Processor Support for Temporal Predictability — The SPEAR Design Example

Martin Delvai Wolfgang Huber Peter Puschner Andreas Steininger
Institut für Technische Informatik, TU Vienna
A-1040 Vienna, Austria
{delvai, huber, steininger}@ecs.tuwien.ac.at,
peter@vmars.tuwien.ac.at

Abstract

The demand for predictable timing behavior is characteristic for real-time applications. Experience has shown that this property cannot be achieved by software alone but rather requires support from the processor. This situation is analyzed and mapped to a design rationale for SPEAR (Scalable Processor for Embedded Applications in Real-time Environments), a processor that has been designed to meet the specific temporal demands of real-time systems. At the hardware level, SPEAR guarantees interrupt response with minimum temporal jitter and minimum delay. Furthermore, the processor provides an instruction set that only has constant-time instructions. At the software level, SPEAR supports the implementation of temporally predictable code according to the single-path programming paradigm. Altogether these features support writing of code with minimal jitter and provide the basis for exact temporal predictability. Experimental results show that SPEAR indeed exhibits the anticipated highly predictable timing behavior.

Keywords: *real-time processor, response time, response jitter, worst case execution time, one path programming*

1 Introduction

Real-time computer systems must not only produce correct results in the value domain but must also meet the temporal requirements of the application. These requirements define the (minimal or maximal) response time to state changes in the environment and the jitter of the response time that the application allows for.

The developers of real-time systems meet the requirements of applications by using appropriate system architectures, scheduling, and implementation models, and by verifying the timing of the resulting system against the specification. Clearly, the quality of the prediction of the temporal behavior of an application strongly depends on the accuracy with which the time consumption of all involved operations on the target hardware can be determined. If the durations of hardware operations are variable then the timing of applications running on this hardware will have a considerable jitter and exact code execution times will be hard to predict. On the other hand, if the timing of hardware operations is (almost) invariable, then applications will be able to meet even tightest jitter constraints. Further it will then be possible to precisely predict the execution time of code executing on this target.

This paper presents SPEAR, a processor that has been designed to meet the specific temporal demands of real-time applications. The processor is not targeted for high-end performance, but it allows applications to trigger and execute code with small temporal jitter. It has a cache that supports cache locking as a first basic aid to obtain predictable memory access times. Further the processor provides an instruction set that only has constant-time instructions and supports the implementation of temporally predictable code according to the single-path programming paradigm [8]. This bundle of properties supports writing of code with minimal jitter and provides the basis for exact temporal predictability, which makes SPEAR an attractive choice for hard real-time applications.

The paper is structured as follows: Section 2 summarizes the requirements that typical hard real-time applications impose on SPEAR. Section 3 describes the architecture of SPEAR. In Section 4 we study the real-time capabilities of the processor. We show how these capabilities contribute to achieving our goals. Section 5 presents some results of experiments that we performed to explore the limits of SPEAR's temporal predictability. Section 6 summarizes and concludes the paper.

2 Real-time Requirements

The distinction between a real-time system and a non-real-time system is given by the application rather than the computer itself. Therefore we will first investigate the distinctive requirements of real-time applications. We map these requirements to the computer architecture in a second step.

2.1 Application View

According to [6] and [9] ”A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced.” Depending on the consequences that a violation of this time-domain requirement yields, we distinguish between *hard* and *soft* real-time systems: While occasionally missing a deadline may be tolerable in a soft real-time system, any timing violation has the potential for severe consequences (personal injury, substantial financial loss, ...) in a hard real-time system.

The temporal behavior of the computer viewed as a control element in a real-time application can be decomposed into *response time* and *response-time jitter*.

- **Response time** is the time required by the computer to generate an appropriate response to an external (or internal) event. Clearly, an appropriate response time is a vital property of virtually every real-time computer. For a given control algorithm it is mainly determined by the performance of the computer but also by other architectural features that will be discussed later on.

- **Response-time jitter** is the indeterminism of the response time. Its effect is not as obvious as that of response time, however it may be the reason for several substantial problems such as insufficient stability margins of control loops [5], limited accuracy of synchronization algorithms [6], or noise in sensor readings. Jitter is not only introduced by the actual variability of the response time, but also by the inaccuracy of response time prediction. Both sources of jitter will be examined in more detail throughout the paper.

Having the minimization of response time in mind it is tempting to rate performance as the most important property of a real-time processor, and experience shows that this is often true. Over-abundant performance can be used as a brute force method to overcome some of the above problems. On the other hand ”real-time” is substantially different form ”high performance” for several reasons. First of all, timeliness does not imply high speed [9]. The dynamics of processes to be controlled is spread over a wide range and so are the requirements on response time and jitter. A considerable portion of real-time applications does not necessarily require high performance. Generally, the cheapest and simplest processor that still meets all deadlines of an application is the optimal one. This is not only true from an economic point of view; minimum die area and minimum power consumption are also important attaining a low failure rate. Furthermore, certification of a complex processor architecture may turn out to be extremely tedious. So in summary there is a quite high price to be paid for over-abundant performance.

The judgment whether the performance of a processor is sufficient for a given application requires a very accurate model of the timing behavior of the whole system. Moreover, in a hard real-time system a *guarantee* is required that all deadlines will be met. Therefore, from an economic as well as from a safety point of view a processor with reasonable performance whose timing behavior is well understood is preferable to a faster one with an inferior timing model.

From the above discussion we can derive two essential properties of a real-time processor:

- The timing behavior must be as predictable as possible. This is important for keeping response-time jitter low.

- The timing behavior must be easy to model. This will allow for an accurate assessment of execution time (and hence response time), which, in turn, serves two purposes:

  – It keeps the proportion of jitter artificially introduced by the uncertainty of the response time estimation to a minimum, and

  – it facilitates a close matching between the processor’s capabilities and the application requirements in the temporal domain, yielding an optimal resource utilization while still guaranteeing all deadlines.

By the definition given at the start of this section every hard real-time system is safety critical. Hence safety and dependability features are as well of concern for a real-time processor. Also the scalability of memory size and processor performance is an important issue because it allows system designers to adopt the processor to different or changing requirements [9].

2.2 Implications on the processor

Both response time and response-time jitter are measured from the occurrence of an event until the respective action is visible at the output. As can be seen in Figure 1 this period includes (1) recognition/synchronization of the event, (2) activation of the corresponding routine (context switch) and (3) execution of this routine[1]. In the following we will take a closer look at these phases and how they contribute to the real-time properties of a processor.

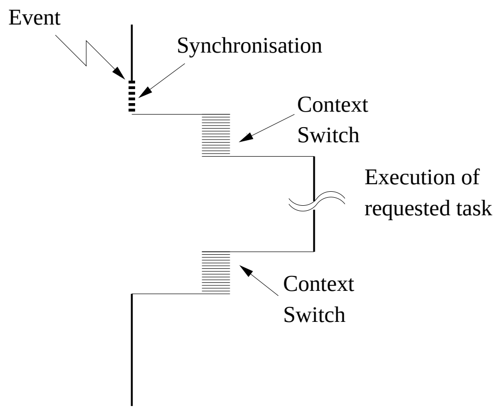

Figure 1. Response to an external event

1. **Synchronization:** Every real-time system must be able to deal with events that are not synchronous to system operation and the system clock (e.g., asynchronous events from the environment or time signals). The interrupts signalling these events must be synchronized at two levels: *Signal synchronization* involves capturing and recognizing the incoming interrupt with the next active edge of the system clock. *Instruction synchronization* on the other hand inserts the pending request into the ongoing instruction flow of the processor. Before the pending request can be serviced, it is necessary to carry on with the execution of the current task to a point that safely allows to resume it afterwards.

   With a processor architecture supporting out-of-order execution, atomic sequences, multi-word and multi-cycle instructions or a sophisticated memory hierarchy this step may significantly complicate the timing model.

2. **Context switch:** This task can be further subdivided into the following two phases:

   - *Save context of current task:* The current processor context must be saved as a starting point when resuming execution after having serviced the request. The processor context usually comprises a set of registers such as program counter, processor status word(s) and potential callee saved data registers. Hardware that automatically saves the context is obviously more efficient than the conventional stack based software strategy. However, from the real-time point of view a context of fixed and moderate size paired with efficient and predictable support for stack manipulation is also sufficient to avoid jitter in this step.

   - *Identify the requested service:* In virtually every system there exists a variety of potential asynchronous events that require different service actions. Often different interrupt signals are available, each of which can be directly associated with an appropriate service. The most important architectural contribution in this phase is an efficient and time-deterministic mapping from the interrupt source to the start address of the corresponding service routine (e.g., vector table access). The hardware architecture, however, can not eliminate the timing uncertainties related to the scheduling of conflicting interrupt requests on principle. Without appropriate means at the system level (such as in the time-triggered paradigm) there will always be the need to postpone one request while servicing another, which inevitably introduces considerable response jitter.

3. **Execution of the requested service routine:** The actual execution time of the service routine usually constitutes the major part of the response time. Its indeterminism (e.g., due to program flow alternatives) plus the uncertainty in its determination result in a response jitter that has to be accommodated. A fixed instruction execution time (no pipeline, flat memory hierarchy) can largely alleviate the timing model of the hardware, and the instruction set can support a single-path programming model (see Section 4.3.2).

A small jitter and good temporal predictability in the three mentioned parts of the response to a trigger event constitute the basis for constructing highly predictable real-time systems. This is true for all types of time critical systems ranging from statically scheduled, time-triggered systems to highly dynamic systems with sporadic task activation. Thus, an architecture that minimizes jitter and supports predictability provides a universal basis upon which real-time systems with dependable timing can be built.

The next section describes the SPEAR architecture in general. Following this we focus on the timing aspects and explain how SPEAR guarantees low jitter and supports the temporal predictability of application software.

[1]We abstract from task preemption from other tasks as preemption handling does not require any mechanism not covered here.

3 The SPEAR–Architecture

The most important design goal for SPEAR [3] was to define a processor with a highly deterministic timing behavior. Every action within the processor core should have an execution time that is predictable and easy to model. Other design goals were modularity and scalability, so that SPEAR can be easily adapted to different requirements and allows real-time studies with varying hardware models.
The processor core is implemented in an APEX 20KE300, an embedded programmable logic device from ALTERA. SPEAR requires about 15% of the programmable gates (45.000 gates) and 45% of the 18 kByte embedded memory blocks[2]. This implementation runs at a maximal clock frequency of 40 MHz.

3.1 Core

SPEAR features a 16 bit processor core comprising a set of 32 registers. 26 registers are general purpose, 6 have special functions: Three are coupled with dedicated instructions to implement stacks in an efficient way and the other three registers are used to save the return address in case of a subroutine call or an exception. On-chip data- and instruction memory are both 4 kB in size.

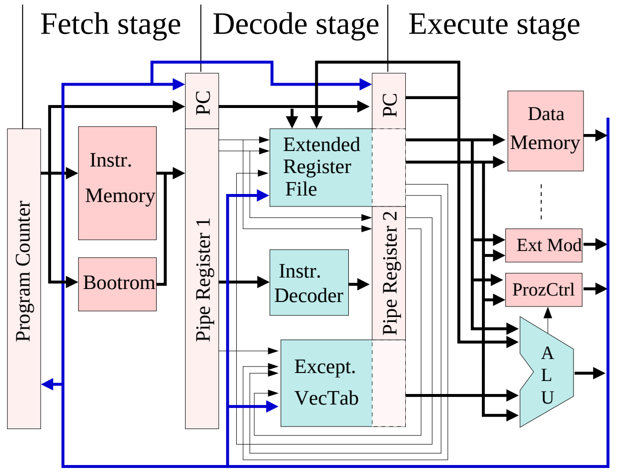

Figure 2. Architecture of SPEAR

Following the arguments of Section 2.1, maximum performance has not been a primary design objective for SPEAR. However, in order to achieve reasonable performance that makes SPEAR still suitable for a wide range of typical applications, we felt that a pipeline architecture was needed. An instruction is executed in the following three pipeline stages: fetch, decode and execute.

[2]The values above refer to the basic processor.

The instruction set of SPEAR exclusively consists of one-word instructions. Every instruction is clocked synchronously through each of these pipe stages yielding a constant execution time of 3 clock cycles. All data hazards potentially introduced by the pipe architecture are completely eliminated by forwarding, while control hazards associated with jump instructions are resolved by automatically flushing the fetch and the decode stage in hardware. This strategy avoids temporal uncertainties and keeps the timing model simple. A detailed description of the processor can be found in [3, 2].

3.2 Scalability

Extension modules are used to adapt the SPEAR-core to different requirements. They communicate with the processor over a well defined interface consisting of 8 registers each 16 bits wide. The extension modules are mapped to the top address space of the data memory. For the processor the extension modules are just storage positions which can be accessed with simple load and store instructions. Therefore from the processors point of view it makes no difference whether the extension is a simple sensor, actuator or a complex SCSI-Controller unit.

Extension modules are not only used for peripheral interfaces but also for adapting the functionality of the core. For instance, a *protection control unit* can be added to provide the processor with resource protection capabilities.

3.3 Memory Architecture

As shown in Figure 2, SPEAR provides separate on-chip memory for data and instructions, each with a capacity of 4kB. For applications that are memory-intensive up to 128 kByte of external memory for instructions and up to 127 kByte for data can be added. The upper 1kB of the data memory space is used by the extension modules.

The external memories are equipped with caches, each of which has a configuration interface implemented as an extension module.

Both caches are two-way set-associative with a block size of two words. The instruction cache is read-only by its nature. The data cache uses write-through policy: Updating a word in the cache forces a write to the memory.

4 Real-time support of SPEAR

As explained in Section 2.2 the response time can be divided into three parts: the recognition/synchronization upon occurrence of an event, the context switch, and the execution of the requested task or interrupt handler. In the following we investigate how SPEAR handles each of these parts.

4.1 Synchronization

We denote the synchronization period as the time between the occurrence of an interrupt and the instant of time at which the address of the first instruction of the respective ISR (interrupt service routine) is loaded into the program counter. The synchronization mechanism is completely implemented in hardware, so it is independent from software and program flow. Figure 3 depicts the timing for entering an ISR upon the occurrence of an interrupt.

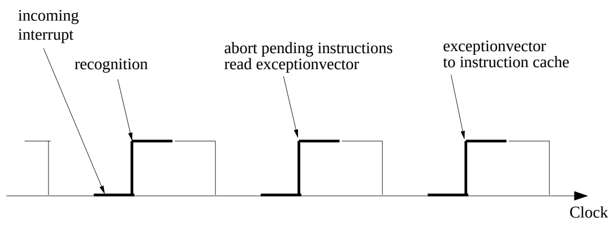

Figure 3. Synchronization steps

The incoming (asynchronous) interrupt is captured with the first active (rising) clock edge. This forms the recognition of the interrupt. Since we cannot assess/control when exactly between two successive active clock edges the event occurred, a detection jitter of one clock cycle is unavoidable.

On the active clock edge immediately next to the interrupt recognition the processor gets ready to fetch the first instruction of the interrupt service routine. At this time there are still instructions in the pipeline that have been loaded before the interrupt had occurred: there is one instruction in the fetch stage and one in the decode stage of the processor. As SPEAR's instruction set consists only of single-word instructions, the pending instructions can be aborted immediately without losing predictability of execution time of both the main task and the ISR (see Section 4.3.1).

At the time of the third active edge the interrupt vector is loaded into the program counter. The information about the interrupt vector is used to fetch the first instruction of the corresponding interrupt service routine. This instruction fetch starts without further delay at the time the third active edge occurs.

Except for the recognition time of the interrupt all actions involved in the synchronization have a constant duration. Thus, for SPEAR the overall synchronization has an unavoidable jitter of one clock cycle and a fixed delay of two clock cycles.

4.2 Context Switch

Programs executing on SPEAR perform three major steps during a context switch. In the following we explore the functionality and the timing of each of these steps.

a) **Save program counter and program status word:** The processor hardware automatically performs these operations when an interrupt is detected. These operations are implemented in hardware and do not introduce any delay or jitter.

b) **Identify the service requested by the interrupt:** The service routine to be triggered by the interrupt is directly identified by the exception vector. Thus there is no need, and hence no overhead, for searching the interrupt event handler.

c) **Save the rest of the context and load the new context:** These operations are performed by software. The respective sequence of instructions saves the contents of registers to main memory and performs register initializations for the interrupt handler or new task. The duration and jitter of this final step depend on the code that implements this functionality.

While the timing of (a) and (b) is clearly predictable, the question for the duration and jitter of (c) still needs to be answered. This question is exactly the question that will be answered in more detail in the next section, the question how SPEAR supports the predictability and prediction of code execution times. The issues discussed in the next section therefore automatically apply to the temporal predictability of (c). There is only one main point of the successive discussion that we want to mention already here: SPEAR supports the absolutely jitter-free and temporally predictable implementation of (c).

4.3 Predictability of Code Execution Times

To allow for the exact prediction of task execution times, SPEAR provides temporal predictability at two levels. First, mechanisms of the processor have been designed to allow for an exact prediction of execution times of single instructions in a code fragment. On top of the predictable instruction timing the instruction set of the processor supports the implementation of complete programs with a constant and thus fully predictable execution time.

Note that temporal predictability of single instructions does not automatically imply that the execution times of complete programs are predictable. Specific instructions are needed to support the latter. In the following we explain how temporal predictability is achieved at the instruction level. Following this we demonstrate how SPEAR supports the single-path programming model [8] and thus allows programmers to develop entire programs with a single, invariable execution time.

4.3.1 Temporal Predictability at Instruction Level

Instruction set: All instructions of SPEAR are single-word instructions with deterministic execution times. This is in contrast to other processors where execution times of some instructions depend on the values of their operands (e.g., some processors implement division or multiplication instructions by algorithms with data-dependent, variable execution times).

Pipeline architecture: In order to accomplish reasonable performance and scalability we decided to implement a pipeline architecture, although it comes at the price of a more complicated timing model. The additional model complexity, however, is very low. We do not use highly sophisticated parallel pipelines and out-of-order execution, but a rather straightforward, simple pipeline instead. As a result, the provision of forwarding was sufficient to yield an absolutely hazard-free pipeline architecture – a vital prerequisite for a predictable instruction execution time.

Memory architecture: In the minimal configuration SPEAR is equipped with on-chip ROM and RAM and does not use cached memory. Consequently memory access times are fixed and there are no unpredictable delays due to different access times of cache hits and misses. The access to these on-chip memories requires one clock cycle.

If used without care, referencing the cached external memory can lead to unpredictable memory access times – memory accesses not local to a task (e.g., memory accesses of an interrupt handler preempting the task) may influence the distribution of cache hits and misses, and thus the memory access times of the task.

However, through the provision of uncached on-chip memory temporal predictability of memory accesses to selected code segments or tasks can be guaranteed.

This memory structure is only a first step in a set of options we want to implement in order to make memory access more predictable without sacrificing the performance improvement yielded by a cache. These measures alone, however, represent an independent field of research that is beyond the scope of this paper. We will therefore not discuss the aspects of the memory architecture in more detail in this work.

Documentation: The instruction timings provided in the manuals of commercial processors usually suffer from substantial shortcomings: As a result of the complex internal structure of processors the instruction timings are often very complicated (dependences on access type, memory timing etc.). Experiments performed by different research groups [1, 4] have shown that processor manuals tend to be incomplete or even incorrect. For a tool or system designer it is usually not possible to improve these models, because no detailed insight is provided into the internal function of the processor.

Due to its simple hardware architecture SPEAR has an instruction timing that is easy to describe and to model. For each instruction there exists a complete and exact timing documentation. This documentation can be easily understood and verified by the system designer.

4.3.2 Temporal Predictability at Task Level

Typical real-time code has a number of different feasible execution paths. Depending on current input data each execution of the code runs on one of the possible paths. Because the number, order, and durations of the actions performed on these distinct execution paths differ, the code has a number of possible (input-data dependent) execution times. This variability in the temporal behavior prohibits the exact prediction of the execution time of the code.

To avoid this problem in predicting execution times we have developed the single-path programming paradigm that allows programmers to write programs with completely predictable temporal behavior [8]. The SPEAR processor supports the single-path paradigm, thus making it possible to exactly and easily predict the execution time of code running on this processor. In the following we briefly summarize the single-path programming paradigm and discuss how a number of mechanisms of SPEAR support it.

Single-Path Programming

The central idea of the single-path approach is that programmers write programs that always execute on the same execution path and whose behavior is independent of input data. Such a behavior leads to predictable code execution times and makes worst-case execution-time analysis (WCET analysis) trivial: First, as there is only one execution path, this path can be determined by observing the path on which the code executes with *any* input data. Second, there is no need for building and using complex and accurate hardware timing models for calculating the WCET of the code. The WCET of code can be simply determined by executing the code and logging its execution time.

Single-path programming yields code with a single execution path. This is achieved by removing all input-data dependent branching operations from the code. As input-data dependent branching decisions occur in both alternatives and loops, both types of constructs are translated into deterministic code that uses predicated execution (see [8] for details on the translation).

Using the conditional `move` instruction, the code of the alternative in Figure 4(a) is translated into the sequential piece of pseudo machine code shown in Figure 4(b) (the code alternatives are assumed to be free from side effects):

```
                         r1 = expr1;
if cond                  r2 = expr2;
then rr = expr1;         test cond;
else rr =                movt rr, r1;
expr2;                   movf rr, r2;
```

(a) traditional (b) single-path

Figure 4. Realization of alternatives

Executing both paths of an alternative may seem like a waste of performance at the first glance. However, relating this overhead to the significant overestimation involved in the current practice of worst-case execution-time analysis gives a different picture: A moderate price remains to be paid for a substantial simplification of the analysis process yielding guaranteed and exact (jitter-free) results.

Processor Support for Single-Path Programming

The single-path programming approach forms the basis for developing programs with predictable execution times. To obtain the deterministic execution times we aim to achieve by the single-path approach, the processor must provide all operations required to implement the single-path paradigm. It can be shown that the alternatives and loops prevalent in conventional code can be translated into predictable code sequences in the above sense, if a number of predicated operations – at least a conditional `mov` – are available [8]. While this formal transformation serves as an interesting feasibility proof, it will certainly not yield optimal code. An incorporation of the single-path paradigm in the programming style and probably on the algorithmic level as well will be necessary to make best use of this approach, and this will presumably require a more extensive set of predicated operations. With this vision in mind we have equipped SPEAR with a reasonable choice of conditional operations and associated compare instructions. A special condition flag is reserved in the status register that is – unlike the typical zero-flag and similar ones – not influenced by arithmetic operations. Rather, a set of dedicated compare instructions is available to manipulate this flag. A number of instructions (move, basic arithmetic operations, etc.) have the option of being executed only when the condition bit is at a specified truth value. If the condition bit does not match the specified truth value, SPEAR interprets the conditional instruction as a `nop` instruction with identical execution time.

5 Evaluation

In Section 4 we have made several claims with respect to the temporal predictability of SPEAR. In order to validate these claims we have conducted several experiments. We will present two of these experiments in the following.

5.1 Assessment of the Interrupt Synchronization Jitter

We used a set of 80 different instructions that were executed by SPEAR in an infinite loop. This set covers the complete instruction set of SPEAR, as we know from the design that instruction parameters like register numbers and immediate operands have no influence on instruction timing.

During the execution of this loop we applied a series of interrupts and measured the interrupt synchronization period. To make sure that the occurrence of interrupts was absolutely uncorrelated with the operation of SPEAR we generated repetitive interrupt pulses by means of a free running external function generator. This ensured that every instruction in the loop was equally likely to be suspended by the interrupt. The uncorrelated operation of the interrupt generator further guaranteed that the interrupt occurrence (i.e., signal edge) was equally distributed over the system clock period. For each interrupt we measured the synchronization period, i.e., the time interval between interrupt event (rising edge of signal INT) and the clock edge that loads the interrupt vector into the PC. The logic analyzer we used for this purpose was capable of generating a statistics (min, max, avg) over the results. As outlined in Section 4.1 we expected the synchronization period to show a minimum value of 2 and a maximum value of 3 clock cycles, which yields 67 and 100ns at 30MHz.

The results were very encouraging: Over a series of more than 100.000 measurements we observed a minimum of 71ns and a maximum of 107ns. This value is higher than expected, because the chip-internal signal path for the interrupt event (which we probe at the pin) is about 5ns slower than the clock path of the internal PLL. If we further take the 8ns resolution of the logic analyzer into account the results perfectly meet our expectations. This proves that SPEAR has reached one ambitious goal: The processor is indeed able to start the interrupt service routine within as little as two clock cycles plus a jitter of one cycle.

5.2 Evaluation of the Temporal Predictability of Code

To demonstrate how the SPEAR processor supports the temporal predictability at the task level, we implemented both a traditional and a single-path variant of a sample application task. The variants were run on the prototype processor platform with a number of representative input data sets. The execution times of all executions were measured and recorded on the logic analyzer to compare the timing of the two different software solutions.

The two sample tasks implement two different variants of *Binary Search*, i.e., they look up a key in a sorted array by using binary search. The traditional variant terminates either as soon as it encounters an array element that matches the key or if the search ends up with an empty subarray. In contrast, the WCET-oriented solution always executes $\lceil \mathrm{ld}\ (N+1) \rceil$ iterations given an array of N elements.

The array used in the experiments had size 16. To guarantee that we could observe all possible execution times, including the WCETs, both algorithms were tested with all possible input-data scenarios, i.e., 33 different data sets. 16 cases looked up the 16 different values stored in the array. 15 cases searched for values between each pair of array values and two cases looked up both a value smaller respectively larger than any of the stored values.

Interval		Traditional	One Path
0 ns	590 ns	3.03%	0 %
590 ns	1040 ns	$0.\%$	0 %
1040 ns	1490 ns	6.06%	0 %
1490 ns	1940 ns	12.12%	0 %
1940 ns	2390 ns	27.27%	0 %
2390 ns	2840 ns	3.03%	0 %
2840 ns	3290 ns	45.45%	100 %
3290 ns	3700 ns	6.06%	0 %

Figure 5. Execution times of Binary Search

Figure 5 shows the execution-time distributions of the two binary-search implementations as recorded by the logic analyzer. The execution times of the traditional implementation span over a wide range, between $0.46\mu s$ and $3.63\mu s$. In contrast, the execution times of the single-path implementation are constantly $3.11\mu s$. (The small variations in the range of $8ns$, i.e., much smaller than the duration of a CPU-clock cycle, in the observed execution-time values are again due to the limited resolution of the measurement setup). The experiments thus demonstrate that the processor provides full support for the single-path paradigm and thus, in turn, supports the cycle accurate prediction of (worst-case) execution times for real-time tasks.

It seems worth mentioning that in addition to being constant, the execution time (and hence WCET) of the single-path implementation is about $500ns$ shorter than the WCET of the traditional implementation. This means that besides being temporally predictable the single-path solution to a programming problem may also show a better worst-case performance than the traditional one. Note however that single-path programming does not automatically yield code that shows a good worst-case performance. To obtain the latter the selection of adequate algorithms is crucial, see [7].

6 Conclusion

The SPEAR processor presented in this paper combines several features that are of special importance in a real-time application. The design of the architecture minimizes response time and in particular response-time jitter. The simple structure of SPEAR facilitates accurate modeling of the timing of instruction execution, interrupt recognition and context switching. Thus, the strength of SPEAR lies in the careful combination of the chosen mechanisms to an integrated concept that provides temporal predictability to the maximum possible extent. Furthermore, this concept is extended to the application level in a consequent manner. The proposed single-path programming paradigm reduces the complex problem of worst-case execution-time estimation to a simple measurement task.

The experiments we performed so far validated our claims and are very encouraging.In our future work, the SPEAR implementation will serve as a platform to further investigate the benefits and tradeoffs of the single-path programming approach, to develop our concept towards higher performance architectures, and to compare different memory architectures.

References

[1] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems*, 18(2/3):249–274, May 2000.

[2] M. Delvai. Spear Handbook. Technical report, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[3] M. Delvai, W. Huber, B. Rahbaran, and A. Steininger. SPEAR – Design-Entscheidungen für den ”Scalable Processor for Embedded Applications in Real-time Environments”. In *Austrochip 2001, Tagungsband*, pages 25–32, Oct. 2001.

[4] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21st IEEE Real-Time Systems Symposium*, pages 163–174, Dec. 2000.

[5] R. Isermann. *Digitale Regelsysteme*. Springer-Verlag, 2nd edition, 1987.

[6] H. Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[7] P. Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

[8] P. Puschner and A. Burns. Writing Temporally Predictable Code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.

[9] J. A. Stankovic. Misconceptions about Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10):10–19, Oct. 1988.