface. We use the number of *static scheduling cycles* (*scheduling_cycles*) to reason about whether the inserted instructions induce additional execution cycles, i.e. increase execution time, or rather just execute in parallel to instructions of the original code, i.e., fill up previously unused issue slots in already existing execution cycles. Neither the *executed_instructions* nor the *scheduling_cycles* alone can capture the dynamic runtime overhead: The *executed_instructions* does not account for parallelism in the instruction pipeline and the *scheduling_cycles* is a static measure of cycles without taking execution weights into account. If, however, the *scheduling_cycles* remain unchanged, we know that a code insertion strategy does not change the dynamic run time of the code. Taken together these metrics can thus adequately express the dynamic impact of our algorithms, while being easy to obtain for different settings of the basic architecture parameters. These metrics are also very portable, as the results apply to all architectures with the same fundamental architectural parameters.

Using the software-only approach, we can model various hardware features by assuming the corresponding variable pipeline timing for the respective instructions. This variable latency triggers the emission of NOPs, which are statically counted during generation and then dynamically counted during simulation without the need of special simulators. This allows for a fair evaluation of size and cycle effects of our algorithm for different hardware parameters.

The measurements for the three algorithms presented in Section IV for two different architectures are presented in Figure 7 and Figure 8. The architectural parameters are given in *number of instructions*. The variable latency instructions are the loads and stores. Lacking a dedicated cache analysis, we excluded load-immediate instructions that load from local *instruction* memory. These are the typical way to load 32 bit literals on ARM - other RISCs use explicit load-literal instructions - the data usually already resides in the instruction cache, so assuming a long latency would make our findings inapplicable to actual RISC architectures. The same was assumed for stack-pointer loads and stores – again a real cache analysis would find out that the stack area resides in the cache. We simply marked these instructions as constant latency through our interface for external analyses.

For our evaluation we assumed that loads take up to four cycles, branches always take two cycles, multi-cycle integer instructions take two cycles and simple integer instructions always take one cycle.

As benchmark set we used the freely available Mälardalen WCET benchmark suite.[1] We fixed the benchmarks to return their specific results indicating success. Only those benchmarks were omitted, that could not produce such a characteristic return value indicating correctness.

A. Results

The general pattern shown in Figure 7 is an increase in code size, but a very modest increase in runtime: Despite the increases in *executed_instructions* the moderate increase in *scheduling_cycles* reflects, that the actual runtime on the desired target configuration did not increase much – or not at all for the $rate$ approach. Still, because the *inserted_instructions* of the $rate$ algorithm is that high, the $dependence$ algorithm provides the better code size vs. execution time trade off.

This effect is best observable with the memory-intensive *insertsort* benchmark: The $rate$ approach shows 580% code size overhead including low-ILP penalty, the $rate$ overhead without low-ILP penalty is 226%, the $sparse$ algorithm has 255% and the $dependence$ approach shows 112%. The *executed_instructions* for the three algorithms reflect, that the overhead code was also inserted into the frequently executed code regions: the numbers go hand in hand with the code size figures- from 466% down to 119%. However, the *scheduling_cycles* show a totally different picture: The $rate$ algorithm shows zero overhead, while the $sparse$ algorithm shows the maximum of 38% on this benchmark. The $dependence$ approach features below 7% execution cycle overhead, which is quite acceptable considering it also has the lowest code size impact.

The general picture mostly follows this extreme case, but shows overall lowest code size impact for the $sparse$ algorithm, but with much higher fluctuations than the $dependence$ algorithm, which is a close second in code size and insensitive to increases in fetch width, issue width, or prefetch window

[1] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

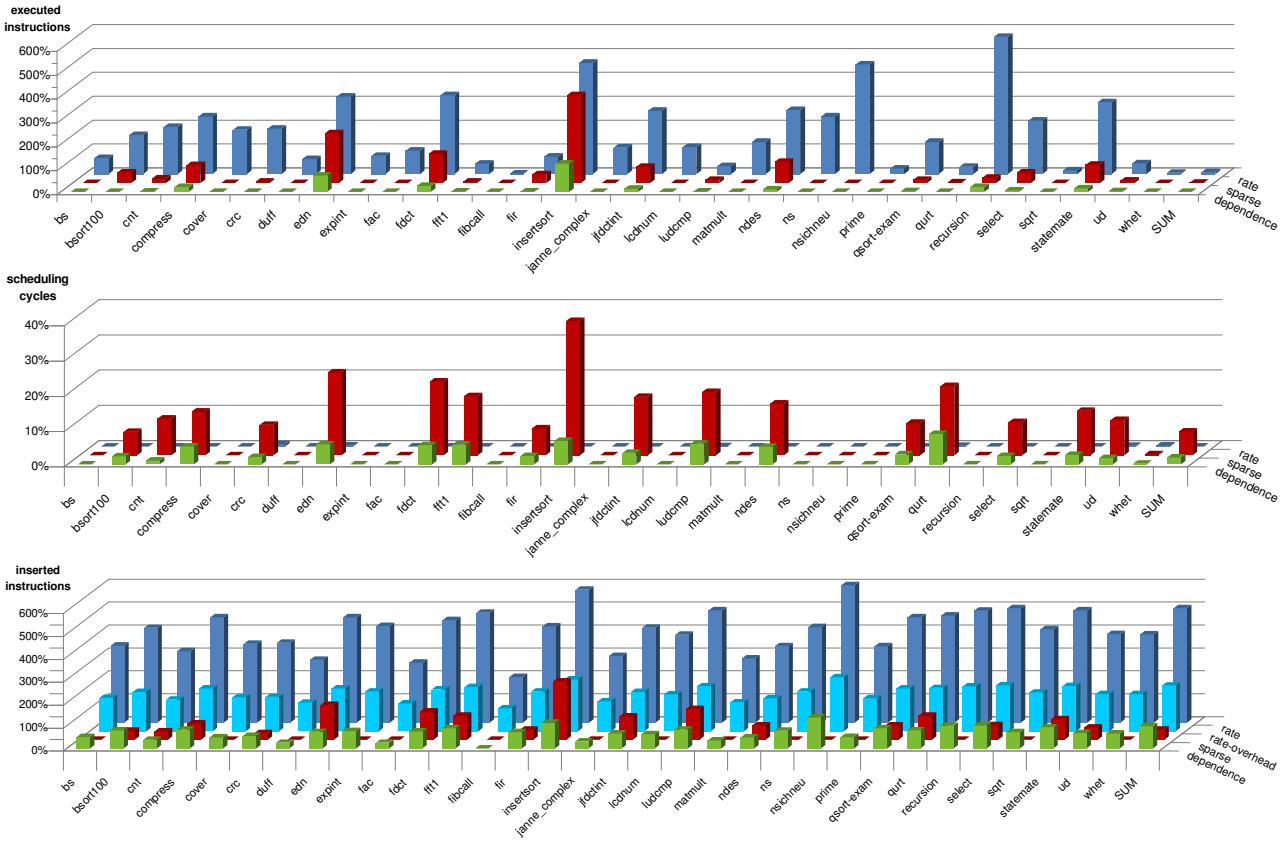


Figure 7: Architecture 1: fetch width: 3 instructions, issue width: 2 instructions, prefetch window size: 6 instructions

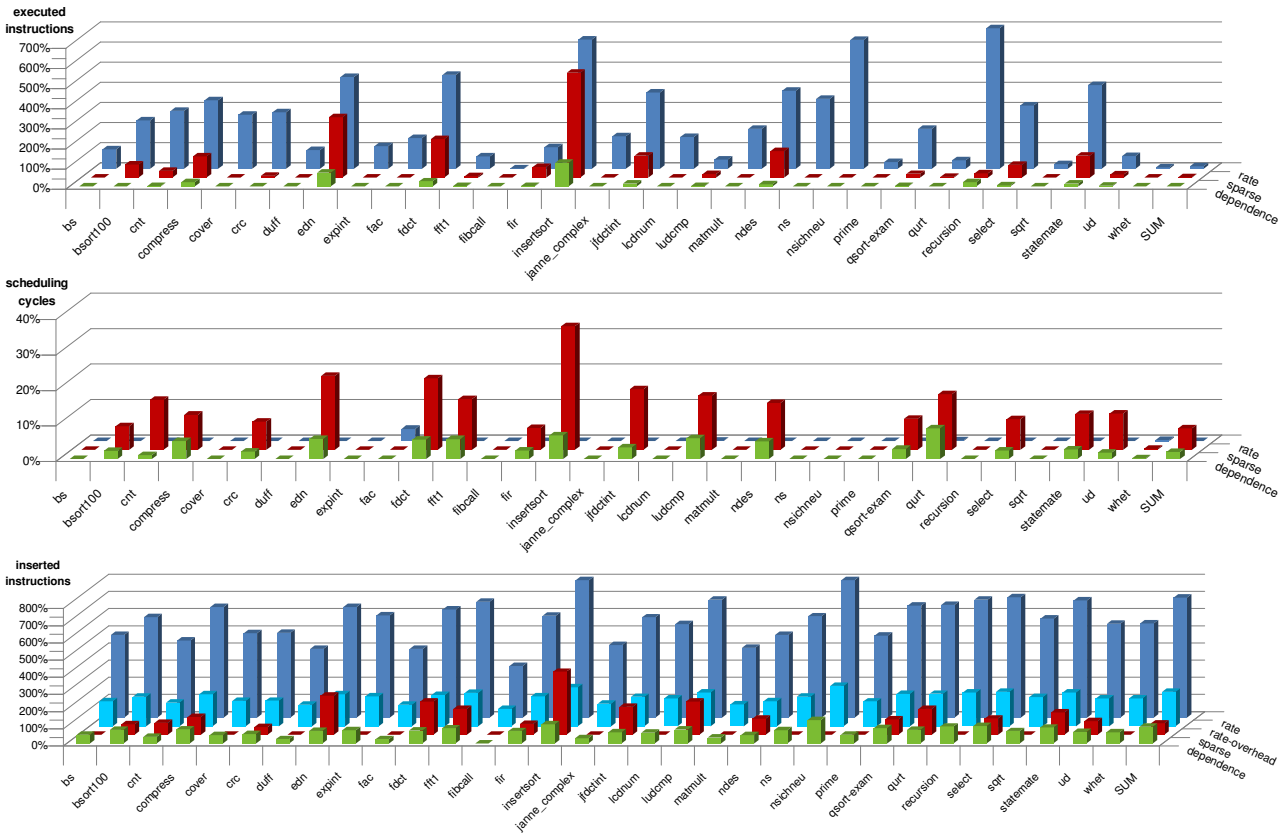


Figure 8: Architecture 2: fetch width: 4 instructions, issue width: 3 instructions, prefetch window size: 8 instructions

size, thus offering the best overall performance. For extremely tight cases, it may be interesting to also evaluate the $sparse$ variant and use the one with the better results. The *recursion* benchmark is one example, where this is beneficial.

Figure 8 shows the results for a wider architecture with larger fetch and issue widths and a larger prefetch window. Except for some scaling, the numbers follow exactly the trend for the narrower architecture, except for one exception: The non-zero increase in *scheduling_cycles* for the benchmark *fac* shows an off-by-one cycle count for this very short benchmark, which results from a different heuristic selection of the scheduling candidates.

B. Discussion

Our measurements demonstrate, that it is viable to eliminate pipeline timing anomalies by software countermeasures: They show quite acceptable properties for the $dependence$ approach, though there is still potential for improvement for the individual approaches.

The high code size of the $rate$ approach is an artefact of the very low-ILP code that the compiler generates and the sensitivity of the metric to the ILP: The $rate$ algorithm needs to fill all previously unused issue slots in the already existing issue cycles with NOPs, before it can fill the excess fetch slots. If only one of four slots was used, filling the other three with NOPs yields 300% overhead. Longer latencies without independent filler instructions lead to the high numbers observed. If three of the four fetch slots can be populated with useful instructions (perfect ILP), then the overhead falls to 33%. Consequentially, the values for *rate-overhead* (low-ILP NOPs ignored) in Figure 7 and Figure 8 all lie between 100% and 200%, memory-intensive *insertsort* being the only exception. They differ more than 1% between the two architectures in only three cases. This is a result of the low ILP of approximately one: in each cycle there is one mandatory fetch-rate NOP, but on average only one useful instruction. The multi-cycle load and store instructions every few cycles worsen that ratio further, if no independent instructions are available.

The high number of inserted NOPs will definitely impact the instruction cache. However, the current WCET benchmarks are all small, wholly fitting into any reasonable instruction cache. As a result, only the compulsory (initial) misses are increased by the increase in code size. Conflict misses are unchanged as are capacity misses. As our transformations do not change the locality of the code, only diluting it, only capacity misses are expected to increase for bigger benchmarks.

The current LLVM-ARM target compiler does not make any effort to increase the ILP. This is reflected by the fact that the number of cycles needed by the compiler's instruction scheduler to schedule the code does not decrease for wider architectures. This means either a different, better parallelizing compiler must be adapted (e.g.: from the VLIW domain), or the current compiler must be reorganized to expose more parallelism. Possible improvements range from addressing anti-dependences stemming from the phase ordering problem of register allocation and scheduling by adding a prepass scheduler, up to high level vectorization and software pipelining. Higher ILP will enable reasonable evaluation for wider architectures.

Thus, the $rate$ algorithm is only acceptable, if the ILP is improved significantly. Then, however, the non-intrusiveness in terms of worst-case runtime is the great benefit of the $rate$ algorithm.

The $sparse$ and $dependence$ algorithms suffer less from the low ILP, because they do not have to maintain the empty state of the prefetch window. Still they would benefit as well from a higher ILP, as more independent parallel instructions would be used instead of NOPs, or would do useful work in parallel to the artificial dependence chains.

Our three algorithms are implemented in a simple meld list scheduler without sophisticated heuristics for candidate selection. Improving on those heuristics is beneficial, once the ILP is raised: For example, it is well known that scheduling instructions with high potential latency as early as possible decreases the overall number of cycles. This will in turn improve the performance of all three algorithms.

The effectiveness of the heuristics can be improved further by better guidance: The insertion of NOPs and XORs currently takes place on the fly. They are not accounted for in the dependence graph, so they do not contribute to the *dependence height*, a metric commonly used by heuristics to reduce the number of total cycles. If they are represented in the DDG, height-based heuristics will benefit and improve the result.

Decreasing the number of variable-latency instructions to handle improves the performance of all three algorithms. Thus, a separate cache analysis should be employed, marking the always-hit and always-miss instructions as constant-latency through our external analysis interface. The same can be done for branch prediction analysis.

VI. Related Work

Of the few contributions in the complex subject of code-based countermeasures against timing anomalies, the closest related ones are the original work of Lundqvist and Stenström and the work of Rochange and Sainrat.

The *program modification method* of Lundqvist and Stenström [2] used the PowerPC-specific *sync* instruction with its high associated runtime costs of full serialization. In comparison, our three approaches should require less run time increase, since less costly and more portable NOPs and XORs are used for partial sequentialization, which mainly populate unused slots in already existing cycles - the $rate$ algorithm not introducing further cycles by design.

The most closely related approach of Rochange and Sainrat [10] already used NOPs for counter measures to pipeline effects in out-of-order processors – the so-called *long timing effects*, but without handling cache effects or jump prediction effects. They require these units to be perfect. However, as their method has been developed out of a suggested hardware solution, they did not control the pipeline on instruction to

instruction level, but used separating blocks consisting of NOPs to let the pipeline run empty on basic block boundaries. In the absence of effects from caches and jump prediction, this eliminates *long timing effects* over basic block boundaries, but it does not eliminate timing effects within each basic block. Hence timing analysis still has to consider all possible schedules. The maximum over all possible schedules is at least as big as any specific schedule, thus the result is costly in both *analysis-time* and *tightness*. Also for long basic blocks as in the innermost loop body of *jfdctint*, they experience a buildup of effects within the 255-instruction-block, that then needs 649 padding instructions to settle. These NOPs drastically lower the effective ILP.

In contrast, we control the pipeline on instruction granularity to select a single schedule and explicitly model cache effects for loads and stores. We also cannot avoid the cache hits or misses and jump mispredictions, but we eliminate the secondary effects on the pipeline - the variable instruction latencies - that create timing anomalies. Further we do offer an interface for separate analyses to improve the performance of our algorithms.

VII. Conclusion

Timing anomalies pose a problem to the designer of safety-critical hard real-time systems, in that they compromise the safety or at least the tightness of the WCET analysis.

We concentrate on the less-researched software-only countermeasures to timing anomalies, in the first step investigating the instruction pipeline.

We presented three solutions for the elimination of timing anomalies that insert NOPs or XORs that limit the usable fetch width or the usable prefetch window size, or introduce new artificial dependences. This eliminates the dynamism from the instruction pipeline, enabling tighter and faster timing analysis.

The solutions using NOP insertion suffer from the low instruction level parallelism generated by the compilation framework, leaving dependence insertion as acceptable compromise in terms of size- and run-time overhead. However, even small improvements in ILP drastically reduce the size overhead of $rate$ NOP insertion, making its invariance in execution cycles more attractive.

Acknowledgments

This work has received funding from the Austrian Science Fund within the research projects “Compiler-Support for Timing Analysis” (CoSTA, http://costa.tuwien.ac.at, contract P18925-N13) and “Sustaining Entire Code-Coverage on Code Optimization” (SECCO, contract P20944-N13).

References

[1] P. Puschner, R. Kirner, and R. G. Pettit, “Towards composable timing for real-time software,” in *Proc. 1st International Workshop on Software Technologies for Future Dependable Distributed Systems*, Mar. 2009.

[2] T. Lundqvist and P. Stenström, “Timing anomalies in dynamically scheduled microprocessors,” in *Proc. 20th IEEE Real-Time Systems Symposium*, Dec. 1999.

[3] L. Thiele and R. Wilhelm, “Design for timing predictability,” *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.

[4] I. Wenzel, “Principles of timing anomalies in superscalar processors,” Master’s thesis, Technische Universität Wien, Vienna, Austria, 2003.

[5] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing - A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann Publishers, Inc., 2005.

[6] L. Wehmeyer and P. Marwedel, “Influence of onchip scratchpad memories on WCET prediction,” in *Proc. 4th Int’l Workshop on WCET Analysis*, Catania, 2004.

[7] A. Arnaud and I. Puaut, “Dynamic instruction cache locking in hard real-time systems,” in *Int’l Conf. on Real-Time and Network Systems (RTNS)*, Poitiers, France, May 2006.

[8] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, “Reliable and precise WCET determination for a real-life processor,” in *Proc. of the 1st Int’l Workshop on Embedded Software (EMSOFT)*, Tahoe City, CA, USA, Oct. 2001.

[9] X. Li, Y. Liang, T. Mitra, and A. Roychoudury, “Chronos: A timing analyzer for embedded software,” *Science of Computer Programming*, vol. 69, no. 1-3, 2007.

[10] C. Rochange and P. Sainrat, “Code padding to improve the WCET calculability,” in *Int’l Conf. on Real-Time and Network Systems (RTNS)*, May 2006.

[11] “Rad750 family of radiation-hardened products,” Web page (Dec 2009): http://www.baesystems.com/BAEProd/groups/public/documents/bae_publication/bae_pdf_eis_rad750.pdf.

[12] J. Schneider, “Combined schedulability and WCET analysis for real-time operating systems,” PhD Thesis, Universität des Saarlandes, Saarbrücken, Germany, Dec. 2002.

[13] J. Reineke, B. Wachter, S. Tesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, “A definition and classification of timing anomalies,” in *Proc. 6th Int’l Workshop on Worst-Case Execution Time Analysis*, Dresden, Germany, July 2006.

[14] R. Kirner, A. Kadlec, and P. Puschner, “Precise worst-case execution time analysis for processors with timing anomalies,” in *Proc. 21st Euromicro Conf. on Real-Time Systems*. IEEE, July 2009.

[15] A. Kadlec and R. Kirner, “On the difficulty of building a precise timing model for real-time programming,” in *14. Kolloquium Programmiersprachen und Grundlagen der Programmierung*, Timmendorfer Strand, Germany, Oct. 2007.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach 3rd Edition*. Morgan Kaufmann Publishers, Inc., 2003.

[17] S. G. Abraham, V. Kathail, and B. L. Deitrich, “Meld scheduling: Relaxing scheduling constraints across region boundaries,” *IEEE/ACM Int’l Symp. on Microarch.*, 1996.