Why current Memory Management Units are not suited for Automotive ECUs

Jörn Schneider
Trier University of Applied Sciences
Schneidershof, 54293 Trier
j.schneider@fh-trier.de

Abstract: A major trend in automotive industry is to enrich driver and passenger experience with an increasing amount of consumer electronics and car-2-x functionality. A close interaction between this added functionality and the classical automotive domains allows for innovations that are valuable to the end customer and cannot be outplayed easily by devices with a pure consumer electronic origin. Innovations of this class require a tight coupling, for instance by executing programs from both worlds on the same microprocessor. The latter introduces many challenges, especially regarding reliability, security and safety of such systems. A unified memory management fulfilling the requirements of the consumer electronics and automotive application could help to address these issues and is a challenge by itself. This paper shows that the prevailing implementation scheme for memory management units (MMUs) is not suited for the needs of such systems and points out a solution direction.

1 Introduction

A common foundation of reliable, safe and secure systems are hardware based mechanisms to protect memory regions. In standard IT and consumer electronics devices memory management units provide this service. While in classic automotive domains still only memory protection units are available for this purpose.

Innovations going beyond the state-of-the-art in driver experience require a close interaction between consumer electronics or car-2-x functionality and the classical automotive domains such as engine control, chassis systems, body control, or occupant safety. Only products and services based on a tight integration cannot be outplayed easily by devices with a pure consumer electronic origin. A promising way to achieve this is to execute consumer electronics/car-2-x and classical automotive control programs on the same microprocessor while sharing common data and peripherals.

Executing automotive control and consumer electronics/car-2-x software on the same microprocessor introduces many challenges, especially regarding reliability, security and safety of such systems. A unified memory management fulfilling the requirements of the consumer electronics/car-2-x and automotive domain could help to address these issues and is a challenge by itself.

Developers of standard IT systems or of smartphone software are used to the benefits
delivered by memory management units. In the automotive domain the situation is quite different for several reasons. A primary issue here are costs per Electronic Control Unit (ECU). In this high volume market with small revenue margins the extra costs for increased die size are a serious issue. Nevertheless, if memory management units would have been really required in automotive systems, they would have been there. This was simply not the case in classic automotive domains.

There are two hard reasons a computer scientist can put forward in favor of MMUs:

**Memory Protection** In a system without memory protection one erroneous piece of code might cause a complete system failure. MMUs allow to protect each process against another.

**Management of Secondary Storage (swapping)** Transferring code or data between main memory and secondary storage manually is an cumbersome and error-prone activity. Thanks to memory management units, the times where a lot of development effort went in to partitioning of programs into overlays and managing these without operating system support, are past for almost all systems.

None of these two reasons was an issue that required MMUs for classic automotive systems so far. Before the AUTOSAR Operating System specification included memory protection as an optional feature, ECUs where usually developed as closed systems without memory protection. Nowadays memory protection is used in some cases, e.g. due to the integration of applications with different ASILs according to ISO 26262 and this is typically achieved with memory protection units (MPUs).\(^1\) Thereby, memory protection is realized with less die-space-consuming and cost-intensive hardware.

Management of secondary storage is still not necessary, because ECUs use NOR-Flash technology, which allows for execution in place, i.e. the code is executed out of the secondary memory immediately. Thus RAM is required for nonconstant data only.

Systems executing consumer electronics/car-2-x applications and safety-relevant control applications with hard real-time demands on the same hardware, fall into the class of so-called mixed-criticality systems, that exhibit diverging requirements regarding safety and real-time behavior. Within the paper the term hybrid system is used to denote such systems. The state-of-the practice solution is to use different processors for different parts, e.g. an application CPU running a general purpose or soft real-time operating system such as Linux or QNX, and a second CPU as network controller running AUTOSAR OS that handles the communication with other ECUs via CAN or FlexRay bus systems. A different, widely propagated, approach is to execute the involved operating systems on separate virtual machines [HH08, GWLC12]. However, as observed by Gu et al. the overhead of traditional virtualization approaches is usually not acceptable in automotive ECUs [GWLC12].

The virtualization based approach uses a different class of hardware than ECUs. It includes MMUs and many other features foreign to classic automotive systems. Leaner approaches are conceivable that execute consumer electronics/car-2-x applications and control applications on one CPU (with or without multiple processor cores) without full-blown vir-

---

\(^1\)Memory Protection Units can monitor memory references to ensure that a piece of code accesses only the allowed memory parts and only in the permitted mode (e.g. read-only).
virtualization. Such systems would allow to use shared memory as communication media without the usual time-triggered message scheme as it is already established in avionics systems [ari97, CRA+09] and proposed for automotive systems by different vendors. Using shared memory without indirection has the potential to eliminate the high latencies and resource demands of the existing approaches but requires new concepts to maintain real-time guarantees. Some insights on this issue can be found in [Sch12]. Both, the virtualization based solution and the immediate usage of shared memory assume one underlying approach to memory management.

The following Section presents the major requirements on memory management units in future automotive systems from two perspectives. The first viewpoint is that of consumer electronics applications and the second of classical automotive applications. Section 3 gives a brief description of the prevailing MMU approach (i.e. page based MMUs) in standard IT systems. It follows a detailed discussion on dimensioning the page size in order to meet the requirements of future automotive systems in Section 4. A major part of this discussion are quantitative considerations regarding memory efficiency by examining number experiments with exemplary task sets. Finally Section 5 summarizes the made observations and points towards an alternative approach to MMUs that is much closer to the requirements of future automotive systems than the prevailing approach to memory management in standard IT.

2 Requirements on MMUs in automotive systems

This Section describes some central requirements on MMUs from the two major perspectives of future hybrid systems in automobiles: consumer electronics and classical automotive.

2.1 Consumer Electronics Perspective

An MMU in consumer electronics devices fulfills three essential needs:

Logical Address Space The key innovation of MMUs is the separation of the two concerns addressing in a program and location in physical storage of code and data. This achieves the degree of freedom that allows the programmer to abstract from the details of memory management. Additionally, separating these concerns, delivers mechanisms to relocate code and data, and to protect the address space between processes.

The prominent underlying feature of an MMU based system is that each process owns its private address space. This is a logical address space since processes access code and data as if these addresses would reflect the actual physical position in memory although this is not necessarily the case. The mapping of logical addresses to physical addresses is handled by the MMU and the operating system. The application developer considers the level of logical addresses only and abstracts from physical memory layout.
Memory Protection Because every process has its private address space and the MMU divides this address space in different areas (e. g. memory pages of equal size), it is straightforward to assign access rights to areas according to protection needs. Some areas might for instance be marked as read-only or execute-only. A violation of the rights causes the MMU to signal an interrupt to the CPU thereby allowing the operating system to handle the situation. This mechanism is one of the essential foundations for establishing secure systems.

Management of secondary storage Consumer electronics devices typically use technologies for secondary storage that do not offer the possibility to execute code immediately from secondary storage, e. g. NAND flash memory, or disk drives. All programs have to be transferred to main memory (e. g. DRAM or SRAM) to be executed. Usually, there is not enough main memory to hold all programs to be executed completely at all the time. Only pieces of different programs might be present in main memory at a given time.

Before MMUs where available a software developer had to manually divide each program into chunks (overlays) that need not be in memory at the same time, and to write code to manage loading of overlays. In current consumer electronics devices the MMU handles these issues together with the operating system.

Note that, the process of fetching needed memory contents from secondary storage into main memory is called swapping in standard IT systems. This includes writing back of changed memory contents to secondary storage if needed. However, this is rarely supported in embedded devices, because disk drives are usually not available. Therefore, only memory content that is unaltered, e. g. code, is replaced by newly fetched memory contents, if no free space is left in main memory.

2.2 Automotive Perspective

The state-of-the-art in electronic control units of the classical automotive domains is as follows:

No Logical Address Space Only physical addresses are used, there are no hardware supported memory abstractions in place.

Memory Protection by MPUs Memory can be protected by memory protection units (MPUs). Each access to a physical memory address is monitored by the MPU that verifies whether the access is in line with the access rights of the executed task. An interrupt is raised by the MPU if this is not the case and the operating system handles the situation as specified by the application developer.

Execution in Place The secondary storage of ECUs uses NOR-Flash technology. This allows to execute code immediately in secondary storage without transferring it to the main memory. Therefore, no hardware support to transfer code into RAM is necessary.

In future systems that integrate applications from both worlds, it is very likely that the described situation changes completely. A logical address space with hardware supported memory abstraction could be very beneficial for automotive systems also, e. g. because the
integration efforts could be reduced and software updates in the field would be easier. Even when MMUs are introduced due to other reasons, it is very likely that they will eventually be used for this purpose.

Memory protection becomes even more important in hybrid systems and is in principle achievable with an MMU, although there are some important differences as described later.

It can be expected for future hybrid systems that code and data are no longer stored in NOR-flash memory but at least parts of it end up in NAND-flash, which is available for the consumer electronics side anyway. If this is the case, secondary storage management by an MMU becomes essential for the automotive part also.

If MMUs are to be used by the automotive control part of hybrid systems three new requirements have to be considered:

**Small protection blocks** In a safety relevant system memory protection has to offer more than just the isolation between address spaces of different processes. Consider an example where an application shall be allowed to access the control registers of peripheral unit A in read-only mode and those of peripheral unit B not at all. In case of memory-mapped IO a memory protection unit could be used to achieve this, even if the control register blocks have adjacent addresses. An alternative solution is an underlying software layer (e.g. the operating system) that enforces protection with an arbitrarily small block size. However, this induces overhead and requires a larger trusted software base in case of a hybrid system.

**High memory efficiency** The market for classical automotive electronic control units is very cost sensitive. Even cost differences in the order of Cents per ECU can make a difference in such a mass volume market. As long as this is the case, the usage of memory has to be highly efficient in order to prevent higher hardware costs.

**Real-time behavior** Classical automotive software has to fulfill hard real-time constraints very often, e.g. to guarantee stability of control loop algorithms, or because of physical conditions, or safety requirements. New technologies often come with two problems regarding real-time requirements. First they often introduce timing deviations, i.e. depending on the execution history the same code might be executed faster or slower, even if the same data values are given. Consider for instance the case of an instruction cache. Second it might be very hard or even impossible to predict in a sufficiently precise way the worst-case execution time (WCET) for a given program, due to the unpredictable nature of a concrete mechanism causing timing deviations. When introducing a new technology such as MMUs the goal should be to add no further sources of timing deviations, if this is not completely possible the design of the technology should at least allow for a precise prediction of WCETs.
3 Prevailing MMU solutions

Standard IT systems such as personal computers or servers nowadays use page based MMU approaches. The logical address space is organized in pages of equal size, e.g. 4 kByte, or 1 MByte. Each process sees a logical address space of the size of the addressable area of the processor. The number of usable address bits according to the processor architecture, e.g. 32, or 64, determines the size of the address space, which is usually quite different to the actually available memory.

Whenever a process accesses memory it uses a logical address that is translated into a physical memory address before the actual memory cell can be accessed. The operating system manages the assignment of logical to physical addresses by keeping one page table per process that contains for each memory page the base address of the assigned physical page frame.

The size of the page table is determined by the size of the address space and the page size. In the DEC PDP-11, a system with 16 address bits and pages of 8 kByte was used, the page table had 8 entries which were kept in special registers [AS98]. In modern computers page tables are far too large to be kept in registers. The ARM Cortex-A8 has 32 address bits and a variable page size (4 KB, 16 KB, 64 KB, 1 MB, and 16 MB) [JLH12]. With pages of 4 KB the number of pages to capture in the page table is more than a million. A page table of this size cannot be kept in registers. The Intel Core i7 uses 48 logical address bits (36 for physical memory) and supports 4 KB pages [JLH12]. With each page table entry requiring one 64 bit word, this yields more than 200 TB just to store the page table. Such a page table cannot be stored in main memory but has to be paged itself. Concepts like multi-level paging and inverted page tables are used to address the page table size problem.

When the page table is too big to be held in registers, the problem occurs that each memory access would require another memory access just to read the corresponding page table entry from main memory. This would reduce the memory performance in an unacceptable way. Therefore, such systems typically use a hardware cache called Translation Lookaside Buffer (TLB) to store recently used page entries. Only if the TLB does not hold the corresponding page entry, the entry is read from the page table in main memory. This case is referred to as soft miss.

In personal computers and servers processes typically use more addresses than are actually available in main memory and the number of pages present in main memory is usually significantly smaller than the number of pages used during the lifetime of a process. Currently unused pages are stored in secondary memory. Whenever a process accesses a page not present in physical main memory the MMU issues an interrupt to the operating system. The OS loads the missing page from the secondary storage, e.g. a hard disk. This situation is referred to as hard miss. Because the space in main memory for pages is limited, another page has usually to be written back to secondary memory whenever the OS loads a new page from secondary memory into the corresponding page frame. This process is called swapping.

Because the complete address space is divided into pages of equal size, the swapping algorithm can select a page to be replaced in an arbitrary fashion, e.g. following a least
recently used, or round-robin policy, without fragmenting the main memory. However, because of the fixed size, memory unused within pages (since the code size of a program is no multiple of the page size) is lost. This is called *internal fragmentation*.

### 4 Design Parameter Page Size

This Section investigates how the page size should be chosen with respect to the three major requirements: protection granularity, memory efficiency, or real-time behavior.

#### 4.1 Protection Granularity

Since access rights are always the same for all addresses within a page, the most flexible approach to protection would be pages containing only one memory word. Consider for instance the former example of memory mapped IO and adjacent control registers of different devices, e.g. the CAN controller and a Digital-to-Analog Converter (DAC). Assume further that a piece of code needs write access to the CAN controller but shall not be allowed to access the DAC. The address areas of the CAN controller and the DAC cannot be expected to be aligned with page frames that are usually larger than the combined address areas of both devices. Therefore, the page size would have to be very small to allow for small enough protection blocks.\(^1\) A similar case would be common data areas for communication used by different processes to transfer small amounts of data, e.g. via the RTE (AUTOSAR Run Time Environment).

#### 4.2 Memory Efficiency

##### 4.2.1 Internal Fragmentation

A central issue regarding memory efficiency is the effect of internal fragmentation. The amount of memory wasted by internal fragmentation is proportional to the number of partly used pages and their size. As with protection granularity this observation calls for small pages. The following quantitative experiments demonstrate the impact of the design parameter page size on memory efficiency.

For the experiments an AUTOSAR compliant system is considered. The only addition is the assumption that memory management is supported and that each AUTOSAR Runnable has its own logical address space. The current AUTOSAR standard only supports memory protection and no management of virtual addresses. Moreover, tasks are the smallest runtime unit for which memory access rights can be defined currently. However, because

---

\(^1\)Note that, even if the alignment between the physical addresses of the control register blocks fits well with different page frames (in general that would be per chance only), the frames have to be small enough to prevent applying wrong access rights to other neighboring address areas.
AUTOSAR software components are based on runnables rather than tasks, it is natural to assume a protection scheme on this level of granularity. Anyway, a system supporting memory protection on task level only, can always be used for protection on runnable level by assigning each runnable to its own task.

For the processor a 32-bit architecture is assumed and a MMU supporting page sizes of 256 Byte, 512 Byte, 1 KB, 2 KB, or 4 KB. Four different exemplary task sets (sets of runnables) are considered. Each task set consists of a number of tasks that use runnables with different characteristics (i.e. code size and data size). For convenience and to limit the space used by tables in the paper, each task uses runnables of the same type, i.e. in a task set with \( n \) runnables in \( m \) tasks, each task has \( \frac{n}{m} \) runnables and there are \( \frac{n}{m} \) different runnable types. Any runnable uses the minimal number of code and data pages depending on the page size. Note that, the examples do not consider the need of separate pages for stack areas, which every task usually requires, or for pages needed to protect IO or to share data areas. The actual loss of usable memory is therefore worse than the given numbers.

**Task Set Tiny** This task set has 8 tasks, each of which uses 6 runnables of the type shown in Table 1 and uses a total memory for code and data of 186560 Byte.

<table>
<thead>
<tr>
<th>Runnable Type</th>
<th>Code Size</th>
<th>Data Size</th>
<th>No. of Pages per Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Code Size</td>
<td>Data Size</td>
<td>256</td>
</tr>
<tr>
<td>T1</td>
<td>400</td>
<td>480</td>
<td>4</td>
</tr>
<tr>
<td>T2</td>
<td>720</td>
<td>920</td>
<td>7</td>
</tr>
<tr>
<td>T3</td>
<td>1280</td>
<td>1680</td>
<td>12</td>
</tr>
<tr>
<td>T4</td>
<td>1800</td>
<td>2040</td>
<td>16</td>
</tr>
<tr>
<td>T5</td>
<td>4000</td>
<td>10000</td>
<td>56</td>
</tr>
<tr>
<td>T6</td>
<td>4400</td>
<td>10800</td>
<td>61</td>
</tr>
</tbody>
</table>

**Task Set Small** This task set has 16 tasks, each of which uses 6 runnables of the type shown in Table 2 and uses a total memory for code and data of 932992 Byte.

<table>
<thead>
<tr>
<th>Runnable Type</th>
<th>Code Size</th>
<th>Data Size</th>
<th>No. of Pages per Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Code Size</td>
<td>Data Size</td>
<td>256</td>
</tr>
<tr>
<td>S1</td>
<td>1040</td>
<td>1920</td>
<td>13</td>
</tr>
<tr>
<td>S2</td>
<td>1600</td>
<td>2680</td>
<td>18</td>
</tr>
<tr>
<td>S3</td>
<td>2760</td>
<td>3920</td>
<td>27</td>
</tr>
<tr>
<td>S4</td>
<td>3240</td>
<td>7216</td>
<td>42</td>
</tr>
<tr>
<td>S5</td>
<td>4000</td>
<td>10312</td>
<td>57</td>
</tr>
<tr>
<td>S6</td>
<td>5800</td>
<td>13824</td>
<td>77</td>
</tr>
</tbody>
</table>

**Task Set Medium** This task set has 64 tasks, each of which uses 3 runnables of the type shown in Table 3 and uses a total memory for code and data of 974592 Byte.

**Task Set Large** This task set has 100 tasks, each of which uses 8 runnables of the type shown in Table 4 and uses a total memory for code and data of 3582000 Byte.

Figure 1 shows the percentage of memory loss due to internal fragmentation of the given task sets for different page sizes. As expected, small pages lead to less fragmentation loss.
<table>
<thead>
<tr>
<th>Runnable Type</th>
<th>Code Size</th>
<th>Data Size</th>
<th>No. of Pages per Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>256</td>
</tr>
<tr>
<td>M1</td>
<td>980</td>
<td>1536</td>
<td>10</td>
</tr>
<tr>
<td>M2</td>
<td>1904</td>
<td>3416</td>
<td>22</td>
</tr>
<tr>
<td>M3</td>
<td>3192</td>
<td>4200</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Runnable Type</th>
<th>Code Size</th>
<th>Data Size</th>
<th>No. of Pages per Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>256</td>
</tr>
<tr>
<td>L1</td>
<td>1048</td>
<td>1904</td>
<td>13</td>
</tr>
<tr>
<td>L2</td>
<td>1556</td>
<td>1284</td>
<td>13</td>
</tr>
<tr>
<td>L3</td>
<td>2040</td>
<td>1360</td>
<td>14</td>
</tr>
<tr>
<td>L4</td>
<td>1612</td>
<td>1800</td>
<td>15</td>
</tr>
<tr>
<td>L5</td>
<td>1868</td>
<td>2572</td>
<td>19</td>
</tr>
<tr>
<td>L6</td>
<td>1560</td>
<td>1380</td>
<td>13</td>
</tr>
<tr>
<td>L7</td>
<td>2344</td>
<td>2616</td>
<td>21</td>
</tr>
<tr>
<td>L8</td>
<td>2756</td>
<td>8120</td>
<td>43</td>
</tr>
</tbody>
</table>

Task Set Small suffers from the least amount of fragmentation, but even there, more than 10 percent loss occurs as soon as the page size is 1 KB or above. A noteworthy observation is that the increase of the fragmentation rate differs substantially between task sets.

Under the assumption that these results are not significantly worse than in future automotive systems with memory management, a page size of 4 KB or above would certainly not be acceptable in practice. Given these numbers even 2 KB would be too much. However, it might still be the case that these numbers are misleading. One issue could be that a large amount of random sets would be quite different. To eliminate this uncertainty, Figures 2, 3, 4, and 5 compare the experimental results to the case that each last page is used by half only. The latter is to be expected for a large random experiment. As these figures show, the difference gives no reason to assume that the experiments lead to unjustified conclusions.

### 4.2.2 Page Table Size

Besides internal fragmentation the size of the page table itself is another important factor regarding memory efficiency. A small page size leads to many page table entries that need to be held in memory. Naturally, the page table is bigger for a larger address space. With 32 address bits 4 GB of memory can be addressed and the page table then has a size of 16 MB for a page size of 2 KB (assuming 8 bytes per page table entry). Of course the number of address bits and thereby the usable address space can be reduced. However, as Table 5 shows, especially for small page sizes this is still a significant factor, because every runnable with an own address space also needs its own page table and the OS will usually keep several page tables in main memory simultaneously. Consider that the amount of RAM in current ECUs is usually in the range of below 100 KB to at most 2.7 MB for the newly announced Infineon AURIX platform. Assume for instance for a future system a
4.3 Performance and Real-Time Behavior

The design of an MMU can influence performance and real-time behavior of future automotive systems in more than one way. As discussed above, the size of the pages determines the number of page table entries. With too many entries, a page table cannot be held in an array of registers but has to be placed in main memory. For each memory access this would require another memory access to look up the proper page table entry.
This dramatic performance penalty is usually reduced by caching the page table in a fully associative register set called Translation Look-aside Buffer (TLB). The effect of such a solution and further issues regarding real-time behavior are presented below.

### 4.3.1 Immediate Influence on Program Level

This Section considers the immediate effects on program execution time and how to predict these. Indirect effects, e. g. late effects (i. e. effects that occur at later program points), are discussed later.

**Address Translation** Introducing MMUs requires to translate logical addresses into physical addresses for each memory access. As explained above, adding another memory access to read from a page table in memory is, although it would prevent timing deviations, not acceptable because of its serious performance impact.
Another approach to prevent timing deviations is to keep the translation table in processor registers. This is only viable, if the number of table entries is reasonably small. From a real-time perspective this is clearly the best approach as it is perfectly predictable and introduces no timing deviations.

The usual approach in standard IT systems is to use TLBs. If it can be guaranteed that a translation entry is always present in a TLB when needed, such a system can be designed to have no (additional) timing deviations. To achieve this, TLB entries could be loaded by the RTOS (Real-Time Operating System) into TLBs before a task needs it.

If the situation of TLB misses (also called soft misses) cannot be avoided completely, it might still be possible to eliminate it for certain parts of the system. This could for example be achieved by partial locking of TLBs. This feature is already available in some hardware architectures.

When a TLB miss occurs in a system with hardware managed TLBs the execution time of the concerned instruction is immediately increased by the memory access to the page table. It is important to consider this when analyzing the worst case execution time (WCET) [WEE+08] of a runnable or task. Depending on the hardware architecture this delay can be (partially) compensated by a pipelined execution. This effect can be considered when using an integrated cache and pipeline analysis as described in [Sch00]. As long as the hardware exhibits no timing anomalies [LS99] it is sufficient, but pessimistic,
to consider this immediate delay as TLB miss penalty outside of the WCET analysis.

The accuracy of a TLB miss prediction will depend on the policy to select TLB entries for replacement in a similar way as for cache misses. The best results can be expected for LRU (least recently used) algorithms, because of the similarity to caches [RGBW07]. For performance reasons a hardware implemented replacement policy is usually preferable. However, if a certain hardware architecture offers no LRU policy, a software based mechanism might be better regarding temporal predictability.

In case of a software managed TLB an interrupt is issued to the CPU and the RTOS has to handle the updating of the TLB. As opposed to hardware managed TLBs the current instruction is interrupted and a context switch into the RTOS is performed. After the RTOS has updated the concerned TLB it returns to the interrupted instruction and the TLB lookup is repeated. This is directly comparable to asynchronous interrupts by the RTOS for other reasons and suitable approaches to consider such cases can be found in [Sch02, Sch00, Sch03].

**Protection** When a memory access violates the protection settings monitored by the MMU, an interrupt is issued to the CPU and the OS has to handle this exception. Regarding immediate influence on real-time behavior this is not different to the above mentioned case of interrupts due to TLB misses in systems with software managed TLBs. Since a protection violation is no expected system condition it would be necessary to find an upper bound on the frequency of such errors to predict whether a system satisfies its real-time requirements. However, this problem occurs alike with systems using memory protection units (MPUs).

**Managing the Translation Table** Setting up the translation table for each runtime unit (e.g. an AUTOSAR runnable) takes additional time that has to be considered when reasoning about real-time guarantees. Depending on the way this is handled in a particular system, this can range from one time initialization at startup to recreating the table after each preemption. Considering the resulting impact on the real-time behavior requires no fundamentally new methods but special support by tools is needed.

### 4.3.2 Indirect Influence on Program Level

In this Section late effects (a late effect is a situation where an initial cause leads to an increase in execution time at a distant place, i.e. at a later point in time) and other indirect influence on program level is considered.

Besides the replacement of entries in the TLB itself further cases have to be considered. When a TLB miss occurs at least the corresponding translation table entry has to be read from memory. In systems with hardware managed TLBs the MMU itself performs this read operation. Unless the hardware designers undertake special measures to prevent this, the memory access involves updating the cache state and therefore potentially influences the later execution of programs. In the case of software managed TLBs additional memory accesses to fetch the associated RTOS function and involved data are necessary and change the cache state. The potential influence within the same or other programs has to be considered. Suitable approaches that could serve as solution pattern can be found.
in [Sch00, Sch03]. Note that in multicore systems the state of shared caches can also be affected.

Another late effect can occur in systems with a write back cache policy. If cache content is displaced that needs to be written back to main memory, it could happen that the matching translation table entry is no longer in the TLB. Thus the write back could cause later TLB misses and these TLB misses could cause cache misses. Considering these cases would introduce significant loss of precision for analysis tools. In general data caches in real-time systems should not be configured to use a write back policy.

In systems with branch prediction unexpected delays can occur if a branch is mispredicted and the CPU fetches the wrong target [RWT+06]. Since this can only happen if the wrong target is in the cache, the situation is counter intuitive (a cache hit leads to longer execution times than a cache miss). Such counter intuitive behavior is named a timing anomaly. In similar manner a TLB hit (in conjunction with a cache hit) could lead to a higher execution time.

4.3.3 Immediate Influence on Scheduling Level

This Section discusses increased context switch costs and other immediate effects at scheduling level.

As mentioned before, TLB misses in case of software managed TLBs cause interruptions of the current code. The system then switches to RTOS context, executes the appropriate RTOS function, and switches back to user level. This sequence has to be considered when analyzing the real-time behavior of a system.

Furthermore, it has to be considered generally that a context switch usually involves changing the page table. If TLBs are used this might involve flushing TLBs. In hardware architectures that share TLBs between processes (runtime units) flushing of TLBs is not necessary. But subsequent TLB misses can still occur due to the context switch.

If TLB entries belonging to one process cannot be affected by other processes, the prediction of the temporal behavior of a process regarding TLB misses can be performed on the associated program alone. Otherwise, the scheduling of processes has to be considered, similar to caches the program level and scheduling level analysis have to be integrated more closely in such a case, cf. [Sch00].

**Protection** When a memory access violates the protection settings monitored by the MMU, an interrupt is issued to the CPU and the OS has to handle this exception. Regarding real-time behavior this is not different to the case of using a memory protection unit. Therefore, this issue is not further elaborated here, even though it is not trivial from a systems perspective.

4.3.4 Indirect Influence on Scheduling Level

**Using the MMU to transfer code or data into RAM** Although this feature is out of the scope of this paper, it should be briefly mentioned that using MMUs to transfer code or
data into RAM induces serious impact on real-time behavior in several ways. Consider a system where code has to be loaded into RAM before being executed. Whenever a requested page is not present in RAM it has to be loaded from secondary memory. Such a situation is called a hard miss.

Either the RTOS immediately loads the missing page from the secondary storage or the current task is blocked to favor another task that can run immediately. In the first case the WCET of the concerned task is artificially increased thus reducing the possible processor load. In the second case the scheduling method has to support blocking of tasks in the middle of their execution. The AUTOSAR standard does not support this and serious changes on the OS level would be necessary to allow for this. For instance, while Tasks in AUTOSAR can wait for events, they have to explicitly call the special system call `WaitEvent()`. In addition tasks may never hold a resource whenever they call `WaitEvent()`. The reason is that the OSEK/AUTOSAR mechanism (Immediate Ceiling Priority Protocol) for mutual exclusion uses priorities to ensure its properties such as absence of deadlocks and predictability of worst case behavior. If a task could be blocked in middle of execution the mechanism could no longer be used.

5 Conclusion

The issues discussed in this paper demonstrate that the prevailing page based MMU approaches have serious drawbacks from the perspective of automotive software engineering. The achievable protection granularity is inferior to memory protection units. In terms of memory efficiency neither large nor small pages lead to acceptable results, as large pages cause a lot of internal fragmentation and small pages lead to serious memory demand by page tables (cf. Table 5). The performance suffers from the need for page table management and the real-time behavior is hard to predict in presence of TLBs and due to other issues.

Still the wish for a hardware supported memory management in automotive systems will probably grow into a nonrejectable requirement over time. Fortunately, there is a long known approach that offers almost anything the automotive industry will need here: a small and variable protection granularity, freedom from internal fragmentation, translation tables that fit into register arrays, and as a consequence no problems to predict the real-time behavior. All this could be provided with a segmentation based MMU. In standard IT systems the segmentation based MMU approach is at the border to extinction. Probably this is the case because many people see swapping as the primary purpose of MMUs and the fact that segments of varying size require sophisticated algorithms in this regard other than pages of uniform size that can be replaced in an arbitrary fashion.

May be we will see a renaissance of segmentation based MMUs due to their advantages in the automotive domain and fields with similar requirements such as avionics. Whether this will be the case or not remains to be seen.

\[3\] This is for instance the case when a system uses NAND Flash as secondary memory.
\[4\] E. g. 32 size adjustable segments per runnable would require 64 registers plus some logics.
References


