Abstract—In this paper we explore a hierarchical memory architecture that simplifies the WCET prediction of tasks. Instead of using cache memories for speeding up code execution, we propose to use hierarchical memories that are similar to scratchpad memories. These memories are filled by explicit prefetch operations that are executed in synchrony with program execution. The instructions respectively the data that control the timing and the content to be loaded by these memory-fill operations are computed at code-generation time. The paper describes the overall system and memory architecture, and design choices for explicitly controlled time-predictable hierarchical memory architectures.

Keywords—timing analysis; time predictability; memory hierarchies;

I. INTRODUCTION

In ultra-dependable real-time systems it is necessary to estimate the Worst-Case Execution Time (WCET) and the response times of application tasks, in order to predictably meet the deadlines imposed on the embedded computer system. In such systems, the presence of cache memories makes the timing prediction of the system significantly more difficult, because the impact of task preemptions on the contents of the cache memory and as a consequence on the timing of memory accesses can only be vaguely modeled, thus yielding pessimistic WCET and completion-time estimates for the tasks. Nevertheless, an increasing number of systems consider cache memory as an inherent part of their memory systems to bridge the gap between CPU and main memory.

During the last two decades, lots of efforts have been spent on finding accurate techniques to calculate tighter bounds of Worst-Case Execution Times. Most of the methods used for WCET estimation are measurement-based or static analyses approaches [1]. Neither today’s static-analysis nor measurement based methods are however capable of producing WCET estimates for task systems running on contemporary architectures (i.e., architectures with hierarchical memory) that are both safe and tight [2]. This is a good moment to start thinking on designing a new memory-system solution or, alternatively, to use scratchpad memories instead of caches.

In this paper we present a model for a hierarchical memory-system architecture that allows us to build ultra-dependable real-time systems whose timing is stable and time-predictable, and provides the short memory access times that are needed to run tasks at highest performance. Instead of relying on cache memories to speed-up the accesses to instructions and data we use a memory similar to a scratchpads to bridge the gap between the main memory and the processor. These prefetch memories are filled by explicit load operations (and written back by explicit writes) that are executed in synchrony with task execution. All instructions respectively the data that control the timing and the content to be loaded and written back are computed at code-generation time, which leaves no room for dynamic decisions in the operation of the memory system.

This work extends our previous work on constructing time-predictable real-time systems [3]. In [3], we proposed a very restrictive software-execution model in which all control decisions and execution contraints between tasks (task scheduling, synchronization and mutual-exclusion constraints, and even control decisions within tasks) are resolved at the time the system is constructed, i.e., all actions in the system are planned such that dynamic conflict resolution, synchronization or run-time decisions are avoided. Here, we extend this principle of explicit, pre-computed control to the hierarchical memory architecture. We will see that the proposed memory-system architecture benefits from the rigid execution model.

Section II presents our assumptions about the overall system architecture and the software-execution model for which the proposed memory architecture has been conceived: precomputed time-triggered task activation, the simple-task model, and single-path code generation. Section III explains why the use of traditional hierarchical cache-memory systems complicates both WCET analysis and the overall timing analysis and introduces the principal ideas behind the proposed hierarchical memory-system architecture. As this work is in an early stage and there are still many design
decisions to be taken, Section IV explores different options and considerations for the design and implementation of the prefetch-memory architecture. Finally, we discuss related work (Section V) and conclude the paper (Section VI).

II. THE OVERALL SYSTEM

The timing of an embedded computer system depends on both, the software running on the computer and the properties of the hardware executing the software [4]. Further, in order to build a time-predictable computer system, the interface of the computer system must not allow for irregularities in the communication pattern as well. In this section, we describe our assumptions about the interface. We further list the hardware and software features we use to make an embedded computer system time-predictable.

A. The Safety-Critical Subsystem Interface

To provide the regular communication pattern that supports temporal predictability our computer system communicates with its environment via a time-triggered state message interface. This model and the terminology we will use to describe it have been adopted from the DECOS integrated architecture [5].

In DECOS, a component consists of two subsystems, the safety-critical subsystem on which all safety-critical tasks are executed and the non safety-critical subsystem that provides all other, non-critical services. The two subsystems communicate with the rest of the distributed computer system via so-called connector units. Connector units implement the architectural services of the distributed architecture: the predictable transportation of messages, the fault-tolerant clock synchronization, fault isolation, and the consistent diagnosis of node failures [5].

For the realization of highly dependable applications we concentrate on the safety-critical subsystem of a component (see Figure 1) and its memory architecture. This subsystem is required to be highly reliable and time predictable. The application computer of this subsystem communicates with the rest of the distributed computer system via so-called connector units. Connector units implement the architectural services of the distributed architecture: the predictable transportation of messages, the fault-tolerant clock synchronization, fault isolation, and the consistent diagnosis of node failures [5].

For the realization of highly dependable applications we concentrate on the safety-critical subsystem of a component (see Figure 1) and its memory architecture. This subsystem is required to be highly reliable and time predictable. The application computer of this subsystem communicates with the rest of the distributed computer system via so-called connector units. Connector units implement the architectural services of the distributed architecture: the predictable transportation of messages, the fault-tolerant clock synchronization, fault isolation, and the consistent diagnosis of node failures [5].

For the realization of highly dependable applications we concentrate on the safety-critical subsystem of a component (see Figure 1) and its memory architecture. This subsystem is required to be highly reliable and time predictable. The application computer of this subsystem communicates with the rest of the distributed computer system via so-called connector units. Connector units implement the Architectural services of the distributed architecture: the predictable transportation of messages, the fault-tolerant clock synchronization, fault isolation, and the consistent diagnosis of node failures [5].

For the realization of highly dependable applications we concentrate on the safety-critical subsystem of a component (see Figure 1) and its memory architecture. This subsystem is required to be highly reliable and time predictable. The application computer of this subsystem communicates with the rest of the distributed computer system via so-called connector units. Connector units implement the Architectural services of the distributed architecture: the predictable transportation of messages, the fault-tolerant clock synchronization, fault isolation, and the consistent diagnosis of node failures [5].

B. CPU Architecture

Although a central idea of our solution is to obtain time predictability by constructing software that has an invariable control flow (see below), we have to impose a few requirements on the CPU as well. To support the time-predictable execution of software, the CPU has to fulfill the following properties:

• Instructions are implemented such that their execution times are constant, i.e., instruction timing does not depend on operand values.
• The CPU supports a conditional move instruction or a set of predicated instructions that have invariable execution times.
• All data are stored in memory regions with homogeneous access times.

C. The Software Architecture

To construct a time-predictable computer system we need to be very strict about the software structure. The proposed software architecture does not allow for any decisions in the control flow whose outcome has not already been determined before the start of the system. This property is true for both the application tasks and the operating system. Even task preemptions are implemented in a way that does not allow for any timing variation between different task invocations.

1) Task Model: The structure of all tasks follows the simple-task model (S-task model) found in [7]. Tasks never have to wait for the completion of an input/output operation and do never block. There are no statements for explicit input/output or synchronization within a task. It is assumed that the static schedule of application tasks and kernel routines ensures that all inputs for a task are available when the task starts and that outputs are ready in the output variables when the task completes. The actual data transfers for input and output are under control of the operating system and are scheduled before respectively after the task execution.

An important and unique property of our task model is that all tasks have only a single possible execution path. By translating the code of all real-time tasks into single-path code we ensure that all tasks follow the only possible, pre-determined control flow during execution and
have invariable timing. For more details about the single-path translation see Section II-C3.

2) Operating System Structure: If not properly designed, the activities of the operating system can create a lot of indeterminism in the timing of a computer system. We have therefore been very restrictive in the design of the operating system and its mechanisms (see [8] for details).

Predictability in the code execution of the operating system is achieved by two mechanisms. First, single-path coding is used wherever possible. Second, all data that are relevant for run-time decisions of the operating system are computed at compile time. These data include the predetermined times for I/O, task communication, task activation, and task switching. They are stored in static decision tables that the operating system interprets at runtime.

Task communication and I/O is implemented by simple read and write operations to specific memory locations. As these memory accesses are pre-scheduled together with the application tasks, no synchronization and no waiting is necessary at run time.

The programmable time interrupt provided by the communication system is used to synchronize the operation of the application computer with the global time base. This way we ensure that the application computer performs all activities in synchrony with its environment and the rest of the distributed computer system.

3) Deterministic Single-Path Task Execution: As all branches in the control flow of some code may cause variable timing, we translate the code of all tasks as well as the operating-system code into single-path code [9]. The code resulting from the single-path translation has only a single execution trace, hence the name single-path translation.

The strategy of the single-path translation is to remove input-data dependencies from the control flow. To achieve this, the translation replaces all input-data dependent branching operations by predicated code. It serializes the input-dependent alternatives of the code and uses predicates (instead of branches) and, if necessary, speculative execution to select the right code to be executed at runtime. All loops with an input-data dependent termination condition are transformed into loops with a constant number of iterations. The termination condition of the original loop is transformed into a condition that occurs in the body of the new loop and makes the loop body execute conditionally, thus simulating the semantics of the original loop. More information about the conversion can be found in [10].

D. Tool Support

The software structure of our architecture is very specialized. Code generation for an application therefore needs to be supported by a number of tools:

- To generate single-path code, either a special compiler or a code conversion tool that converts branching code into single-path code is needed.
- A tool for worst-case execution-time analysis returns the execution times of the tasks and the operating system routines.
- An off-line scheduler generates the tables that control all operations of the application computer. The scheduler has to resolve all precedence and mutual exclusion constraints between scheduling decisions and task execution times: the point in time at which a task is preempted (as decided by the scheduler) influences the execution time of the task — preemptions occurring at different points of program execution affect the cache

III. MEMORY SYSTEM ARCHITECTURE

So far we described the ingredients for building time-predictable applications running on computer systems with a flat memory architecture. In previous work we have studied the use of cache memories in time-predictable systems [3], and while we found that we can build fully time-predictable systems with direct-mapped cache memories, constructing off-line schedules for these systems turns out to be tedious, because of mutual dependencies between scheduling decisions and task execution times: the point in time at which a task is preempted (as decided by the scheduler) influences the execution time of the task — preemptions occurring at different points of program execution affect the cache
content, and therefore the subsequent hits and misses on memory accesses in different ways. On the other hand, if the execution time of a task changes we also have to expect different scheduling decisions, which causes a cyclic dependency between task execution times and scheduling decisions.

Besides the undesirable co-dependency between scheduling decisions and task execution times the use of caches has also another major disadvantage for WCET analysis. In order to make WCET analysis easy, one would like to compute WCETs by using divide and conquer strategies, i.e., split the program code into segments, compute the WCETs for all the segments and combine these partial results to obtain the WCET for the entire piece of code under consideration. Such a divide and conquer strategy requires that the timing of instructions and memory accesses can be determined locally, by analyzing the code of segments in isolation. In computer systems with cache, however, memory access times can hardly ever be determined locally. Often a large set of long traces describing the possible histories of memory accesses preceding an instruction have to be analyzed to determine the possible contents and timing of the memory accesses associated with that instruction. Thus divide and conquer strategies become at best pessimistic if not infeasible. This situation gets even worse when the contents of the cache memory can be invalidated by task preemptions [11].

A. Using Memory Hierarchies with Prefetching

To avoid the problems associated with the use of cache architectures we decided to build a hierarchical memory system that is based on explicit control of the content. Instead of relying on probabilistic arguments about the locality of memory accesses (and hoping that a memory accesses would result in cache hits) we build the system on code analysis and the static planning of all load and write-back operations of the hierarchical memory system. We call the small and fast memory layer that is similar to a scratchpad memory a prefetch memory, The prefetch-memory architecture fits very well into our framework for building time-predictable real-time systems:

- Planning and determining the point in time of all load and write-back operations for the prefetch memory before runtime is consistent with the pre-planning approach applied for task and communication schedules as well as the single-path code generation that produces code with a fully determined control flow.
- Tools that generate the single-path code and thus have the full knowledge about its control flow can emit the code or data that control the load and write-back operations between the prefetch memory and the main memory during code generation.
- As single-path code does not have any dynamic control-flow decisions, the sequence of instructions following a particular instruction is fully determined at any time during task execution. The control logic for the prefetch memory can benefit from this “knowledge of the future” in that it always prefetches the instructions and data that will be needed in the near future right in time to avoid misses on access to the prefetch memory.

IV. DESIGN CHOICES FOR THE PREFETCH ARCHITECTURE

Our work on the prefetch architecture is still at a very early stage. So while we have defined the basic principle of operation there are still a number of decisions about the design and the realization that need to be taken.

We propose our work to be based on hardware design, instead of using a prefetch model that uses instructions in the application code. Even if the software solution is cheaper to realize, we still consider it inappropriate because of its overhead. On the other hand, hardware solutions increase the number of transistors on the processor chip due to the hardware that is needed to detect and prefetch instructions. In the following, we describe some of the main points that need to be studied before starting with the design of a new prefetch-memory model:

- Considering the granularity of a basic block, we should find the optimal size of memory blocks in order to obtain short transfer times for memory content, thus maximizing the number of hits on accesses to the prefetch memory.
- The next point to be considered is the design of the hardware for instruction prefetching, which has to be cheap and effective. In this step we will also have to devise algorithms to realize the prefetch mechanism. The design of a prefetcher depends also on the prefetching-information source. Today, many systems are replacing caches by scratchpads. This practice can be considered as a good solution for prefetching if we have an adequate algorithm to control and run the prefetch operations.
- Partitioning the prefetch memory might also be a promising subject for exploration. While one portion of the prefetch memory is being accessed by the CPU for accesses to instructions and data, we could in parallel copy instructions and data between other partitions and the main memory by using additional hardware. At the appropriate time, the partition used by the CPU can be switched to one of the pre-loaded partitions so that at all times the CPU accesses the fast prefetch memory, in this way shadowing the loads and write-backs of the other partitions.
- If multi-tasking systems with table-driven scheduling, as used in our approach, the sequencing of tasks and task preemptions as well as the timing of task switches is clearly defined. This exact knowledge about task
sequences and timing can also be utilized when programming the hardware that controls the operations of the prefetch-memory system.

- Our strategy of preplanning the contents of the prefetch memory can also be applied to multi-core systems. Without taking any counter measures, tasks running on different cores of an MPSoC system can run into memory-access conflicts with unpredictable timing when attempting to load the same instruction (or data) from the shared main memory to their local buffer memories at the same time. In our strictly pre-planned system, such conflicts can already be resolved before run-time, by producing control code for the prefetch logic that serves the local memories of the different processor cores at different times, thus ensuring the temporal predictability of all memory accesses.

The mentioned issues will provide the starting point for our further research steps towards creating a prefetch hardware for instructions and data which guarantees predictable timing.

V. RELATED WORK

Several attempts have been made to improve the performance and predictability of real-time systems by modifying the design of their memory architecture. Kirk [12] proposes a cache-memory design that partitions the cache into segments, each of which can be accessed by a dedicated task. However, this solution guarantees cache hits only for a limited set of partitions of a program which are located in cache. If these partitions are not used frequently, cache performance drops rapidly.

Another approach is cache locking [13], which loads cache contents with some values and locks the cache to ensure that the contents will remain unchanged afterwards. The cache content can be locked entirely or partially for the whole system lifetime (static cache locking) or it can be changed at runtime (dynamic cache locking) [14].

Lee et al. [15] suggest a dual-mode instruction prefetch scheme as an alternative for using an instruction cache. The idea of this design is to improve worst-case execution time by associating a thread to each instruction block that is part of WCET. This improvement is only applicable for single-task execution. Threads are generated by the compiler and they are static during task execution. In case of parallel multi-task execution these threads are no more valid. Also, this solution gives a good performance only when block execution time takes longer than prefetching time. Furthermore, bus contention between data and instruction prefetching is not taken into consideration.

In [16], the authors demonstrate the benefit of using scratchpad instead of cache in real-time systems. When compared, scratchpads offer better performance. They are smaller and consume less energy than conventional caches of the same size. The main drawback of scratchpads is that contents has to be explicitly controled by the programmer or compiler. Industry is still missing a comprehensive algorithm for the efficient utilization of scratchpad memories.

Miller in his dissertation [17] has designed a tool called Flexicache to manage local memory as a cache. First, a programmer writes his application and then the software caching system is applied to the binary file and becomes integrated into the executable. When the application runs the Flexicache system automatically begins managing the local memory as a cache. This solution increases the overhead compared to using traditional cache and also the inserted code makes debugging very hard for the user: when looking for bugs the user has to determine whether the problem resides in his/her own code or in the code generated by the tool. Today, most of the new proposed predictable architectures are based on scratchpad memory [18], [19].

Schoeberl [20] has made an attempt to organize a cache in a new way which is called method cache, as the full code of methods is completely stored in cache. All instructions except call and return instructions are guaranteed to be cache hits. Compared with conventional cache this solution increases the overhead, it is predictable only for a JVM, and the cache has to be large enough to hold entire methods.

An interesting solution for predictable prefetching is given in [12]. The authors had modified the Next-N-Line prefetching policy by simply adding a few hardware components that adapt the new policy even for loops in the code. However, when a loop branch is encountered, the loop-based method prefetches N lines of instructions from the beginning of the loop, and not the next N lines after the loop.

VI. CONCLUSION

Most of the developments in computer industry focus on optimizations that decrease the average-case execution times of programs. As a part of these developments, caches and pipelines have been conceived, which are nowadays the main source of temporal unpredictability in real-time systems. Within this paper we do not aim to restrict the cache behavior, but to find a solution that reduces cache misses to 0% and thus yields a fully time-predictable memory subsystem as needed for the construction of highly reliable hard real-time systems.

We propose a hierarchical memory system, called prefetch memory, in which all instruction and data transfer operations between the different memory levels are explicitly controlled. The code and data for this memory control are computed at system design time. Together with the other mechanisms that obviate dynamic influences on application timing – the temporal-firewall interface, pre-planned time-triggered task scheduling, invariable hardware-instruction execution times, and the single-path code architecture – the use of the prefetch memory allows us to build real-time systems whose timing is invariant and both fully and safely predictable.
So far, we have conceived the central concepts for this new memory architecture. In the next steps we will explore and evaluate various options for its realization.

REFERENCES


