Time-Predictable Virtual Memory

Wolfgang Puffitsch and Martin Schoeberl
Department of Applied Mathematics and Computer Science, Technical University of Denmark
wopu@dtu.dk, masca@dtu.dk

***Abstract*—Virtual memory is an important feature of modern computer architectures. For hard real-time systems, memory protection is a particularly interesting feature of virtual memory. However, current memory management units are not designed for time-predictability and therefore cannot be used in such systems. This paper investigates the requirements on virtual memory from the perspective of hard real-time systems and presents the design of a time-predictable memory management unit. Our evaluation shows that the proposed design can be implemented efficiently. The design allows address translation and address range checking in constant time of two clock cycles on a cache miss. This constant time is in strong contrast to the possible cost of a miss in a translation look-aside buffer in traditional virtual memory organizations. Compared to a platform without a memory management unit, these two additional clock cycles per cache miss introduce only a small performance overhead.**

I. Introduction

Virtual memory is an important concept in general-purpose computer architectures. On the one hand, it enables the extension of the address space beyond the size of the physical memory. On the other hand, virtual memory enables memory protection, i.e., the memory of a process is protected from memory accesses from other processes. The latter feature is especially important in ecosystems where not all components can be fully trusted. Memory protection keeps faulty processes from crashing the system by writing to arbitrary locations and malicious processes from violating privacy by reading from arbitrary locations.

In hard real-time systems, failures may lead to catastrophic consequences such as the loss of human life [**?**]. Such systems are therefore developed to the highest standards and must pass the scrutiny of certification authorities. Virtual memory provides benefits for such systems even if the software has been verified to be correct and all software modules can be trusted. Without memory protection, errors such as memory bit flips could cause a single process to crash the whole system by writing to random memory locations. Memory protection provides a fault containment mechanism that protects the overall system from the behavior of a single rogue process.

Apart from functional requirements, hard real-time systems must also fulfill timing requirements. To prove the timing correctness of an application, it is necessary to bound the worst-case execution time (WCET) of the application's tasks. The WCET bounds must be safe in the sense that actual execution time can never exceed the computed bound. However, the WCET bounds should not be overly loose to avoid over-provisioning of resources. Unfortunately, common virtual memory mechanisms such as paging are unsuited for systems that require tight WCET bounds.

This paper investigates virtual memory mechanisms that are suitable for hard real-time systems. We review existing techniques under the light of time-predictability and examine which virtual memory features hard real-time systems actually require. Our findings lead to the design of a memory management unit (MMU) that supports the required features in a time-predictable manner.

A. Contributions

The contribution of this paper is twofold. On the one hand, the paper discusses the requirements on virtual memory from the perspective of hard real-time systems. These systems are quite different from general-purpose applications, which leads to different requirements on virtual memory. In particular, hard real-time systems require predictability, whereas general-purpose systems require a high degree of flexibility and favor average-case performance over worst-case performance. We find that the requirements of hard real-time systems warrant an MMU design that deviates from the conventional MMU design for general-purpose systems.

On the other hand, this paper presents a concrete MMU design that takes into account the requirements of hard real-time systems. Our evaluation shows that the proposed MMU can be implemented efficiently and introduces only a small performance overhead, compared to a platform without MMU.

B. Organization

The remainder of this paper is organized as follows. Section II describes virtual memory and its benefits in general. Section III covers related work on virtual memory for real-time systems. Section IV discusses the requirements of virtual memory for real-time systems and describes the proposed design for a time-predictable MMU. Section V provides details on our implementation of a time-predictable MMU. Section VI evaluates the proposed design and implementation. Section VII concludes the paper.

II. Virtual Memory

Virtual memory provides an abstraction of physical memory resources [1]. The application uses virtual memory addresses, which are mapped to physical addresses by the hardware in cooperation with the operating system.

A. Properties of Virtual Memory

The mapping of virtual to physical addresses enables the realization of several beneficial features, as detailed in the following.

a) Address space beyond physical memory: With virtual memory, processes can use more data than would fit into the physical main memory. Excess data is stored in secondary storage such as a hard disk; the mapping of the corresponding virtual address then encodes that the data cannot be found in physical memory. When encountering such a virtual address, the required data are loaded into main memory. The exchange of data between the main memory and the secondary storage is handled by the operating system and transparent to the application. Historically, this feature has been one of the most important arguments in favor of virtual memory.

The cheap and large memories available today have rendered this feature somewhat less important. However, extending main memory through secondary storage is still relevant in general-purpose systems. When hundreds of processes on a general-purpose computer[1] share the main memory, providing more memory than physically available may still be necessary.

By using secondary storage, the main memory can be seen as a *cache* for the data in secondary storage. Data can be first allocated in secondary storage and just loaded in on demand. This mechanism can also be exploited to map files into memory to enable efficient file I/O. Whole blocks of a file can then be loaded into memory and written back on demand, without the need for explicit, manual buffering.

b) Memory protection: With virtual memory, each process lives in its own virtual address space and has its own mapping to physical memory. Thereby, the address spaces of different processes are isolated such that a single faulty process cannot crash the whole system. With only few additional resources, the mapping between virtual and physical addresses can also encode whether a memory location is readable, writable, or executable by a process. The operating system can take appropriate action if a process tries to violate its access rights. Additionally, memory protection is vital for ensuring that processes cannot violate privacy by reading data from other processes.

Memory protection needs at least two processor modes: a user mode and a privileged mode; the latter mode is sometimes also called "kernel" mode or "supervisor" mode. The memory translation and protection is set up in privileged mode by the operating system, as part of starting a process. The process itself executes in user mode and cannot manipulate the translation and protection settings.

c) Separate linking: In general-purpose systems, it would be impossible to decide the memory locations of all processes ahead of time. Applications are compiled independently, without knowing which other processes will run concurrently. With virtual memory, the physical locations of code and data are decided at run-time, when the operating system knows which processes are present in the system. In principle, executables could be compiled to use position-independent code, or the operating system could relocate the binary when loading it to memory. For example, μCLinux [2], a variant of Linux that does not require an MMU, uses these strategies. However, virtual memory greatly simplifies compilation and/or the loading process.

[1]At the time of this writing, 371 processes are active, consuming 12 GB of memory, on the lightly loaded laptop of one of the authors.

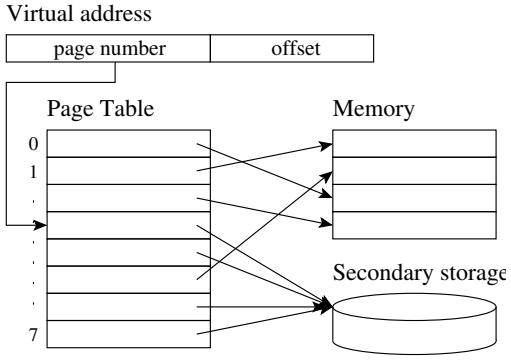

Fig. 1. Paging

B. Paging

The most popular mechanism to implement virtual memory is *paging*, which divides memory into blocks of fixed size. These blocks are called *pages* and are typically a few kilobytes in size (with 4 KB being a popular size). Paging uses a *page table* to translate between virtual and physical addresses. The upper bits of the virtual address are used to index the page table, which contains the location of the page in physical memory. The lower bits of the virtual and the physical address are the same and encode the offset of data within the page. Figure 1 illustrates this mechanism; in that figure, pages 0, 1, 2, and 5 are mapped to physical memory, whereas pages 3, 4, 6, and 7 are in secondary storage. In addition to the physical addresses of pages, the page table also contains the access permission flags for each page.

Page tables are typically too big to be kept in on-chip memory. For example, on an architecture with a 32-bit address space and 4 KB pages, the page table for the full address range would have 2^{10} entries and occupy 4 MB. Therefore, page tables reside in external memory, and a single memory access involves, in principle, two accesses to external memory: one access to access the page table for the address translation, and a second access to access the actual data. To achieve good performance, architectures that implement paging include a *translation look-aside buffer* (TLB), which caches page table entries and thus speeds up the address translation.

In addition to the benefits of virtual memory in general (i.e., address space beyond physical memory, memory protection, separate linking), paging also enables the dynamic growing (and shrinking) of memory areas such as the heap or the stack. New entries can be added to the page table as needed, and room in physical memory can be made by evicting a lesser-used page to secondary storage. As the memory is split into fixed-size pages, any page-aligned location in physical memory is suitable to allocate a page.

Even when setting aside the issue of swapping pages to secondary storage, paging increases the timing variability of

memory accesses compared to the direct use of physical memory. With paging, the timing of a memory access depends on the contents of the TLB and the cache. Both, either, or none may be hit, such that a memory accesses may exhibit different timings depending on the contents of the cache and the TLB.

C. Segmentation

An alternative to paging is *segmentation*, where the memory is divided into segments of variable length. Segmentation was used as early as 1961 in the Burrows B5000 computer, which was "perhaps the first commercial computer to provide virtual memory" [4]. Each segment occupies a contiguous area of the physical memory. A segment is described by its start in physical memory, its length, and the associated access privileges. As there are typically only few segments for each process (e.g., a code, a data, and a stack segment), the segment descriptors of the active process can be stored in on-chip memory.

Segmentation could in principle be used to provide only protection. In that case, physical memory address boundaries would be checked by the hardware without performing address translation. However, segmentation can also support virtual addresses by providing an address translation per segment.

With segmentation, virtual addresses consist of a part to identify the segment and a part for the offset within the segment. These two parts are often realized as a pair of registers, as in the Intel 8088/8086, but the segment to be used could also be encoded within the address.

If the length of a single segment is limited and too small to fit all code or data, such as 64 KB in the 8088/8086, the two-part addressing scheme may become problematic. The programmer and/or the compiler then have to specify in which segment the code or data reside, which is a complication that is not present with paging.

Another disadvantage of segmentation is that memory may become fragmented. Repeated allocation and deallocation of segments may lead to a situation where small chunks of memory are unused between the existing segments. The allocation of a new segment may then fail because none of these chunks is large enough to accommodate the allocation request, even though the total unused space would be large enough to fit the new segment. Similarly, it is impossible to enlarge a segment that is immediately followed by another segment, even if there may be unused memory elsewhere. In contrast, paging does not suffer from fragmentation and supports the growth of memory areas such as the stack or the heap.

D. Virtual Memory and Caches

A processor can use virtual or physical addresses to access its caches. When using physical addresses, the address translation has to happen before the cache can be accessed. As slowing down cache accesses would severely affect performance, fast address translation is of utmost importance when using physical addresses to access the cache.

Using virtual addresses for the cache has the advantage that address translation only needs to be done when accessing external memory on a cache miss. Therefore, the address translation can be placed outside the processor pipeline, on the path between the caches and the external memory. In contrast, using physical addresses requires tight integration with the processor pipeline to minimize the penalty for the address translation.

However, addressing the cache with virtual addresses complicates sharing data between different processes. The shared (physical) memory may be mapped to different virtual addresses in the sharing processes. This may lead to aliasing: different cache lines from different virtual addresses can correspond to the same physical memory location. If one cache line is modified, the other cache line will have a stale (wrong) value. Changing the address mapping could also lead to stale cache entries. Consequently, the cache needs to be cleared when switching between processes, which incurs high costs for context switches.

In a multiprocessor setting, using virtual addresses for the caches would also complicate the implementation of hardware cache coherence mechanisms. Coherence must be established for cache lines that refer to the same physical address. A cache coherence mechanism would either have to do a reverse address translation, or the cache has to also include information about the physical addresses of its cache lines.

Due to these issues, the use of physical addresses to access the cache prevails in modern processor architectures [1].

A hybrid approach is to use virtual addresses to index the cache, but physical addresses to decide whether an access is a cache hit or miss [1]. This technique puts the address translation in parallel to the access of the cache memory and therefore reduces the pressure on the speed of the address translation. However, this technique also limits the size of the cache because the number of bits to index the cache may not exceed the number of bits to index a page.

III. Related Work

Bennett and Audsley [5] investigate how MMUs of (at that time) modern processors can be utilized in a time-predictable way. They conclude that good predictability can be achieved through proper configuration of the MMU and careful organization of page tables. However, they also note that real-time operating systems typically avoid virtual memory altogether. A particularly interesting feature exploited by Bennett and Audsley are the Block Address Translation (BAT) registers of the PowerPC 603e architecture [6]. These registers can be used for the address translation for large contiguous memory areas. Address translation through a BAT register takes precedence over address translation via the TLB, leading to a faster and more predictable execution.

Zhou and Petrov [7] present a page table organization that aims for low memory requirements while supporting predictable page table lookups. They observe that, in embedded systems, consecutive virtual pages often correspond to consecutive physical pages. Consequently, the proposed page table organization encodes such contiguous areas in a single entry.

Puaut and Hardy [8], [9] propose an approach where pages are exchanged with secondary storage at defined points in the

program. In the initial variant of the approach [8], they model the exchange of pages with secondary storage similar to the spilling of registers onto the stack. To minimize the exchange of pages with secondary storage, they propose a graph coloring approach. A later variant of the approach [9] uses integer linear programming to minimize the worst-case costs of exchanges with secondary storage.

The approach by Puaut and Hardy is predictable in the sense that the program points at which pages are exchanged with secondary storage are known. However, they do not investigate other sources of unpredictability such as page table lookups. Furthermore, unpredictable or excessive access times for secondary storage would render the approach by Puaut and Hardy unusable.

Meenderinck et al. [12] present the design of a predictable MMU in the context of a system-on-chip platform. They achieve predictability by keeping page tables in on-chip memory and prohibiting the swapping pages of to secondary storage. The evaluated applications are relatively small, such that the limited page table size does not constitute a major restriction. However, Meenderinck et al. report a performance loss of 39% due to address translation.

Böhnert and Scholl [13] describe a virtual memory solution that takes into account the requirements of real-time systems. They use sophisticated data structures to ensure that operations like allocating or deallocating memory can be performed in constant time. Their evaluation shows that their solution achieves more predictable behavior compared to a conventional virtual memory solution. However, the MMU proposed by Böhnert and Scholl still relies on a cache that resembles a TLB for the address translation and suffers from the associated unpredictability.

All approaches described in this section take paging for granted and optimize the paging mechanism rather than proposing a different MMU design. We agree that paging is by far the most successful mechanism to implement virtual memory for general-purpose systems. However, we find that hard real-time systems have requirements that are quite different from general-purpose systems, and that these different requirements call for a different MMU design.

IV. Design

This section first analyzes the requirements on virtual memory from the perspective of hard real-time systems, and then presents the design of an MMU that fits these requirements.

A. Requirements

The following paragraphs list requirements for virtual memory in real-time systems.

a) Time-Predictability: The requirement with highest priority is predictability. In the context of hard real-time systems, virtual memory mechanisms that annul predictability are inadequate. Not using virtual memory would be a better solution than using a mechanism that might cause tasks to miss their deadlines. It should be possible to compute tight execution time bounds even when using virtual memory.

b) Memory protection: The second most important virtual memory requirement for real-time systems is memory protection. This feature ensures that processes cannot read or modify each other's memory contents, neither maliciously nor by accident. Memory protection is vital to ensure proper partitioning between different parts of a real-time system and helps to contain errors. Furthermore, memory protection catches issues such as wrongly sized memory areas. For example, without protection, the heap and the stack could grow into each other and silently corrupt each other's data. A time-predictable implementation of virtual memory must support this feature.

c) Independent linking: The feature that the binaries for different applications can be linked independently is of lesser importance. While large real-time systems may benefit from this feature, the software of other real-time systems may be small enough to be linked at once. Support for this feature is desirable, but should not come at the cost of unpredictability.

d) No address space beyond physical memory: Extending the address space beyond the size of the physical memory is not required for hard real-time systems. Quite to the contrary, swapping data to secondary storage could introduce unacceptable unpredictability and should be avoided. Support for this feature would be useless at best and harmful at worst.

e) No extension of memory areas: In hard real-time systems, the dynamic growing of memory areas is not needed. The memory consumption of hard real-time tasks must be bounded just like their execution time must be bounded. Therefore, the amount of memory needed must be known beforehand, and the required memory can be allocated before the task starts execution.

f) Memory areas may be contiguous: We can classify the memory areas of a hard real-time application into a handful of categories, such as the code, the data, or the stack. As observed in previous work [7], the memory areas of embedded real-time applications are typically mapped to contiguous regions of memory. Therefore, we think that requiring that these areas be contiguous is a reasonable constraint.

g) Caches may use virtual addresses: Both using physical addresses and using virtual addresses for caches may affect performance negatively. On the one hand, using physical addresses is likely to increase the cache access time and hence decrease performance. On the other hand, using virtual addresses requires that caches be flushed upon context switches and therefore increases preemption costs. As preemption costs are already fairly high – in particular when considering multiprocessor systems – reducing the number of preemptions is probably more effective than reducing the costs of an individual preemption. Therefore, using virtual addresses to access caches may be a viable option for a time-predictable MMU.

B. Proposed Solution

We believe that the requirements outlined above are best fulfilled by a solution that resembles segmentation rather than paging. Segmentation enables memory protection and independent linking. As the number of segments is relatively small, all relevant information can be kept on-chip, which is

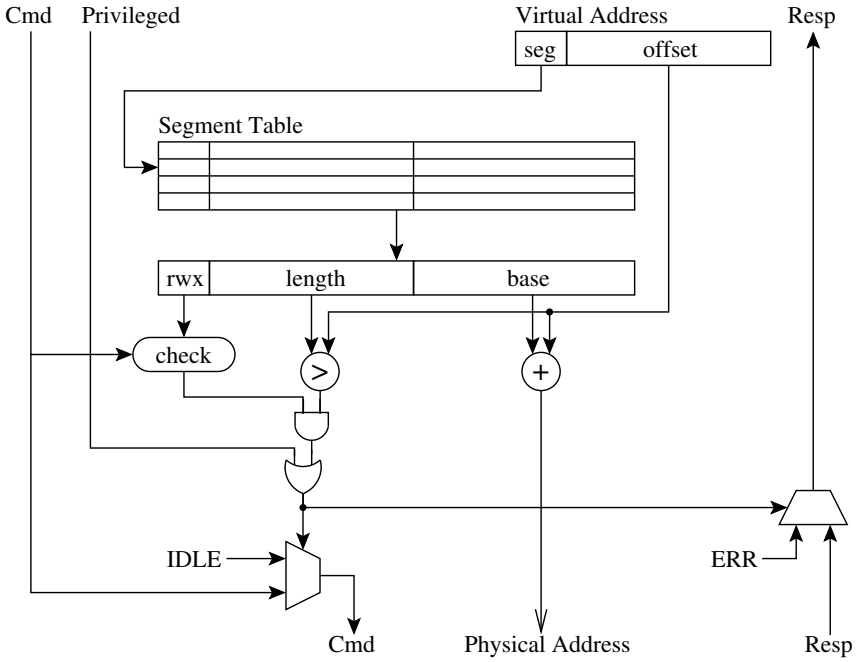

Fig. 2. Address Translation and Permission Check

favorable in terms of time-predictability. The short segment table can be considered part of the context of a process and is loaded at the process start and on a context switch to that process.

Storing all address translation information on-chip as part of a context switch is in strong contrast to the operation of TLBs that are used with paging, where TLB misses can occur during normal process execution. The TLB, a cache in its own, would need to be integrated with the cache analysis of the WCET analysis.

The strengths of paging would be simpler exchange of data with secondary storage, dynamic extension of memory areas, and memory areas that are non-contiguous in physical memory. However, these features are not required in hard real-time systems, such that the downsides of paging outweigh the benefits.

Some segmentation solutions use pairs of registers for virtual addresses and allow only rather small segments. However, such addressing schemes are not a good fit for the flat memory model of languages such as C. Therefore, we propose to encode the segment in the upper bits of the virtual address.

Figure 2 shows a simplified diagram of the envisioned MMU. The upper bits of the virtual address ("seg") are used to index a segment table. This table contains the access permissions (read/write/execute), the lengths, and the base addresses of the segments. If the respective access is permitted and the offset is within the bounds of the segment, the base address is added to the offset from the virtual address to form the physical address. If the access is not permitted or out of bounds, the issued command is suppressed (i.e., replaced with an IDLE command), and the MMU responds with an error response (i.e., the response value ERR).

To simplify system software, the permission and bounds checks are disabled when the processor is in privileged mode. In principle, it would be possible to use separate permission flags or segment lengths for execution in privileged mode. However, an application in privileged mode can modify these values at will, such that memory protection would be incomplete. We decided to disable the checks altogether rather than implementing an incomplete solution.

With regard to the number of segments, we face a trade-off between flexibility and hardware resources. Providing many segments (potentially hundreds or thousands) would lead to a solution whose flexibility could approach paging. However, such a solution would also require a substantial amount of hardware resources to keep all segmentation information on-chip. Instead, we propose to provide few segments, thus minimizing the required hardware resources. More concretely, we propose a solution with eight segments, which could be used as shown in Table I. As the segment is encoded in the upper bits of the address (rather than in other parts of instruction set), the number of segments could be easily increased if future experience proves the need to do so.

In the proposed segment usage convention shown in Table I, segment 0 is reserved for operating system code, such as code that implements system calls. Segments 1-4 are used for the different memory areas of the application. We distinguish between read-only data and writable data to catch the modification of data that is meant to be constant. Segments 5-7 are intended for segments that are shared between applications, such as code and data that are part of the C library.

TABLE I
PROPOSED SEGMENT USAGE

Segment	Usage	Permissions
0	system	`--x`
1	code	`--x`
2	read-only data	`r--`
3	writeable data/heap	`rw-`
4	stack	`rw-`
5	shared code	`--x`
6	shared read-only data	`r--`
7	shared writeable data	`rw-`

When using eight segments and 32-bit addresses, the size of a segment is limited to 512 MB. We do not expect that embedded real-time systems require larger segment sizes, such that some of the traditional inconveniences of segmentation are alleviated.

We propose the use of virtual addressing of the caches to avoid increasing the memory access times on cache hits. Address translation and permission checks are performed when the cache is filled or data is written (back) to external memory, i.e., only cache misses may suffer delays caused by the MMU. The downside of virtual addressing is that caches need to be invalidated upon context switches, increasing the costs of preemptions. However, in our opinion, worst-case analyses will benefit more from low cache access times than they would from low preemption costs. Furthermore, virtual addressing requires less tight integration with the processor pipeline, which simplifies implementation.

Issues regarding virtual addressing of caches and cache coherence protocols do not pose a problem for the platforms we envision. On the one hand, hardware cache coherence protocols are problematic with regard to time-predictability [**?**] and should be avoided in a time-predictable platform. On the other hand, hardware cache coherence protocols do not scale to platforms with many cores. For such many-core platforms, communication between cores should use the on-chip network rather than shared memory [14]. Therefore, we do not consider hardware cache coherence mechanisms in the design of our time-predictable MMU.

V. IMPLEMENTATION

We implemented the proposed MMU in the context of the time-predictable processor Patmos [15], which is part of the time-predictable T-CREST platform [16]. Among other features to aid time-predictability, Patmos implements a cache architecture that is designed for time-predictability. For example, the default configuration includes a separate cache for stack data [17], and a method cache [18], i.e., an instruction cache that caches whole methods. The details of the cache architecture had to be taken into account for the integration of the MMU. However, the MMU itself is rather generic and can be included in other architectures as well.

A. Hardware

In our implementation, the MMU is placed outside the processor core and intercepts transactions between the processor core and the memory controller. When building Patmos without virtual memory support, the MMU can be replaced by a dummy hardware module that simply forwards signals without changing them. The segment table of the MMU is exposed to the processor as an I/O device; modification of the table is only allowed when the processor is in privileged mode.

Patmos uses variants of the OCP protocol [19], [20] for internal communication and for accessing external memory. The MMU signals invalid memory accesses to the processor by responding with the value ERR, which is part of the OCP standard. The memory stage triggers an exception whenever receiving a response value ERR instead of a response value DVA (data valid). Some of the data caches required minor adaptions to properly forward ERR responses, but otherwise, the triggering of exceptions on invalid data cache accesses was straightforward. However, the instruction cache and the stack cache require a different mechanism to trigger an exception. These caches handle memory accesses internally and stall the pipeline rather than responding back to the memory stage. Therefore, we had to add signals to notify the memory stage of invalid memory accesses originating from these caches.

The OCP protocol distinguishes reads and writes, but does not include signals to distinguish reads that originate from the data cache or from the instruction cache. To enable this distinction and hence the proper handling of read and execute permissions, we added an appropriate input signal to the MMU. Conceptually, a command handled by the MMU comprises this signal and the OCP command.

The hardware implementation closely follows the organization shown in Figure 2. However, the implementation includes some additional logic to correctly handle OCP burst transactions.

In the OCP variant used between the processor core and the external memory, a slave must accept a whole burst transaction once it accepts the initial command. To keep the critical path between the processor core and the external memory reasonably short, the MMU registers its inputs before processing them. As the MMU cannot know whether the external memory will accept the transaction at the time when it registers the inputs, the MMU itself must be capable of buffering a whole transaction. This is particularly important in multicore configurations, where an individual core has to share access to external memory with other cores.

B. Compilation

Adapting the linking process to use virtual addresses was relatively straightforward. A simple linker script is sufficient to map different parts of the executable to different segments. However, we needed to adapt the compiler for Patmos to make the generated code work correctly with virtual memory.

By default, the Patmos compiler emits call instructions where the target address is given by an immediate value. In the instruction set of Patmos, this immediate value is 22 bits wide. This means that the code must fit into the first 4 MB of the address space, which should be sufficient for most embedded systems. With virtual addresses however, this would mean that

only segment 0 would be usable for code, which would be an undue restriction. In particular, having only one segment available for code would make it impossible to switch between operating system code and application code in a clean fashion. We adapted the compiler to load full 32-bit addresses into a register and use an indirect call when compiling for virtual memory.

Future work could extend the Patmos instruction set to support immediate calls with 32-bit immediate values, similar to the already present support for ALU instructions with 32-bit immediate values. An alternative extension could be the introduction of calls that are relative to the program counter. Such extensions would (at least partially) eliminate the code-size and run-time overheads of calls when compiling for virtual memory.

C. Software

To experiment with the MMU, we developed an application that downloads ELF binaries via TFTP and executes them. Due to the use of virtual memory, the segments of the downloaded binary may be placed arbitrarily in the physical address space. Read-only segments (such as code and read-only data) can be used in-place, where the binary is downloaded. Only segments that are larger than in the binary (such as the heap) or not present in the binary (such as the stack) are freshly allocated.

We currently support only statically linked binaries. Future work could also explore the use of shared segments for common code such as the C library or low-level arithmetic functions.

VI. Evaluation

To evaluate the proposed MMU design, we evaluate the hardware resources required to implement the MMU in an FPGA and the performance impact of the MMU.

A. Hardware

We synthesized Patmos with the MMU for an Altera Cyclone IV FPGA (part number EP4C115F29C7N). This FPGA is found on the Altera DE2-115 development board, the default target for Patmos. We configured Patmos with a single-issue pipeline, a 4 KB method cache with 16 entries, a 2 KB stack cache, and a 2 KB direct-mapped data cache with a write-back policy. The external memory in our setup is an asynchronous SRAM with an access time of 21 clock cycles for a 16-byte burst access. We used a clock frequency of 80 MHz; the MMU is not on the critical path and hence does not influence the maximum frequency.

Table II presents the hardware synthesis results. The table shows the resource usage of the whole Patmos core (excluding I/O devices, but including caches), the MMU, and the buffer for burst transactions inside the MMU. The figures for the whole core include the MMU, and the figures for the MMU include the burst buffer.

The results in Table II show that the MMU consumes less than a tenth of the logic cells of the processor. About a third of the logic cells of the MMU are used for the buffer that buffers whole OCP burst transactions. The fraction of registers used for the MMU is relatively larger, with a share of almost sixteen percent of the whole processor. Again, about a third of the registers is used for the OCP burst transaction buffer. The rather high share of registers is due to the segmentation table, which is too small to implement with an on-chip memory block. Using a larger number of segments would cause the hardware synthesis to implement the segment table in on-chip memory block and thus reduce the number of registers. About 500 logic cells and registers are spent on the implementation of the actual virtual memory implementation, which we consider a quite efficient implementation of an MMU.

TABLE II
Hardware Resource Usage on Altera Cyclone IV FPGA

Unit	Logic cells		Registers		Memory Bits	
Core	9374		4455		88640	
⊢ MMU	814	(8.7%)	707	(15.9%)	0	(0%)
⊢ Buffer	245	(2.6%)	237	(5.3%)	0	(0%)

The hardware of the proposed solution is similar to the hardware to implement a TLB in a traditional paging solution. We expect that a TLB implementation would require a comparable amount of hardware resources. However, the proposed solution eliminates TLB misses and page faults as part of normal operation, making it trivial to guarantee predictable behavior of the MMU.

B. Performance

When using the MMU in Patmos, performance is affected by two factors. On the one hand, the MMU introduces a latency of two clock cycles due to the pipelining registers that keep the critical path reasonably short. These two clock cycles increase the cache miss time of 21 clock cycles by 10%, which we consider a small, but predictable performance decrease. On a multiprocessor version of Patmos, with time-predictable memory arbitration [21], the worst-case cache miss time is multiplied by the number of processors, making those additional two clock cycles negligible.

On the other hand, the use of full 32-bit addresses for calls carries a performance overhead, compared to the use of immediate calls with 22-bit addresses. As these overheads do not vary at run-time, the proposed MMU may affect performance, but does not affect time-predictability.

As mentioned, the MMU increases the latency of cache misses by about 10% for the evaluated configuration. As not every instruction causes a cache miss, we can expect the overhead introduced by the MMU itself to be less than 10%. The overhead for using full 32-bit addresses for calls is harder to estimate, because it combines the effects of executing more instructions, a larger code size, and a slightly higher register pressure.

Table III shows performance measurement results for the CoreMark benchmark [22] in different setups. The Coremark benchmark consists of several sub-benchmarks, such that the results reflect the behavior over these different sub-benchmarks. The row labeled “short calls” presents result when using 22-bit

TABLE III
CoreMark Results for Different Setups, Iterations per Second

	No MMU	With MMU
Short calls	131.96	130.91
Long calls	117.39	116.12

immediate values for calls, whereas the row labeled "long calls" presents results when using full 32-bit call addresses. Two columns in Table III show results with and without the MMU present in the processor.

The results in Table III show that the performance impact of the MMU is very small, around 1%. However, using long call addresses has a noticeable impact, reducing performance by about 11%. We conclude that the MMU itself meets the goal of providing good performance while being time-predictable. However, future revisions of Patmos should extend the instruction set to reduce the performance penalty for using 32-bit call addresses.

VII. Conclusion

Hard real-time systems can benefit from the use of virtual memory, in particular from the memory protection provided by virtual memory. This paper presented the design and implementation of a time-predictable MMU.

We analyzed the requirements of hard real-time systems on virtual memory and found that these requirements are rather different from the requirements of general-purpose systems. In particular, hard real-time systems require predictability, whereas general-purpose solutions favor flexibility and average-case performance over predictable worst-case performance. Traditional virtual memory solutions are therefore not a good fit for hard real-time systems.

We presented the design of a time-predictable MMU that takes into account the requirements of hard real-time systems. Furthermore, we implemented the proposed solution in the time-predictable processor Patmos. Our evaluation showed that the proposed MMU can be implemented efficiently in hardware, and that it introduces only a small performance overhead.

Acknowledgements

The work presented in this paper was funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP,[2] contract no. 12-127600.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[2] D. J. Dionne and M. Durrant, "μClinux embedded Linux/microcontroller project," Arcturus Networks Inc., 1998. [Online]. Available: http://www.uclinux.org

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014.

[4] A. J. Mayer, "The architecture of the Burroughs B5000: 20 years later and still ahead of the times?" *ACM SIGARCH Computer Architecture News*, vol. 10, no. 4, pp. 3–10, 1982.

[5] M. Bennett and N. C. Audsley, "Predictable and efficient virtual addressing for safety-critical real-time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2001, pp. 183–190.

[6] *PowerPC 603e^{TM}RISC Microprocessor Technical Summary*, Motorola/IBM, 1996.

[7] X. Zhou and P. Petrov, "The interval page table: virtual memory support in real-time and memory-constrained embedded systems," in *Proceedings of the 20th annual conference on Integrated circuits and systems design*. ACM, 2007, pp. 294–299.

[8] I. Puaut and D. Hardy, "Predictable paging in real-time systems: A compiler approach," in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2007, pp. 169–178.

[9] D. Hardy and I. Puaut, "Predictable code and data paging for real time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2008, pp. 266–275.

[10] J. Whitham and N. Audsley, "Studying the applicability of the scratchpad memory management unit," in *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 205–214. [Online]. Available: http://dx.doi.org.globalproxy.cvt.dk/10.1109/RTAS.2010.21

[11] ——, "Implementing time-predictable load and store operations," in *Proceedings of the International Conference on Embedded Software (EMSOFT 2009)*, 2009.

[12] C. Meenderinck, A. Molnos, and K. Goossens, "Composable virtual memory for an embedded SoC," in *Digital System Design (DSD), 2012 15th Euromicro Conference on*, Sept 2012, pp. 766–773.

[13] M. Bohnert and C. Scholl, "A dynamic virtual memory management under real-time constraints," in *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, Aug 2014, pp. 1–10.

[14] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. T. Müller, K. Goossens, and J. Sparsø, "Argo: A real-time network-on-chip architecture with an efficient GALS implementation," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, 2015. [Online]. Available: http://www.jopdesign.com/doc/argo-jnl.pdf

[15] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, and C. W. Probst, "Towards a time-predictable dual-issue microprocessor: The Patmos approach," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems, PPES '11*, ser. OpenAccess Series in Informatics (OASIcs), vol. 18. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, pp. 11–21.

[16] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, J. Sparsø, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[17] S. Abbaspour, F. Brandner, and M. Schoeberl, "A time-predictable stack cache," in *Proceedings of the 9th Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2013.

[18] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl, "A method cache for Patmos," in *Proceedings of the 17th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2014)*. Reno, Nevada, USA: IEEE, June 2014, pp. 100–108.

[19] Accellera Systems Initiative, "Open Core Protocol specification, release 3.0," 2013. [Online]. Available: http://www.accellera.org/downloads/standards/ocp/

[20] M. Schoeberl, F. Brandner, S. Hepp, W. Puffitsch, and D. Prokesch, *Patmos Reference Handbook*, 2014. [Online]. Available: http://patmos.compute.dtu.dk/patmos_handbook.pdf

[21] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø, "A time-predictable memory network-on-chip," in *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, Madrid, Spain, July 2014, pp. 53–62.

[22] Embedded Microprocessor Benchmark Consortium, "Coremark," 2009. [Online]. Available: http://www.eembc.org/coremark

[2] http://rtemp.compute.dtu.dk