Requirements for and Design of a Processor with Predictable Timing

Christoph Berg[1] *, Jakob Engblom[2], and Reinhard Wilhelm[1]

[1] CS Dept., Saarland University
D-66041 Saarbrücken, Germany
{cb,wilhelm}@cs.uni-sb.de
[2] IT Dept., Uppsala University
Box 337, SE-751 05 Uppsala, Sweden
jakob.engblom@it.uu.se

Abstract. This paper introduces a set of design principles that aim to make processor architectures amenable to static timing analysis. Based on these principles, we give a design of a hard real-time processor with predictable timing, which is simultaneously capable of reaching respectable performance levels.
The design principles we identify are *recoverability* from information loss in the analysis, *minimal variation* of the instruction timing, *non-interference* between processor components, *deterministic* processor behavior, and *comprehensive documentation*. The principles are based on our experience and that of other researchers in building timing analysis tools for existing processors.

Keywords: WCET, hard real-time, embedded systems, computer architecture

1 Introduction

When designing real-time systems, it is important to be able to obtain accurate bounds for the execution time of programs in order to show that a system will always react in time. Execution time estimates are used in real-time systems development to perform scheduling and schedulability analysis, to determine whether deadlines are met for periodic tasks, to check that interrupts have sufficiently short reaction times, and for many other purposes [5, 7, 16].

The aim of *timing analysis* is to give an estimate for the time a given program will take to execute. As the program execution time will vary with input data and variation in the state of hardware features like caches, we are interested in the minimum and maximum value of the execution time. It is very inefficient, or even impossible to obtain these values by simulating all possible combinations of input parameters. We therefore approximate by computing lower and upper bounds, traditionally—and confusingly—called *best case* and *worst case execution time*, respectively (BCET and WCET). These bounds have to be *safe*, in that they must not overestimate the lower bound or underestimate the upper bound. Moreover, they should be *tight*, i.e. they should be as close as possible to the exact values (which in general are not computable).

* Supported by the DFG graduate studies program "Quality Guarantees for Computer Systems"

The focus of timing analysis research has traditionally been WCET, but the survey by Wilhelm et al. [38] indicates that 70% of industrial users also want information about the BCET. This has some consequences for computer architecture, since some designs cannot be modeled for the best case even though there is a worst case [15].

Unfortunately, modern computer architecture trends like superscalar pipelines, out-of-order execution, branch prediction, and caches conspire to make accurate timing estimates very hard to obtain. Architectural innovation generally aims at maximizing average throughput. However, for a predictable real-time system it is more important to have a small span of possible execution times. Such predictability has to be built into a system from the bottom up, and cannot be retrofitted into an unpredictable system by clever analysis techniques.

In this paper, we present design principles that an architecture must meet to minimize the span of execution times and to enable static timing analysis. We also present a design for a predictable processor. We build upon recent results in the field of worst case execution time analysis, where several research groups have documented cases where prediction of execution times becomes difficult [4, 6, 11, 12, 15, 21, 28, 32]. Our goal is to have a processor with good performance where it is easy to obtain safe and tight timing predictions for the best and worst case execution times using automatic analysis techniques.

The goal of this paper is to demonstrate that such processors can be designed, and to serve as a guideline for commercial vendors that want to implement a predictable processor. Such interest exists, and for example, ARM has a “hard real-time” macrocell called the ARM966E-S [2].

Paper Outline. Section 2 introduces previous work, Section 3 shows why execution time prediction can be tricky on current processors, and Section 4 gives general principles for designing predictable processors. Sections 5 to 9 cover our design. We conclude in Section 10 and give considerations for future work.

2 Previous Work

In the past few years, research in the WCET analysis field has identified several hardware mechanisms and implementation choices that make the prediction of execution times difficult.

Timing Anomalies and Long Timing Effects. Lundqvist and Stenström defined *timing anomalies* [28] and showed how they could appear on out-of-order processors. The problem is that, for example, a cache hit for a certain instruction can lead to an overall greater execution time than a cache miss (the commonly assumed local worst case for cache analysis).

Engblom [11, 13] investigated the predictability of processor pipelines, and defined *long timing effects* (LTEs). The presence of an LTE means that more than the immediately adjacent instructions need to be analyzed in order to correctly account for the timing of an instruction (see Section 3.2).

Branch Prediction and Caches. Ferdinand et al. [15] found that branch prediction in conjunction with speculative fetching of cache lines (based on the outcome of the branch prediction) can also cause timing anomalies. Also, the cache system of the ColdFire 5307 processor was found to be impossible to analyze tightly due to the replacement strategy used (see Section 3.1).

Heckmann et al. [21] followed up by showing that the 8-way pseudo LRU replacement algorithm used in the PowerPC 750/755 instruction caches [30, 34] cannot be precisely modeled. When modeled for prediction, only four of the eight ways of each cache set can be used, which reduces the cache size as far as predictability is concerned.

Engblom reported on the occurrence of *inversions* in processors with dynamic branch prediction. An inversion is a case where executing a loop for more iterations reduces the execution time [12]. Petters noted that the global history effect of global branch predictors as employed on the AMD Athlon processor makes determining the worst case very difficult [32].

Memory Management. Bennet and Audsley demonstrated that memory management units (MMU) introduce unpredictability in the execution time of a program since the time required to load a value from memory gets quite variable [6]. This is due to the use of translation lookaside buffers (TLB) to cache virtual-to-physical address translations, where misses may require a table walk in main memory.

DRAM. Atanassov and Puschner [4] report on the effects of DRAM refresh on execution time prediction. While the average effect is small (about 2% increase in execution time compared to not taking refreshes into account), it can have greater effects if refreshes hit at inopportune moments.

Real-Time Processors. Due to a perceived market opening for a processor with predictable timing and especially short interrupt latencies, ARM Ltd. has produced a “real-time tailored” variant of their ARM9 processor core, the ARM966E-S macrocell [2]. The main design goal was to keep the worst case interrupt latency down. Instead of caches, this core uses tightly coupled static RAM (SRAM) for instruction and data memory. There is no MMU. The ARM966E-S actually achieves a rather high predictability, but we believe that it is far from perfect. The instruction set is not very suited for analysis, and the memory system is quite restrictive.

Colnarič and Halang [9, 18] claim that it is infeasible to use any modern computer architecture features in a predictable processor, and propose the use of an asymmetrical real-time multiprocessor without any caches, pipelines, or other dynamic features. In contrast, we think that a standard design with performance enhancing mechanisms can be used, thanks to advances in analysis techniques.

3 Timing Analysis and Current Processors

Our goal is to have a processor architecture whose timing is predictable by static program analysis on the object code. Before defining what we mean by predictability in Section 4, we will list some examples of problems encountered in existing processors

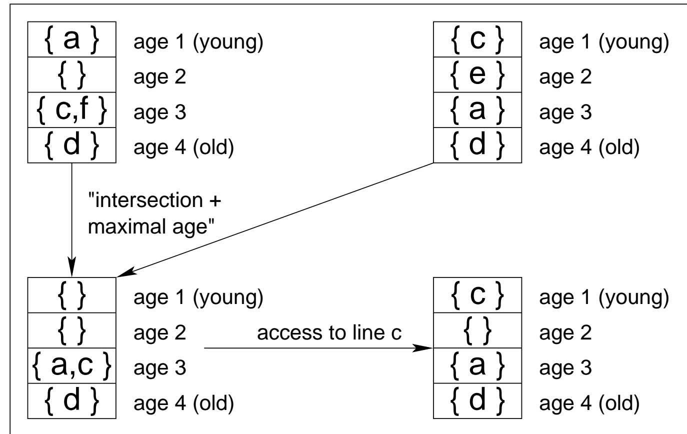

Fig. 1. Merging of abstract cache states

that, independently of the specific method used, make any kind of static timing analysis difficult and—in our view—unnecessarily imprecise.

3.1 Cache Analysis

In general, caches consist of several *sets*, each of which consists of n *ways*. n is called the *associativity*, and is usually 1, 2, 4, or 8. Caches are called *direct mapped* for $n = 1$, and *set associative* otherwise. Each way can hold one *line* from memory. On cache updates, the *replacement strategy* determines the way a new line is put into, evicting the line stored there before. Common strategies are *least recently used* (LRU) and *round robin*.

Cache analysis aims to determine for each memory access which set it will go to, and whether it will hit or miss the cache. *Must analysis* determines whether the access is always a cache hit, and *may analysis* whether it may be a hit (and is a definite miss otherwise) [15].[1] They are necessary for worst and best case analysis, respectively, as a predicted hit allows to decrease the WCET, and a predicted miss allows to increase the BCET. It is possible that a cache is analyzable in the *must* sense, but not in the *may* sense, as we will see below.

An example on how information can be lost and recovered in the *must* analysis is shown in Figure 1. Here we are analyzing a set associative LRU cache. On entering a control flow join, we are forced to merge two different abstract states of a cache set into one. To be safe, we can only include a line the resulting state if it is contained in both states, and we must assign the maximum of both ages to the new age. For example, line e drops out of the cache, and line c is given age 3. A later memory access to c brings the line to age 1.

[1] In [15], abstract interpretation is used, but other methods are also possible [20].

Problems with the ColdFire Cache. The Motorola ColdFire 5307 cache is 4-way set associative, with 128 sets of four 16-byte lines. The replacement strategy is *global round robin*: a counter cyclically selects a way to be replaced. There is only one counter for all sets; the counter is incremented when a line is replaced.

Unfortunately, the ColdFire cache counter cannot be modeled [15]. If a memory access cannot be predicted to be a cache hit or miss, we do not know whether the cache counter will be incremented, and after four such accesses, any information about the counter value is lost irretrievably. This means that each memory accesses can replace any of the four ways in a set, i.e. we only have the certainty that the last line placed in a set will be in the cache. Effectively, round robin replacement behaves like random replacement, and the 4-way set associative cache must be treated like a direct mapped cache with only a quarter of the capacity in *must* analysis (only one way is analyzed for each set). On the other hand, we can never be sure which lines will definitely be evicted, and thus the *may* analysis will collect all lines that were in a set arbitrarily long ago, which is not a useful result.

The ColdFire cache is also *unified*, i.e. the same cache is used for instructions and data. The interdependencies this introduces in the analysis have several consequences. First, instructions and data will evict each other from the cache. Second, as some data accesses cannot be predicted statically to go to a uniquely determined set, they must be treated like going to any set. This evicts *all* data and instructions from the cache model. Third, branch prediction will pre-fetch instructions into the cache. As prediction can span multiple branches, the resulting damage can be large.

3.2 Pipeline Analysis

What can be considered a predictable pipeline depends on the power of the available analysis methods. For all practical purposes, purely manual analysis is limited to using non-pipelined processors where each instruction has a fixed execution time. Using automated analysis techniques, more complex pipeline designs can be accommodated. There are two complexity levels in automatic pipeline analysis, which we call *one-shot analysis* and *fixed-point iteration*.

In a one-shot analysis, each instruction is only visited once or a few times [8, 11, 20, 27, 35]. Each instruction is assumed to have a single deterministic pipeline behavior each time it occurs in the analysis. One-shot analysis is very fast, but has problems handling processors with complex pipelines like out-of-order and superscalar pipelines. Usually, only pairs of instructions are analyzed, but analyzing longer sequences of instructions is required to make the analysis safe for many pipeline structures [11, 20].

In a fixed-point analysis, abstract interpretation is used to collect sets of concrete pipeline stages, iterating until all possible states have been found [15, 21, 33]. Instructions are allowed to have nondeterministic behavior, and the analysis will evaluate all possibilities. Fixed-point analysis can handle more complex pipelines than one-shot analysis, but at potentially higher computational costs. For example, a model of the PowerPC 755 processor used up to 1000 states per instruction in extreme cases [21].

The main culprit in making one-shot analysis infeasible and fixed-point analysis complexity run away is the occurrence of *long timing effects* (LTEs). LTEs occur when an instruction affects the execution of another instruction which is not its immediate

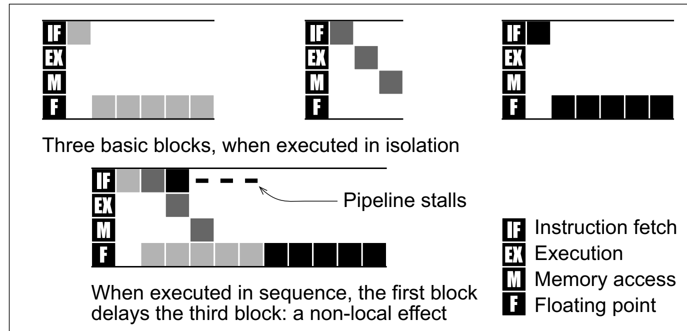

Fig. 2. Non-local pipeline interaction

neighbor in the instruction flow, as exemplified in Figure 2. LTEs commonly occur in processors with multiple, parallel pipelines, but also occur in even the simplest pipelines [13]. In some processors, LTEs can propagate for potentially unbounded sequences of instructions, which makes one-shot analysis impossible. Fixed-point analysis will end up with a high complexity due to the very long LTEs.

The timing anomalies of Lundqvist and Stenström [28] are another example of non-local effects in pipelines. Here, the issue is that *decreasing* the execution time for a single instruction can mean that the execution time of the entire program actually *increases* (and vice versa). This typically occurs in out-of-order pipelines, where the change in one instruction can have an avalanche effect and affect the scheduling of many future instructions. The large search space makes the use of fixed-point analysis necessary, and will cause a great increase in the number of states to be investigated. Timing anomalies have also been found as the result of speculative cache fetches caused by dynamic branch prediction [15].

3.3 Documentation of Timing Semantics

Processor documentation is typically targeted at programmers, and contains optimization hints rather than deep information on the processor implementation. In many cases, the precise design is not publicly documented for competitiveness reasons. For example, Intel does not disclose the precise cache replacement policies used in the cache system for recent machines like the Pentium III and Pentium 4, and the branch prediction algorithms used are only described in very vague marketing terms [24].

The processor documentation describes the functional semantics of the instruction set, i.e. for each instruction, how it modifies the processor state. The state that is "visible" to these *ISA semantics* is called *ISA state*, and includes the the contents of memory and various processor registers. Not included are cache contents, the pipeline state, and other processor components transparent to the programmer.

For timing analysis, the semantics have to also include the timing properties of each instruction. The *timing state* considered in *timing semantics* includes much more components. Cache and pipeline contents, interrupts, branch prediction, and data prefetch

do not affect a program's computational results, but do affect its timing. These features interact with each other, e.g. cache hits may lead to branch prediction reaching further away, and out-of-order pipelines may change the timing of memory accesses. Timing semantics are highly complex, and it is not surprising that the documentation usually does not cover it in the detail needed for precise timing analysis. For example, Atanassov and Puschner found DRAM timing to be incorrectly documented in their study [4].

A more subtle example was observed in the pipeline control of a RISC processor. It inspected the bits in instructions where register numbers were to be found in regular arithmetic instructions to stall dependent instructions until results were available. Since it did not check the type of the instructions, however, instructions containing large immediates instead of register numbers could also cause stalls, depending on the bit patterns in the immediate fields. This was completely undocumented.

4 Principles for a Predictable Processor

Considering the above, it is clear that achieving predictability requires that some care is spent when designing a processor. We have identified principles that need to be adhered to:

Recoverability. Analysis methods have to deal with parts of the ISA or timing state being or becoming unknown. Lack of knowledge can have several sources:

- At the beginning of the analysis, parts of the state may be unknown. For example, the analysis may start with unknown cache contents.
- Analysis methods typically abstract from the concrete state of the processor, combining several concrete states into a single abstract state, either for computability reasons (the analysis may be uncomputable on the concrete state), or for efficiency (the concrete state space is usually very large).
- At control flow joins, information coming from different paths through the program has to be merged. The resulting state is imprecise where the merged states differ.

Given this, recoverability means that knowledge lost can be recovered. Starting from a partially or completely unknown state, the state should become progressively known as we analyze more instructions. No knowledge is lost irretrievably.

A nice example of this is cache analysis, where starting with an unknown cache state, each cache replacement increases our knowledge of the cache contents (for reasonable replacement strategies). A problematic design is the ColdFire cache discussed in Section 3.1, where the cache counter could not be recovered after its value was lost, preventing the precise modeling of the cache behavior.

Another example are global branch prediction schemes where the branch history is hashed with branch addresses to generate indexes into a branch prediction table. It is very hard to track the behavior since as soon as any information is unknown, each branch can affect any other branch [12].

Minimal Variation. To make the analysis of the processor more precise, designers should strive to eliminate variations in the instruction timing. Since we want to analyze

both best and worst case, optimizations that improve the average case, but make the extreme cases drift apart, should be avoided.

For example, it is a bad idea to use shortcut multiplication evaluations, where certain operand values cause the multiplication to run faster (as found in the ARM9 [1]). This likely causes the analysis to assume a range of execution times instead of a fixed execution time for the instruction, which will make the analysis less precise.

Non-Interference. The processor components have to be decoupled from each other in the sense that one component's behavior should influence other components as little as possible. The reason is that if the timing state of a component is unknown, we have to assume a worst case influence on other components that it might affect. Furthermore, interference makes it harder to abstract away parts of the timing state in the analysis.

Examples of interference are diverse: branch prediction influences cache contents and vice versa, and out-of-order instruction scheduling influences the order of memory accesses (hitting the cache analysis). Self-interference occurs in unified caches where instructions and data can evict each other.

Determinism. We want processors to be deterministic in the following sense:

- *Concrete determinism.* The hardware must behave deterministically. In particular, this means that the design needs to be fully synchronous, so that sub-clock-cycle variations in timing have no effect on the aggregate timing. Bus timings need to be conservative so that memory accesses can reliable and consistently be completed in a stipulated number of cycles.
- *Observable behavior.* The timing should only depend on information in the timing state that is observable in an analysis, i.e. information that can be known, or, if lost, is recoverable. Once again, one counterexample are round robin caches that behave non-deterministically in analysis in practice.
- *Abstract determinism.* To make the analysis efficient, the timing state should have a compact approximate representation, and the approximation should not impair the analysis precision significantly. If parts of the state are unknown, this should have a bounded effect on the prediction precision. Otherwise, analysis has to assume a worst case behavior that does not model the real behavior tightly. Component non-interference and minimal variations are prerequisites for this requirement.
  In particular, the number of LTEs should be kept to a minimum for a more precise pipeline analysis (cf. Section 3.2).

Comprehensive Documentation. In order to analyze the timing of a processor, it is necessary to have precise and complete documentation available [3, 10]. Preferably, this documentation takes the form of a cycle-accurate reference simulator that allows for the inspection of the timing state of each functional unit of the processor. The reference simulator should be validated against the RTL code of the processor.[2] To use the sim-

[2] If a pipeline analysis in the style of Engblom [11] can be employed, a black-box processor simulator could be suffi cient to construct a WCET analysis tool. This requires a pipeline with hard bounds on the lengths of long timing effects.

ulator for cycle-accurate verification of a particular setup, it is necessary to be able to parametrize its cache system and memory system with timing information [40].

The printed documentation should give the pipeline profiles for all instructions, as well as a comprehensive description of the functional units in the processor, their behavior, and all interdependencies.

Of these properties, recoverability is the most interesting and the most worthwhile. Overall, we note that while some architectural techniques can significantly lower the overall execution time, as commonly observed with caches, the effort needed for the analysis often increases as these processor features have to be modeled. There is a trade-off between the analysis effort needed (both in construction of the analysis and its execution time), the execution time speedup gained, and the precision lost if the speedup cannot be predicted in the analysis. In a hard real-time system, an average speedup that the analysis cannot translate into a lower worst case bound is useless.

Designing a Predictable Processor

In the rest of this paper, we will present a design for a predictable processor. We will sketch the architecture of a 32-bit embedded RISC processor that is easy to analyze while still providing respectable performance. We do not want to rely on special compiler tricks to provide the predictability, but to provide predictability in hardware where possible. Unfortunately, this is not completely achievable in general. Especially the use of the memory system requires some cooperation by programmers and compilers to enhance predictability (see Section 6).

5 Instruction Set

5.1 Instruction Length

Considering the instruction set encoding, a basic choice to be made is if variable or fixed length instructions should be used. Current embedded systems trends indicate that variable length instruction sets have a better code density, which is an important design parameter for embedded systems. For example, architectures like the Motorola ColdFire, NEC V850 [31], and Texas Instruments C55 [36] use a limited set of instruction encoding lengths to provide compact code while keeping the instruction decoders simpler than for classic CISC architectures. ARM and MIPS have specially designed 16-bit instruction sets (THUMB [1] and MIPS16 [29]) that can be mixed with 32-bit code to improve code density.

On the other hand, fixed length instructions make the instruction cache analysis simpler, and avoid the need of adding extra cycles for branches to misaligned instructions (as found with most variable length instructions). Also, this avoids instructions straddling cache lines, simplifying and improving the precision of cache analysis.

To stick to the principle of minimal variation, we choose a fixed-length, RISC-style (load/store architecture), 32-bit instruction encoding.

5.2 Function Calls

We want to be able to construct a static analysis tool that can be used without access to the compiler that has generated the code for a program, since this case is quite common in practice (even though working inside a compiler is in most ways a more convenient technical solution). It is also desirable to be able to analyze handwritten assembly code. A critical part of static analysis on binary code is to reconstruct the control flow graph of the program, and here recognizing function calls and returns is essential.

The instruction set should help here by providing dedicated jump instructions for function call handling. Typical RISC philosophy has been to use other instructions for this purpose; the PowerPC 755, for example, uses the `blr` instruction for both for function returns and jump tables, making the analysis difficult [37]. To recognize these cases, knowledge about the instruction patterns generated by the compiler was necessary, which we want to minimize since it is a very fragile method. Similar problems have been observed by Holsti et al. [23].

5.3 Immediate Values

To address memory and perform calculation, we need to be able to load arbitrary 32-bit constants into registers. The ARM instruction set has poor support for immediates; in some cases, four instructions are needed to construct a 32-bit constant. To avoid this overhead, compilers employ constant tables embedded within the program code, which are accessed using an offset from the program counter register (which is architecturally visible).

This conflicts with our desire to separate data and instruction memory spaces (cf. Section 6.1), and thus we need to support efficient constant loading using immediate values embedded in instructions. We require that any 32-bit value can be constructed using two instructions (each supplying 16 bits of data in the instruction encoding).

5.4 Memory Addressing

Memory is accessed by explicit load and store instructions. Most computations are done on registers, which makes it easier to predict data accesses. It also reduces the pessimism incurred when having to assume worst case data access times (since fewer instructions access memory).

Our experience in compiler construction indicates that the most important addressing mode to support in an instruction set is register-plus-offset. This can be used to access values on stack frames efficiently (for reasonable-sized frames), as well as for global variable and constant tables using a a reserved register as base pointer.

6 Memory System

We are aiming at embedded systems where programs and constant data are typically stored in ROM or FLASH type memories, and scratchpad RAM is used to hold variable data and stacks during program runs. Figure 3 shows an overview of the memory system we designed. Below follows a detailed discussion and rationale for the design choices.

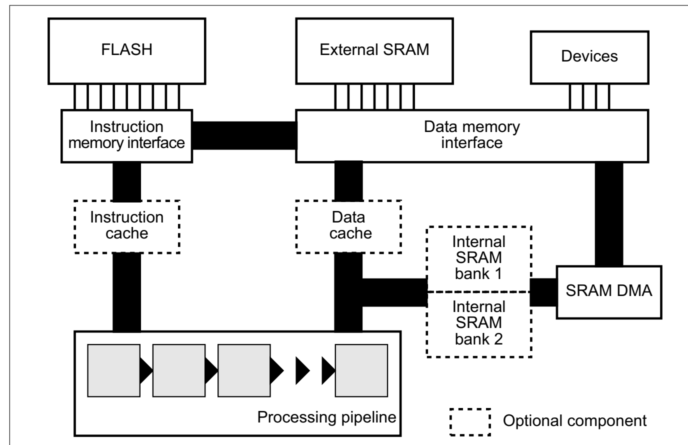

Fig. 3. Overview of the memory system

6.1 Core Memory Interface and Layout

If we use a single memory interface for both code and data accesses, there will be contention between instruction fetches and data accesses. This makes predicting the behavior of the processor more complex, since we need to model and analyze how the processor arbitrates the accesses.

To avoid this contention, we employ a Harvard style memory architecture with separate memory interfaces for instruction fetches and data accesses. We will use a single 32-bit address space, without segmentation. (Segmentation would introduce memory address aliases, which would make the analysis of memory accesses harder.) The address space is statically divided into sections for instructions and data. An interconnecting bus makes it physically possible access data and instructions in the entire memory space, which requires some discipline on the part of programmers and compilers to ensure that predictability is maintained—no data accesses must be made to code space during the timing critical parts of a program.

Both the data and instruction memory interfaces will be at least 32-bit wide, to allow fetching a complete instruction and a complete data word on each clock cycle. Using narrower buses to save some cost is not a good idea on a machine built for predictable and robust performance.

Constant Data. Only the instruction memory will be persistent and have well-defined contents on power-up. Most programs contain data like constants (constant values like strings or initializers for variables) and jump tables, which have to be stored in persistent memory. A simple mechanism that makes sure such data can be used via data accesses at run-time is to copy data over from the persistent memory in instruction space to the RAM in data space at program initialization time. The other possibility is to place some persistent memory in data space, which complicates the memory map but does not impact overall predictability.

Note that most of the variables used in a program will have simple constant initializers that are most efficiently stored as immediates in the program code. This is especially common for local variables that will likely be allocated into registers.

6.2 Bulk Memory

Regular dynamic RAM (DRAM) is problematic in a real-time system due to its need to block memory accesses during its periodical refreshes [4]. This would require pessimistic assumptions about timing to be taken into account at each memory access, which would give us a more pessimistic worst case estimate.

Atanassov and Puschner show that the effect is sufficient to change the observed longest path of a program [4]. However, this change of path is not relevant for analysis safety, as long as the overall WCET is correctly estimated. What is called the *longest path* by most authors is the path on which the WCET tool computes the WCET estimate. Adding the *maxdelay* caused by an unlucky DRAM refresh to the computed WCET makes the results safe. The real worst case execution time may well occur on a different path, since the WCET estimate is by its very nature potentially pessimistic. In the case of two paths competing for being the longest, each having WCETs of less than *maxdelay*, it may indeed happen that the—still safe—WCET is computed on the wrong one of these two. Even so, we conclude that the imprecision introduced by DRAM refreshes is too large to make an accurate timing prediction.

Thus, static memories are preferable for hard real-time systems. Considering our target of embedded systems, we propose to use FLASH for storage of code and constant data. FLASH memory is almost as fast to read as RAM memory, but does not require any refresh. It has become the dominant form of permanent data storage for embedded processors in recent years [39].

For data, we use a block of static RAM. Typically, the need for RAM is much lower than for code memory, and this makes the use of SRAM affordable. FLASH is not an alternative for read/write data storage due to its long write latencies.

6.3 Caches

There is a basic choice to be made whether to use caches or not. Some designers advocate the use of static RAM instead of caches to enhance predictability [2, 14], while others contend that certain cache systems can be analyzed and thus are suitable for use in real-time systems. Generalizing the discussion in Section 3.1, three problems have to be addressed when analyzing a program's memory accesses:

- For set associative caches, the way the loaded or stored datum (or fetched instruction) will be placed into has to be determined. This is only possible if the cache replacement strategy allows to keep track of the history of the data in a set. Clearly, *random* replacement cannot be used as it is unpredictable. *Round robin* replacement uses a counter, which is not recoverable. Once we lose track of the counter, it is lost forever. The strategy we have to choose is *least recently used* (LRU) replacement. It allows to assign an "age" to each cache line; young lines will certainly be in the cache, and old lines will be evicted eventually. It features excellent recoverability.

- Some data accesses cannot predicted to go to a particular set statically. For cache analysis, these accesses have to be treated as possibly going to any set, increasing the ages of all lines in the cache.[3] For a direct mapped cache, this will evict *all* lines from the abstract cache model. Therefore, higher associativity is better for precision. Conventional computer architecture wisdom indicates that set associative caches have superior (average case) performance, but also that associativity beyond four ways is almost impossible to implement perfectly [19].
- If instruction and data accesses share a common cache, it is hard to predict access times, especially in the light of accesses to unknown data addresses that will also affect cached instructions. Conversely, branch prediction would ruin the data cached unnecessarily (see also Section 7).

The current state-of-the-art indicates that instruction caches are quite analyzable, as addresses are always known. In the analysis, it is typically profitable to unroll the first iteration of a loop (or even several) to cope with different execution contexts. Once the cache contains the instructions, the predicted behavior models the real behavior tightly [15].

In the benchmarks supplied by Airbus, Ferdinand et. al found over 90% of the data access addresses to be completely predictable [15], which makes us confident that data caches can indeed be used in a processor designed for predictability.

Hence, we propose separate, 4-way set associative data and instruction caches with LRU replacement for our processor. Since all instruction addresses are known statically, a lower associativity can be used for the instruction cache without losing too much (predictable) performance. Coherence between the two caches is not a problem because the instruction memory is read-only.

Using Caches or Not. If the data SRAM fits on-chip and can be accessed in a single cycle, we will connect it directly to the processor, without using a cache. Otherwise, we will use a larger off-chip block of data SRAM, and a data cache to hide the longer access latencies. The same applies to the instruction FLASH memory: we will only use an instruction cache if the FLASH read latency requires it.

Still, it often makes sense to replace the data cache with static RAM controlled by the application program. For example, multimedia processing operates on streams of data that are only read once, and a cache will not see much reuse.

6.4 Memory Management

A traditional memory management unit (MMU) using page tables in memory cannot be used for a predictable processor, since it gives variable latency at TLB misses[4] [22]. The large latencies of disk accesses forbid the use of demand-paged virtual memory completely. In embedded processors running general-purpose applications, MMUs are

[3] It may be possible to restrict the access to a limited number of sets, but the same discussion applies then as well.

[4] Of course, it is possible to use an analysis analogous to the cache analysis to predict TLB hits and misses, but it is yet unclear how this analysis would perform.

used to provide both memory protection between tasks running on an operating system and convenient virtual memory spaces for dynamically loaded tasks.

For hard real-time systems, where tasks are statically allocated, there is no need for address translation. We will address the physical memory directly in our processor. We will serve protection needs by using protection registers. Such protection registers mark areas of memory that cannot be written to by a certain task, but have no need for backing tables in memory.

6.5 Device Interface

No computer system is meaningful without access to input and output devices that connect it to the surrounding environment. Such devices can be addressed either by accessing memory locations that are mapped to the devices (memory-mapped I/O) or by using special I/O instructions (that address a special memory space used only for I/O, as is done on the x86 architecture). There is no real gain with a special I/O space, and since it would eat up instruction encoding space (for a second set of memory access instructions), we select memory mapping for the peripheral devices added to our predictable processor. The memory mapped devices are connected to the data bus interface of the pipeline.

To keep predictability, we require that all memory accesses to devices complete in a single cycle. The processor-visible registers of a devices need to be clocked synchronously with the core clock (even if the logic in the device is slower, this will only affect how quickly registers change their values, not how fast they are to read).

6.6 Direct Memory Access

Another issue related to devices is the use of direct memory access (DMA) by devices. DMA means that data is moved to and from devices without the processor having to intervene, which reduces the load on the processor. However, from a predictability viewpoint, DMA is bad since it generates memory accesses that can interfere with the processor's own memory accesses. So DMA is not generally suitable for a predictable processor.

There is a DMA scheme that fits well with a predictable design, however. We divide our internal data SRAM into two blocks, and use DMA to move data between the SRAM blocks and external RAM (implicit in the device part in Figure 3), where devices will put their data. While processing data in one block of SRAM, the other block will be written to or refilled from external data sources. Such transfers will be scheduled by the programmer, and thus be completely predictable. The DMA controller will not occupy the bus between the processor and the data memory in this scheme; it will have its own separate bus to the "back" of the SRAM banks.

Note that we need to make a WCET analysis for the transfer. But that is easy since all it involves is moving data between SRAM and some external data source. The WCET of the code executed and the WCET of the DMA will need to be checked against each other.

7 Branch Handling

Dynamic branch prediction algorithms offer many examples of techniques that violate the recoverability requirement. A typical dynamic branch predictor uses the history of the past n branches together with some bits of the instruction address to index a table of branch predictors. Often, the two values are hashed together, which provides a very good prediction ratio on average, but makes static prediction difficult. We agree with Heckmann et al. and Engblom who conclude that dynamic branch prediction is not suitable for real-time systems usage [12, 21].

Static branch prediction—where branches are predicted based on statically known information like their direction or special hint bits in the instruction—on the other hand, can improve the branch handling quite extensively compared to no branch prediction, while still remaining completely predictable. The simple BTFN static predictor (backward branches taken, forward branches not taken) is usually 60-70% accurate for typical embedded applications [17, 25], and provides a significant performance boost at no cost in predictability.

Branch prediction interacts with the instruction cache, superficially violating the non-interference principle. However, this is not a problem for static branch prediction. The prediction chosen for each branch is known to the analysis, and no precision will be lost because of this.

8 Pipeline

To enable an efficient and tight analysis, a pipeline that is amenable to one-shot analysis is preferable. This means that we need to minimize the occurrence of long timing effects (LTEs), as discussed in Section 3.2. The following principles are given in [11, 13]:

- No hardware interlocks between instructions. This means that compilers or programmers will have to ensure that instruction dependencies are respected. This is not a big complication, since instruction latencies will be well defined and the pipeline shallow.
- No parallel pipelines. This is necessary to avoid an explosion in potential instruction combinations.
- Minimize the number of pipeline stages that can have variable latencies, or make sure that they are packed close together.

Based on these principles, our pipeline design is the classic five-state RISC pipeline, where instructions pass through the following stages, as illustrated in Figure 4:

- **IF**: Instruction fetch: fetch instructions to be executed. If we have an instruction cache, this stage will have a variable latency (either one cycle when we hit the cache, or the time required to fetch instructions into the cache in the case of a miss).
- **ID**: Decode: always takes a single cycle.
- **EX**: Execute: arithmetic instructions are computed, and branches are decided. If the branch prediction was wrong, instruction fetch will be redirected and up to two instructions will be squashed. For most instructions, execution will take a single cycle, but integer divides typically cannot be executed in a single cycle. For these

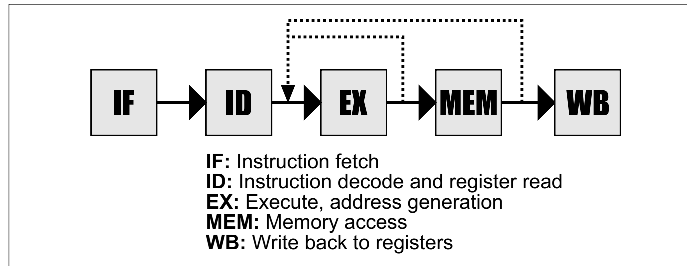

Fig. 4. The five-stage pipeline

instructions, we need to allow a latency of tens of cycles (or remove the integer divide instruction from the instruction set).

- **MEM**: Memory access: access data memory. This stage will always take one cycle if we only use internal SRAM, or a variable amount of time if we use a data cache.
- **WB**: Write back: results are reported back to the register file. This always takes a single cycle.

Considering the absence of interlocks and their implication for instruction scheduling, we note that like in all current processors, there are forwarding paths (dashed lines in Figure 4) in the pipeline that make results computed in the execute stage available immediately to the next instruction. For loads, there needs to be a one-instruction gap between the load and the instruction using the result of the load. Thus, scheduling instructions is not going to be difficult in this pipeline design.

We note that if we do not have a data cache, only the instruction fetch and execute stages have variable latency. In this case, it is possible to limit LTEs to length three by careful selection of the possible latencies in the **IF** and **EX** stages. Thus, it is analyzable using one-shot analysis.[5]

As soon as we also make the latencies of the memory access stage variable (by introducing a data cache), we get a pipeline that suffers from the potential for infinite LTEs for most combinations of latencies for data cache and instruction cache misses. This means that fixed-point pipeline analysis is necessary to perform a safe and tight analysis.

A five-stage pipeline like this easily reaches 500 MHz in current fabrication processes, as exemplified by the IBM PowerPC 750, PMC-Sierra RM7000, and ARM ARM9. This will offer respectable performance, sufficient for most hard real-time applications. In most cases, a lower clock in the order of 100 MHz is sufficient.

[5] Using simulation, we have determined that assuming a 32 cycle latency for divide instructions, useable instruction cache miss latencies are 4 to 16 cycles.

9 System Level

Interrupts are a thorny issue for real-time systems design. On one hand, the popular definition of a system suitable for real-time systems usage is that it has short interrupt latency (indeed, the ARM1136E-J processor has a choice of high performance and fast interrupt modes [26]). On the other hand, predicting the execution time of a program when interrupts are employed becomes much trickier since interrupts cause cache pollution and pipeline flushes, at essentially random points in a program's linear execution. Thus, we need to restrict the use of interrupts to maintain predictability.

The most predictable method is to use a fixed static cyclic schedule. Interrupt handlers will be part of this fixed schedule, which means that they will not occur in the middle of other tasks, upsetting the system.

If preemptive dynamic scheduling is desired, we propose the use of limited preemption in the form of preemption points (where the program can be interrupted and switched out). Preemption points are most easily implemented using a special "preemption check" instruction. When such an instruction occurs, the processor will check for an interrupt. Preemption points will have to be inserted into the code by the programmer or the compiler. Since we assume that all loops are bounded and all code trusted, it is not necessary to require preemption points in all loops (which is the common requirement when using preemption points in virtual machine languages like Java and Erlang).

10 Summary and Future Work

We have identified several principles that processors need to adhere to have a timing behavior amenable to static program analysis. They are *recoverability* (knowledge lost in an analysis can be recovered), *minimal variation* (the difference between best and worst case behavior should be small), *non-interference* (processor components do not have tightly coupled behavior), *determinism* (starting from a timing state, it must be possible to predict the timing tightly, and approximations of the timing state must not impair the overall analysis precision excessively), and *comprehensive documentation* (to enable the construction of analysis tools).

Based on these principles, we have proposed a processor architecture for hard real-time systems that is suited for timing analysis, while providing respectable performance. The main design features are a memory system that separates instruction and data memory as much as possible to reduce memory access interference, the use of FLASH and SRAM as main memory to provide uniform access times, cache designs that are amenable to static analysis, static branch prediction, and a pipeline structure that exhibits few long timing effects.

We intend this design both to show how a particular predictable processor can be designed, and more broadly to serve as a guideline for computer architects that want to implement processors or processor cores for hard real-time systems. The authors hope that this work will have a beneficial impact on the predictability of processors on the market, and that the use of static analysis in real-time systems design will be encouraged.

Future Work. The processor proposed here is currently not more than a design. Detailed design simulation or implementing it as an FPGA prototype will show whether the decoupling of processor features provides the intended predictability boost. More work can be done on improving the predictability of features that we only covered shortly here, such as dynamic branch prediction schemes, and in obtaining more insight into the effect of the pipeline structure on the occurrence of LTEs.

Acknowledgements

The authors would like to thank Andreas Ermedahl, Christian Ferdinand, Reinhold Heckmann, Bengt Jonsson, Marc Langenbach, and Stephan Thesing for fruitful discussions.

References

1. ARM Ltd. *ARM 9TDMI Technical Reference Manual*, 3rd edition, Mar. 2000. Document no. DDI 0180A.
2. ARM Ltd. *ARM9E-S Flyer*, 2001. Document no. D0I 00798A.
3. P. Atanassov, R. Kirner, and P. Puschner. Using Real Hardware to Create an Accurate Timing Model for Execution-Time Analysis. In *Proc. IEEE Real-Time Embedded Systems Workshop, held in conjunction with RTSS 2001*, Dec. 2001.
4. P. Atanassov and P. Puschner. Impact of DRAM Refresh on the Execution Time of Real-Time Tasks. In *Proc. of the Int'l Workshop on Application of Reliable Computing and Communication (WARCC), held along with the IEEE 2001 Pacific Rim International Symposium on Dependable Computing (PRDC 2001)*, Dec. 2001.
5. N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed priority pre-emptive scheduling: an historical perspective. *Journal of Real-Time Systems*, 8(2/3):129–154, 1995.
6. M. D. Bennet and N. C. Audsley. Predictable and Efficient Virtual Addressing for Safety-Critical Real-Time Systems. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*. IEEE Computer Society Press, June 2001.
7. L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano – a revolution in on-board communications. *Volvo Technology Report*, 1:9–19, 1998.
8. A. Colin and I. Puaut. A Modular and Retargetable Framework for Tree-Based WCET Analysis. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*. IEEE Computer Society Press, June 2001.
9. M. Colnarič and W. A. Halang. Architectural support for predictability in hard real time systems. *Control Engineering Practice*, (1):51–57, 1993.
10. J. Engblom. On hardware and hardware models for embedded real-time systems. In *Proc. IEEE Real-Time Embedded Systems Workshop, held in conjunction with RTSS 2001*, Dec. 2001.
11. J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Dept. of Information Technology, Uppsala University, Apr. 2002. Acta Universitatis Upsaliensis, Dissertations from the Faculty of Science and Technology 36, `http://publications.uu.se/theses/91-554-5228-0/`.
12. J. Engblom. Analysis of the Execution Time Unpredictability caused by Dynamic Branch Prediction. In *Proc. 8th IEEE Real-Time/Embedded Technology and Applications Symposium (RTAS'03)*, May 2003.

13. J. Engblom and B. Jonsson. Processor Pipelines and Their Properties for Static WCET Analysis. In *Proc. Second International Workshop on Embedded Software (EMSOFT 2002), LNCS 2491*, Oct. 2002.
14. J. Eyre. The Digital Signal Processor Derby. *IEEE Spectrum*, 38(6), June 2001.
15. 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. First International Workshop on Embedded Software (EMSOFT 2001), LNCS 2211*. Springer-Verlag, Oct. 2001.
16. J. Ganssle. Really Real-Time Systems. In *Proc. Embedded Systems Conference San Fransisco (ESC SF) 2001*, Apr. 2001.
17. L. Gwennap. New Algorithm Improves Branch Prediction. *Microprocessor Report*, 9(4), Dec. 1995.
18. W. A. Halang and M. Colnarič. On Safety-Critical Computer Control Systems. In *Proc. Tenth IEEE Symposium on Computer-Based Medical Systems*, June 1997.
19. J. Handy. *The Cache Memory Book*. Academic Press, 2nd edition, 1998.
20. C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.
21. R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *IEEE Transactions on Real-Time Systems*, 2003. To appear.
22. J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2nd edition, 1996. ISBN 1-55860-329-8.
23. N. Holsti, T. Långbacka, and S. Saarinen. Using a Worst-Case Execution-Time Tool for Real-Time Verifi cation of the DEBIE software. In *Proceedings of the DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, Sept. 2000.
24. Intel. *Intel Pentium 4 and Intel Xeon Processor Optimization*, 2001. Document Number: 248966-04.
25. M. Levy. Exploring the ARM1026EJ-S Pipeline. *Microprocessor Report*, April 30, 2002.
26. M. Levy. MPF Hosts Premiere of ARM1136. *Microprocessor Report*, October 21, 2002.
27. S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An Accurate Worst-Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.
28. T. Lundqvist and P. Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS'99)*, Dec. 1999.
29. MIPS, Inc. *MIPS32 Architecture for Programmers Volume IV-a: The MIPS16e Application-Specifi c Extension to the MIPS32 Architecture*, Aug. 2002. Document no. MD00076.
30. Motorola Inc. *MPC750 RISC Microprocessor User's Manual*, Aug. 1997.
31. NEC Corporation. *V850 Family 32/16-bit Single Chip Microcontroller User's Manual: Architecture*, 4th edition, 1995. Document no. U10243EJ4V0UM00.
32. S. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Technische Universität München, Aug. 2002.
33. J. Schneider and C. Ferdinand. Pipeline Behaviour Prediction for Superscalar Processors by Abstract Interpretation. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, May 1999.
34. F. Sebek. Cache Memories and Real-Time Systems. Technical Report 01/37, Dept. of Computer Engineering, Mälardalen University, Sept. 2001.
35. F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.
36. Texas Instruments Inc. *TMS320C55x DSP Mnemonic Instruction Set Reference Guide*, Apr. 2001. Literature Number: SPRU374E.

37. H. Theiling. Extracting safe and precise control flow from binaries. In *Proc. 7 th International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, Cheju-do, South Korea, December 2000.
38. S. Thesing, R. Wilhelm, J. Engblom, and D. Whalley. Industrial Requirements for Worst-Case Execution Time Analysis Tools. In *Worst-Case Execution Time Analysis Workshop 2003 (WCET03)*, Porto, Portugal, June 2003.
39. J. Turley. *The Essential Guide to Semiconductors*. Prentice Hall Professional Technical Reference, Upper Saddle River, New Jersey, 2003.
40. Virtutech Inc. *Simics User Guide for Windows, version 1.6.5*, Apr. 2003.