Predictable and Efficient Virtual Addressing for Safety-Critical Real-Time Systems

M. D. Bennett* N. C. Audsley
Real-Time Systems Research Group, Dept. of Computer Science, York, UK
{mikeb,neil}@cs.york.ac.uk

Abstract

Conventionally, the use of virtual memory in safety-critical real-time systems has been avoided, one reason being the difficulties it provides to timing analysis. The difficulties arise due to the Memory Management Unit (MMU) on commercial processors being optimised to improve average performance, to the detriment of simple worst-case analysis.

However, within safety-critical systems, there is a move towards implementations where processes of differing integrity levels are allocated to the same processor. This requires adequate partitioning between processes of different integrity levels. One method for achieving this in the context of commercial processor is via use of the MMU and its support for virtual memory.

The focus of this paper is upon the provision of virtual memory for processes of all integrity levels without complicating the timing analysis of safety-critical processes with hard deadlines. Also, for lower integrity processes without hard deadlines, the flexibility of the virtual memory provided does not restrict the process functionality. The virtual memory system proposed is generic and can be implemented on many commercial architectures eg. PowerPC, ARM and MIPS. This paper details the PowerPC implementation.

1. Introduction

Real-time safety-critical systems often require predictable execution to enable off-line guarantees to be made regarding their run-time behaviour. This implies the need to perform static and dynamic analysis to show that key functional and non-functional properties of the system will hold at run-time. Typical properties include timing, resource usage and safety.

Usually, safety-critical systems are developed such that software processes with similar integrity levels are allocated to the same processor. This is potentially inefficient and a wasteful use of resources because of the necessity to over-engineer lower integrity tasks. However, it is a valid approach to ensure spatial and temporal separation between processes of differing integrity levels (as they are allocated to physically different processors). Spatial and temporal separation ensure that a process of high integrity cannot be unduly affected by a the execution of a processed with lower integrity.

Future safety-critical systems may allow processes of differing integrity levels to be allocated to the same processor. One such example is the Integrated Modular Avionics (IMA) effort [2, 14, 20]. Traditional avionics systems are designed using separate systems for different functions eg. autothrottle and autopilot. IMA allows for functional integration, removing the need for separate computing systems for each aircraft function. IMA systems are based around a distributed computing resource, with a standard operating system interface [3], permitting software functions to be allocated to any processor.

The allocation of processes with differing integrity levels to the same processor within a safety-critical system requires that both temporal and spatial separation between processes is maintained. Temporal partitioning can be provided by a standard operating system (to limit general resource usage) and standard scheduling techniques [8]. One such initiative in the commercial avionics industry is APEX (APplication/EXecutive) [3].

One approach to the provision of spatial separation is the use of the Memory Management Unit (MMU) on the processor. Unfortunately, safety-critical systems are forced to use commercial processors (eg. PowerPC) designed for the mass-market with their complex and large MMU tuned to average-case rather than worst-case performance. This leads to difficulties when modeling and analysing the worst-case performance of virtual memory accesses, which requires the translation of a virtual to a physical address on every memory access [6]. Software Fault Isolation (SFI) [17, 19] is an alternative method to provide spatial separation that does not use the MMU. However the MMU is a practical, well-understood feature found on virtually all

*This work is supported by EPSRC and BAE SYSTEMS

modern processors.

Provision of spatial partitioning by using the MMU in a very restrictive manner can lead to good timing analysis, eg. by using a base / limit approach (see section 5.1). This approach alone is adequate for processes with the highest integrity requirements, eg. those with hard deadlines. However it is restrictive, particularly for processes of lower integrity levels, eg. processes with soft deadlines. Such processes may require dynamic memory allocation; paging to solid-state backing stores might be required for huge database applications such as terrain modeling.

This paper focuses upon the provision of flexible predictable virtual addressing as a means of providing spatial partitioning in safety-critical systems (including IMA) that use modern commercial processors. We show that it is possible to provide virtual address spaces for soft real-time tasks, retaining their benefits; whilst at the same time providing hard real-time tasks with predictable and efficient address translation. This result is general to many modern processors and several examples (PowerPC, MIPS and ARM) are chosen to show how specific hardware features can aid implementation. In this paper we are only concerned with the temporal properties of MMUs and their functional correctness is assumed.

Section 2 gives a brief review of the memory management facilities found in modern processors and how that corresponds to the mechanisms provided by operating systems. From this mismatch comes the requirements (section 3). The general solution and implementations is given in sections 4 and 5. The PowerPC then provides an excellent example of how processor-specific features can aid an efficient implementation (Section 6). Conclusions are given in section 7.

2. Background

2.1. Virtual Addressing in Modern Processors

Modern architectures, typified by the PowerPC (64-bit implementations), support a large virtual address space (80 bits). Upon a reference to an effective address (EA), hardware translates this to a physical address ($\leq$64-bits) by looking up the required mapping. Part of the virtual address is used to divide the address space into equally sized *pages* shown in figure 1. The rest of the address indicates the offset into this page. The PowerPC supports a 4K page size, which is typical of modern processors.

The mapping between virtual and physical pages is usually cached in a translation look-aside buffer (TLB) or Address Translation Cache (ATC). This provides quick and predictable lookup. Unfortunately not all mappings can be stored on-chip. In the event of a TLB miss a combination of software and hardware look for the mapping in a *pagetable*

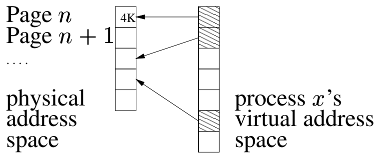

Figure 1. Only some of process x's virtual pages are mapped to physical page frames. If the process attempts to access a page not mapped there will be a page fault.

structure in memory. Following the RISC philosophy it is now the trend to generate an exception and allow the programmer the flexibility of programming the *pagetable entry* lookup routine, rather than executing a fixed hardware table walk. The pagetable entry, once found, is placed in the TLB and the memory reference can continue. If the entry does not exist a page fault in signaled and the software must recover. The whole process is shown in figure 2.

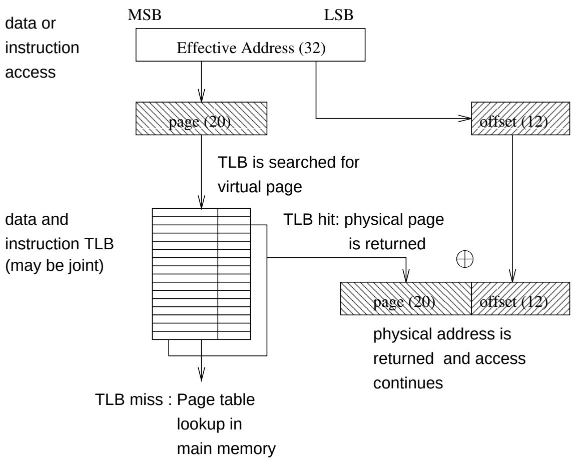

Figure 2. Simplified virtual to physical translation

A review of common pagetable designs is given in [6]. Typical criteria for good pagetable design are: (1) minimising **average case** table lookup time, and (2) minimising **average case** table size. Hardware assistance may be provided to achieve (1) (eg. The PowerPC provides a hardware hashing function). (2) is a growing requirement because of increasingly massive ratio between virtual and physical memory and the need to support sparsely populated address spaces. It is infeasible to keep the the virtual-to-physical mappings in a simple array or tree structure because physical memory would soon become exhausted. Elphinstone [6] suggested that guarded pagetables combined with a cache fulfill requirements (1) and (2) in a modern microkernel.

Unfortunately the average case parts of criteria (1) and

(2) are the biggest impediments to using most implementations of virtual address spaces.

2.2. Previous Work on Virtual Address Spaces in Real-Time Operating Systems

Available Real-Time Operating Systems (RTOS) have rarely tackled unpredictability due to MMU address translation. This is unfortunate and naïve because the powerful abstraction of a virtual address space is made useless for safety-critical real-time systems with hard deadlines, when it need not be. Two approaches are generally seen: (i) the standard pagetable algorithm is used or (ii), the MMU and all address translation is disabled and processes run in the same memory space:

i. Real-Time Mach [18] only makes one concession to real-time memory management through the `vm_wire` call, which allows a region of virtual memory to be *pinned down*. Mach [1] only actually allocates physical memory on a page fault, so called *lazy evaluation*. The `vm_wire` call ensures the whole region of memory is actually allocated before a thread accesses it. RT Mach does not mention TLB misses and does not specify how to calculate the worst-case lookup time for a memory access, even assuming it is in memory. Fiasco [9] is another real-time kernel for the 80x86 architecture that provides all tasks with a separate address space. It is, however, constrained by the architecture's fixed pagetable walk and the time for translation from virtual to physical address may be very large, in the worst case. The EMERALDS kernel [23, 22] uses the MMU to implement important spatial partitioning, but restricts the size of the pagetable to fit in limited memory embedded systems. However, the kernel is primarily for embedded systems and every task must use this restricted functionality.

ii. QNX Neutrino links user threads together to give multi-threaded execution, without memory protection. RT-Linux [4] loads all real-time tasks into the kernel memory space as modules, again losing protection. However, this is the closest to the ideal, because it still allows normal tasks to run, above the kernel.

Attempts to abstract away from the particular implementation of pagetable designs have been made in Mach [16] and Chorus [1], both microkernels. This is achieved by defining a machine independent interface to the required basic operations. An example is `pmap_extract` in Mach, which performs the virtual-to-physical mapping. The benefits of these efforts are portability – abstracting away from the particular MMU hardware and allowing arbitrary pagetable designs to implemented in the kernel to suit application design. In this case a pagetable for real-time tasks. However the properties of such a pagetable would make it inefficient for other types of tasks due to excessive size or poor average case lookup time. Current operating systems do not give predictability, flexibility and protection. Low-level manipulation of address spaces is hidden behind the kernel which is optimised for good average case performance, with poor real-time properties.

An alternative to hiding the pagetable behind the kernel is found in the Exokernel [7]. Its central claim is that abstractions restrict performance. It promotes the controlled exporting of low-level processor features to the user-level. This allows the application designer total control and would give access to TLB and cache management functions. This would allow applications to manage their own real-time requirements and adapt easily to new architectural features. However this approach suffers through complexity and as modern processors have broadly similar architecture, the simplicity of the abstraction outweighs the possibility of it becoming obsolete.

In summary changing the pagetable design alone does not give the flexibility required. The next section discusses the requirements for a kernel that will solve these problems.

The Spring kernel [15] does discuss predictability due to address-translation. They propose explicitly managing the contents of the TLB to guarantee a TLB hit. However this report does not discuss the integration of soft and hard-real address spaces. Since the report was written processors have emerged with richer address translation schemes, enabling more practical mechanisms for guaranteeing fast and predictable translation.

3. Requirements for a Safety-Critical Real-Time Kernel

A kernel suitable for running multi-criticality processes must cater for task models in all of the unshaded boxes shown in figure 3. Tasks that require total predictability may have to co-exist with tasks that require total efficiency. It will be seen later that the shaded box is key to being able to implement all these requirements: Hard real-time tasks should never need to dynamically allocate memory.

This paper emphasises these important requirements:

1. *The kernel must provide primitives that are inherently predictable in order to support hard deadlines so that high-level design and analysis are not compromised.*
2. *The kernel must support the co-existence of tasks with different real-time constraints and enforce strict spatial and temporal partitioning between them.*
3. *The kernel must be able to adapt to the rapid and constant changes in computing hardware.*

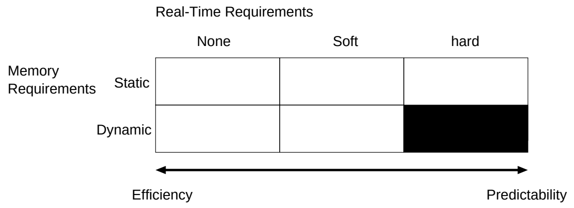

Figure 3. Classification of tasks in a safety-critical real-time system

Getting the best performance on a particular platform requires exploiting the features provided on a specific processor within a broadly similar architecture. This includes real-time performance, exploiting the particular facilities of the MMU.

4. *Adding extra functionality to the kernel should not adversely affect the performance of the existing features.*

   In practice efficiency is often a prime consideration. Adding real-time features should not impose a performance penalty on soft or non-real-time applications. For example adding extra kernel calls and data has been shown to reduce the performance of microkernels [12].

4. Generic Solution: Real-Time Address Spaces

L4 [11] is an ideal starting point for the integration of real-time semantics because it is minimal and efficient. It is a modern microkernel, incorporating only 7 system calls and three basic abstractions: *threads*, *address-spaces* and *inter-process communication*. Its implementation is extremely hardware dependent in order to achieve maximum performance, fulfilling requirement (3). The memory subsystem in based on a paged memory architecture similar to that found in most modern processors. The L4 memory management operations are such that each address space can be described by a pagetable. Sharing memory is simply achieved by copying pagetable entries and this is recorded in a *mapping tree* per physical page.

To fulfill requirement (1) we introduce the concept of a **real-time address space**. Here pages may not necessarily be equal to the system page size (eg. 4K) and even within address spaces pages may be of different sizes.

Definition: An address space σ is a **real-time address space** if:

> The worst-case execution time of the virtual-to-physical translation for all pages that do not result in a page fault in the address space is known.

This is useful because one of two possible analysis techniques is usable on many modern processors for real systems:

- This worst-case time is used in timing analysis for modeling **every memory reference.** This is the simplest possible technique, independent of scheduling paradigm and avoiding all need to model the processor's internal structure.

  Whether this worst-case is small enough to result in a schedulable system depends on the characteristics of the real-time tasks. Tight deadlines might require efficient implementation such as the hardware solution supported by the PowerPC (see Section 5.2). Conversely the worst-case translation time in a traditional pagetable might be sufficient for a single real-time task with a long deadline, though lowering utilisation of the processor in the system. A predictable pagetable design with non-pathological worst-case lookup time might be chosen as a compromise.

- In some cases the processor design may make it impossible to create an efficient real-time address space, yet there exists enough information that it is not necessary to assume the worst-case occurs on every memory access. In this case it is helpful to know the number of pages that are actually mapped to physical pages because by bounding this value we can still analyse processors that make the first analysis technique difficult or inefficient. In this case we will show that pessimism can be reduced significantly.

4.1. An Enhanced Kernel Model

An address space is a basic abstraction and it is preferable to separate this completely from the notion of a thread. The ability to create active tasks (In L4 threads and address spaces are created in the same call) is removed and instead an `as_create` (address space create) call is introduced.

This call accepts extra parameters which allow the address space to be initialised if this is desired. To accommodate the novel initialisation requirements of real-time applications (eg. contiguous memory) `as_create` will perform a RPC to the specified pager. This means that instead of an arbitrarily complex system call, custom pagers can be implemented that extend or change the σ_0 protocol to suit application requirements and processor features.

When L4 creates an active task the first instruction fetch will cause a page fault which will be serviced by this pager. For real-time address spaces there may be no pager so there must be a way to fix the virtual-to-physical mappings before the first instruction is executed.

Because of the different semantics and implementation of normal and real-time of real-time address spaces, sharing memory as done in L4 is difficult and we prohibit it. However if the hardware supports sharing memory and this is desirable property we must find a way to share memory efficiently. It is envisaged that this will be a mechanism provided by the real-time pager.

In the implementations outlined in the rest of the paper the virtual address space available for hard real-time tasks is restricted (ie. some virtual pages have a fixed mapping to the nilpage) because it is the only way to ensure an efficient implementation and maintain the definition of a real-time address space. The tradeoff between predictability and restricted functionality is valid because of the real world characteristics of hard real-time applications: their memory requirements are known or bounded and dynamic memory allocation is never needed.

The kernel shall provide sufficient information to deduce why mapping or granting failed.

5. Implementation

5.1. Modular Page-Tables

Support for multiple pagetables in the kernel helps fulfill requirement (2) because of the multitude of designs available to cater for different needs. Architectures that support software TLB replacement such as MIPS and PowerPC support multiple pagetable designs.

Multiple pagetables add minimal overhead to a kernel, satisfying requirement (4). This is particularly important because in a microkernel, TLB refill performance is critical and good performance in general is derived from the kernel being small [12].

Overheads for modular pagetables:

- *A test to select and branch to the correct TLB miss handler.* Typically only a few instructions would be added per pagetable.
- *The code for the each table lookup*. This would be a few tens of machine instructions executed in a tight loop.
- *The size of the pagetable.* This would be dependent on application requirements.

Figure 4 shows an example of modular pagetables. Address Space A is initialised with a default address space (defined by the architecture) and is bound to the default pager (σ_0) and pagetable handler. Address Space B is bound to a specialised real-time pager, σ_c so the application can request an amount of memory (128K) be initially mapped into the address space. This allows the first instruction to execute inside this space without generating a page fault. The pager protocol ensures that accesses beyond this will generate a synchronous exception which can be handled. The pagetable design is a simple array indexed by virtual address. This can be done in constant time, with a single memory reference. The physical memory is not exhausted by the pagetable as virtual address space is limited to 128k, meaning only 32 pagetable entries are required (Assuming a 4K frame size). Upon a TLB miss the kernel conceptually queries the thread's pagetable design and performs the appropriate call. The worst case execution time of the translation in B is constant and small because the `vm_to_pm_array` call is O(1).

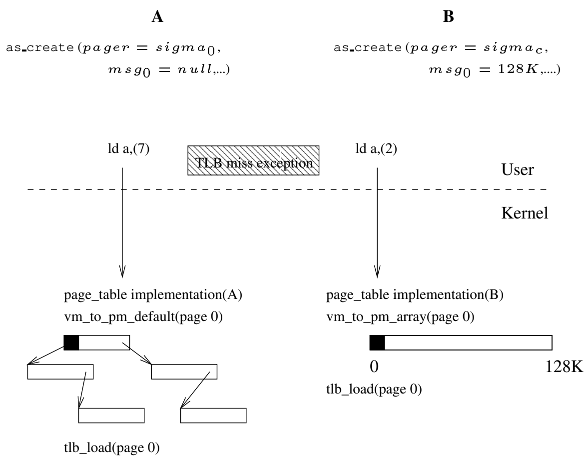

Figure 4. Modular Pagetables

5.2. Implementation on Modern Processors

Most modern architectures can implement efficient real-time address spaces. This section will discuss how this can be achieved in the ARM, PowerPC and MIPS architectures.

StrongARM SA-110 The ARM architecture performs pagetable lookup in hardware. There is no ability to customise these routines as would be required for modular pagetables. However implementations of these architectures still cache these translations in the TLB. The key idea is that as long as it can be guaranteed that the entire set of mapped virtual pages remain in the TLB, the worst case translation time is the TLB lookup time. This is heavily optimised in all architectures because of the frequency of memory accesses.

The StrongARM SA-110 processor [10, 21] is a suitable IMA component. It is high-speed and low-power. Architecturally, it has a separate 32-entry instruction and data TLB for storing pagetable entries that are replaced in a round-robin fashion. Replacement is not programmable. Each pagetable entry stores the translation for 4K (small page), 64K (large page) or 1MB (section) pages. If the address space is constrained to 32 mapped pages, real-time address spaces can provide 0 to 32 MB of physical memory.

It is simple to argue about the execution of threads in these address spaces. Each thread can incur a maximum of 64 TLB misses in each execution (32 instruction and 32 data). For analysis this time can be added to the task switch time [5]. With its hardware pagetable lookup, this time is not likely to be unduly pessimistic, two memory fetches the dominant contributers in terms of time. These will be negligible in all but the briefest executions of threads. For a small real-time address space of 1MB a maximum of 2 TLB misses will be incurred per thread execution.

PowerPC 603e The PowerPC 603e provides TLB load instructions which allows total control over where a pagetable entry is placed in the TLB. In a 32-bit virtual address the 12 least significant bits [20-31] are copied directly into the physical address. The next 5 bits [15-19] are used to select an entry in the TLB, which is 2-way set associative. The `tlbld` and `tlbli` instructions load a pagetable entry into a position specified by the contents of a general purpose register. Note that this allows $32 \times 4 \times 2 = 256$K of contiguous memory to be accessed without incurring a TLB miss.

The PowerPC also provides Block Address Translation (BAT) registers that translate large blocks of virtual address space to physical addresses by simply matching a number of bits of the virtual address and adding an offset. This resides completely in hardware and is guaranteed to take precedence over the TLB. There are 4 BAT registers that individually can cover from 128KB to 256MB. Using the BAT and the TLB allows any range of contiguous memory from 4K to 4096K to be translated without incurring a TLB miss.

Thus if the mapped pages in hard real-time address spaces on the 603e are limited to this range, the worst case translation time from virtual-to-physical address is simply the longer of the time it takes to search the TLB and time it takes to match against the BAT registers. Best of all other tasks are free to use normal virtual address translation only incurring negligible overhead (one simple register test to check if the thread is in a real-time address space).

Our work at York has involved improving the hard real-time characteristics of an L4 type microkernel on the PowerPC platform. For simplicity a hard real-time address space is translated exclusively by the BAT registers. The four sets of BATs allows mapping of other areas of memory for device driving and data sharing. However arbitrary granting and mapping is not supported because the BAT registers can only represent a limited set of (4) contiguous areas of memory. This is deemed to be sufficient for majority of hard tasks.

MIPS R4600 The R4600 [13] is another low-cost, embedded RISC CPU featuring a 48 entry, fully associative joint instruction and data TLB (this is actually implemented as separate data and instruction TLBs which are invisible to programmers). Each entry contains an odd/even pair of virtual pages which are refilled manually by software, upon a JTLB miss. Each entry holds a pagetable entry, which can be configured to be from 4K to 16MB in powers of 4. This makes it an ideal CPU for implementing modular pagetables. If efficiency is of prime concern then different pagetable sizes can be used to translate large blocks of memory.

An example kernel implementation would allow an application to create a virtual address space and map in a limited number of pages. On a thread execution in this address space the TLB would be pre-loaded with the corresponding pagetable entries. Analysis is simple and the cost of the virtual-to-physical translation is the cost of a JTLB lookup, which should be small. For a 1MB address space only two pagetable entries would need to be loaded. The pagetable entries would translate 512K of virtual address space each (Remembering each entry actually represents an odd/even pair of pages). This is akin to using the BAT registers in the PowerPC.

Summary The central message of these three examples is that real-time behaviour is not just architecture, but processor specific. Modern CPU's do give the ability to provide real-time memory management.

6. Evaluation of real-time address space implementation on the PowerPC 603e

We illustrate in the following example how using BAT registers on the PowerPC leads to significant predictability improvements when implementing real-time address spaces.

The following C fragment defines a 450 item array of 2K elements assuming an integer takes 4 bytes:

```
typedef struct bigstruct {
      int id;
      int data[511];
} bigstruct_t;

bigstruct_t mydata[450];
```

Consider repeated execution of the following simple array access where the index is subject to change. The worst-case execution time must be known.

```
x = mydata[index].id;
```

The corresponding PowerPC assembler generated from `gcc` is shown below. The data access that requires translation is marked. The other instructions are simpler to analyse (immediate loads and register operations).

```
lis   9,x@ha
lis   11,mydata@ha
```

```
lis   10,index@ha
lwz   0,index@l(10)
mr    10,0
slwi  0,10,11
la    11,mydata@l(11)
lwzx  0,11,0          % Actual Load
stw   0,x@l(9)
```

This was executed repeatedly in three different ways:

1. Linear - Every element in the array is accessed, incrementing the index by one each time. This corresponds to a control loop or initialisation phase of real-time software.
2. Random - The array is accessed, using a pseudo-random index. This represents a piece of real-time software reacting to a random sensor input by looking up a data item.
3. Alternate - Two elements that map to the same TLB entry are accessed alternatively

The *linear* pattern generates alternate TLB misses, as each data item is 2K and the next item is guaranteed to be immediately following and hence in the TLB. The *random* pattern results in poor performance because the size of the array is considerably more than the size of the TLB, resulting in a majority of TLB misses. The *alternate* pattern should result in extremely poor performance because every data access should result in a TLB miss.

These routines were executed a number of times equal to the size of the array (450) with standard pagetables and with BAT registers. The page table structure is an efficient guarded pagetable [6] design combined with a 128 entry cache. The true execution time was obtained by subtracting the total execution time from the execution time with the array access line commented out. Times for 3 repeated executions are shown in table 1 to emphasise how little execution time varies. One execution unit is equal to 4 bus cycles (6×10^{-8} s on 603e). Level 1 and level 2 caching is disabled and the software TLB cache is emptied between executions.

Although the speedup in all three cases is impressive the important result is the reduction in variation of execution times when moving from standard pagetable translation to translation by the BATs. The execution time becomes independent of access pattern making analysis considerably simpler. The variation between the individual trials is also reduced by large factors.

The large difference in executions times reflects the number of TLB misses. The alternate result is anomalous as poor performance should be expected. However the PowerPC 603e TLB structure is two-way set associative which guarantees that two data items mapped to the same TLB entry can be mapped at the same time. Hence this access pattern is almost optimal, only resulting in two initial misses.

		Linear	Random	Alt.
Page Table Translation	Trial 1	19237	23491	12548
	Trial 2	18432	22821	12036
	Trial 3	18432	22379	12032
	Average	**18700**	**22787**	**12205**
	Variation	**4.3%**	**4.9%**	**4.2%**
Translation via BAT registers	Trial 1	4095	4123	4082
	Trial 2	4096	4107	4080
	Trial 3	4096	4130	4081
	Average	**4096**	**4120**	**4081**
	Variation	**0.02%**	**0.5%**	**0.05%**
Saving		14604	18667	8124
Saving (ms)		0.88	1.12	0.49
Speedup		457%	553%	299 %

Table 1. Comparison of the execution time for 450 array accesses with standard and BAT translation

	Cache on	Cache Off
bus cycles * 4	14545	28876
ms	0.87	1.73

Table 2. The effect of using a large software cache with worst-case access pattern

Results for an access pattern illustrating worst-case behaviour are shown in table 2. Here *three* data items that map to the same TLB entry are accessed in turn. The 603e TLB operates a FIFO replacement algorithm which ensures that every data access will trigger a pagetable lookup. The pagetable algorithm incorporates a separate cache. The table shows results for when this is enabled and disabled. Disabling the cache gives an accurate reflection of the worst-case where the three data items map to the same software entry and cause a cache miss every access.

This extra cache illustrates that small changes in pagetable design can make a big difference to performance and hence analysis. Without complex analysis it would be necessary to assume a pagetable miss on every data access introducing a pessimism of up to 237% on just this small section of code.

Release Jitter Using the BAT registers in a kernel for safety-critical real-time systems introduces an overhead due to release jitter. When switching from a soft real-time process with standard pagetables to a hard real-time process release jitter must be considered. This is due to the non-pre-emptibility of pagetable lookup shown in figure 5.

The release jitter is equal to the worst-case pagetable lookup time. An accurate measurement for this can be estimated from table 2 , when a miss occurs on every access:

$$(28876 - 4081/450) * 6 \times 10^{-8} = 3.3\mu s$$

This is equal to under 1% of the total execution time of the random access pattern when translated by the BATs. This can be reduced further by re-enabling external interrupts during the page table search. However this results in a more complex kernel as pagetable lookup state must be saved.

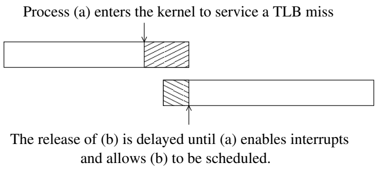

Figure 5. Jitter due to disabled interrupts

7. Conclusion

In this paper it is argued that future safety-critical real-time systems will require hard and soft real-time tasks with different criticalities to co-exist. It is shown that current kernels do not give flexible and predictable memory management yet modern processors support this, as demonstrated by experimental analysis of PowerPC code. Significant performance and predictability improvements are achieved.

Simple changes to a microkernel API to incorporate *real-time address spaces* are valid and can be implemented efficiently. This allows less pessimistic worst case execution time analysis to be performed, whilst still allowing conventional tasks good average case performance.

References

[1] V. Abrossimov, M. Rozier, and M. Shapiro. Generic Virtual Memory Management For Operating System Kernels. *Operating Systems Review*, 23(5):123–36, 1989.

[2] AEEC. ARINC Specification 651: Design Guidance For Avionics. Technical report, Airlines Electronic Engineering Committee, 1996.

[3] AEEC. ARINC Specification 653: Avionics Application Software Standard Interface (Draft 15). Technical report, Airlines Electronic Engineering Committee, June 1996.

[4] M. Barabanov. A Linux-Based Real-Time Operating System. Master's thesis, New Mexico Institute of Mining and Technology, Sorocco, New Mexico, June 1st 1997.

[5] J. Busquets and A. J. Wellings. Adding Instruction Cache Effect to Schedulability Analysis of Preemptive Real-Time Systems. In *Proceedings of RTAS96*, Boston,MA, June 1996.

[6] K. Elphinstone. *Virtual Memory In A 64-bit Microkernel*. PhD thesis, UNSW, 1999.

[7] D. R. Engler, . M F Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture For Application-Level Resource Management. *Operating Systems Review*, 29(5):251–66, Dec. 1995.

[8] A. Grigg and N. C. Audsley. Towards a Scheduling and Timing Analysis Solution for Integrated Modular Avionic Systems. In *Proceedings ERA Avionics Conference and Exhibition*, November 1997.

[9] M. Hohmuth. The Fiasco Kernel: Requirements Definition. Technical Report TUD-FI-12, Dreseden University of Technology, December 1998.

[10] D. Jagger. *Advanced RISC machines Architectural Reference Manual*. Prentice Hall, 1996.

[11] J. Liedtke. L4 Reference Manual Version 2.0. Technical Report RC 20649, IBM T.J. Watson Research Center, Yorktown Heights, NY, September 1996.

[12] J. Liedtke. μ-Kernels Must And Can Be Small. In *Proceedings of the Fifth International Workshop on Object-Orientation in Operation Systems, 27–28 October*, pages 152–155. IBM Thomas J. Watson Res. Center, Hawthorne, NY, USA, 1996.

[13] J. Liedtke and K. Elphinstone. Guarded Page Tables on the MIPS R4600. Technical Report UNSW-CSE-TR-9503, University of New South Wales, School of Computer Science, November 1995.

[14] M. Morgan. Integrated Modular Avionics for Next-Generation Commercial Airplanes. *IEEE Aerospace and Electronic Systems*, 6(8):9–12, August 1991.

[15] D. Niehaus, E. Nahum, J. Stankovic, and K. Ramamritham. Architecture and OS Support for Predictable Real-Time Systems. Technical report, March 1992. Spring internal document.

[16] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. J. Bolosky, and J. Chew. Machine-Independent Virtual Memory Management For Paged Uniprocessor And Multiprocessor Architectures. *IEEE Transactions on Computers*, 37 A01(8):896–908, Aug. 1988.

[17] J. Rushby. Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance, institution = NASA Langley Research Center. NASA Contractor Report CR-1999-209347, June 1999. Also to be issued by the FAA.

[18] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Toward A Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop*, pages 73–82, Oct. 1990.

[19] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-Based Fault Isolation. In *Operating Systems Review (USA) Fourteenth ACM Symposium on Operat ing Systems Principles*, volume 27, pages 203–216. ACM, Dec. 1993.

[20] A. Wake, P. R. Miller, P. Moxon, and M. A. Fletcher. Modular Avionics Operating System – Software Concept. In *Proceedings ERA Avionics Conference and Exhibition*, London, UK, 1996.

[21] A. Wiggins and G. Heisner. Fast Address-Space Switching on the StrongARM SA-1100 Processor. Technical Report UNSW-CSE-TR-9906, University of New South Wales, School of Computer Science, July 1999.

[22] K. Zuberi, P. Pillai, and K. Shin. EMERALDS: a small-memory real-time microkernel. In *Operating Systems Review*, volume 34, pages 277–291, Dec. 1999.

[23] K. Zuberi and K. Shin. EMERALDS: A Microkernel for Embedded Real-Time Systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 241–249, June 1996.