On Memory Protection in Real-Time OS for Small Embedded Systems *

Shoji Suzuki
The 1st Department of Systems Research
Hitachi Research Laboratory
Hitachi Ltd.
7-1-1 Omika
Hitachi, Ibaraki 319-12, Japan
suzukish@hrl.hitachi.co.jp

Kang G. Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer
The University of Michigan
1301 Beal Avenue
Ann Arbor, MI 48109-2122, USA
kgshin@eecs.umich.edu

Abstract

Memory protection is an important OS feature for the reliability and safety of real-time control systems. In this paper, we study the feasibility of memory protection in small embedded systems in which memory size ranges from several tens of KBytes to several hundreds of KBytes. We evaluate various protection methods in terms of memory consumption, processing overhead, multiple-thread support, region enlargement, and hardware support. We present a new protection method called Intermediate-level Skip Multi-Size Paging which skips unused intermediate-level page tables of Multi-level Paging and supports several page sizes. Our evaluation results show that this method along with Paged Segmentation and Short-Circuit Segment Tree are more cost-effective than other known memory protection methods. Also, the feasibility of Intermediate-level Skip Multi-Size Paging can be improved if a MMU supporting several page sizes is available for microprocessors.

Key Words:*Embedded system, memory protection, virtual memory, paging, segmentation, memory management unit (MMU)*

1. Introduction

In next-generation control systems, devices will be interconnected by field networks instead of the wire harness in use today. This means not only the reduction of wiring cost but also a paradigm shift from centralized control using high-performance microprocessors to cooperative and distributed control with several small microprocessors controlling up to tens of devices such as sensors, motors, valves, etc., and communicating with each other via field networks. Moreover, the progress of VLSI technologies will make low-cost single-chip small embedded systems available soon, consisting of high-performance RISC microprocessors, built- in memory of several tens to several hundreds of KBytes, communication transceivers, A/D converters, and ten or so I/O ports. The processes running on such embedded systems are usually quite small (at most several KBytes of code and hundreds of bytes of data and stack). Most of their memory requirements are known *a priori* and fixed (though the data and stack area sizes may vary dynamically which requires OS support). Also, embedded software is constructed with many modules, so it requires efficient multiple-thread service. Hence, any real-time operating system (RTOS) to be used in such embedded systems must not only have a small code size (10K to 20 KBytes) but must also provide support for the above-mentioned requirements of embedded systems.

Zuberi and Shin [1] developed a small distributed RTOS for embedded systems, called EMERALDS (Extensible Microkernel for Embedded ReAL-time Distributed Systems). EMERALDS provides various RTOS functions particularly suitable for small- to medium-sized embedded systems. The micro-kernel architecture of EMERALDS consists of process/thread management (real-time scheduling and memory protection), communication primitives, synchronization primitives (semaphores and condition variables), timer, interrupt handler and device driver primitives, and supports the extensibility of many necessary functions. These features are realized with a small RTOS of 13-KByte kernel code.

One key feature of EMERALDS is memory protection among processes and kernel. Memory protection is effective in enhancing system reliability and safety. For exam-

*The work reported in this paper was done during S. Suzuki's visit to the RTCL under a cooperative contract from Hitachi Research Laboratory.

ple, though 99% processes on some system may be bug-free and execute correctly, if the remaining 1% processes have bugs in them, they may corrupt the data of other bug-free processes and result in a system crash. Memory protection is effective in preventing such a crash and is essential for safety-critical embedded applications such as avionics, life support systems, and various other products in transportation or consumer electronics. Memory protection is generally realized by memory management software with MMU hardware support making the memory-protection method hardware-dependent. For example, microprocessors like Motorola 68040 [2], PowerPC[1] [3], etc., have hardware support for Paging while Intel386[2] [4] has support for Paged Segmentation. As far as software management aspects of memory management are concerned, they have been studied for more than 30 years, including working set [5] and virtual memory [6] concepts.

EMERALDS uses 3-level Paging of the MC68040 microprocessor for memory protection. Besides EMERALDS, there are several RTOSs for embedded systems supporting memory protection. MiThOS [7] supports physical-address-based memory protection with Paging. Chorus[3] [8] and LynxOS[4] [9] support logical-address-based memory protection with Paging. VRTX[5] x86 [10] supports logical-address- based memory protection with Paged Segmentation. In general, they are for mission-critical embedded applications or scalable RTOSs ranging from small embedded systems to general-purpose systems.

However, at present, we cannot just apply current memory-protection methods to cost-sensitive small embedded systems. Memory protection requires a MMU and memory management tables (e.g., page tables or segment tables). The MMU adds to hardware costs while the memory management tables consume RAM. The memory-protection procedure (equivalent to translation lookaside buffers (TLBs) miss handling routine in case of software-managed TLBs) decreases the processing speed of the microprocessor, or these penalties require higher-performance microprocessors with MMU and more memory. Furthermore, the page sizes which current microprocessors support are 4-8 KBytes, but these are too large for small embedded systems. We must therefore reduce page size, but the smaller the page size, the larger the amount of memory needed for memory management tables. Small page sizes also increase the TLB miss ratio and decrease the processing speed. This indicates a need to modify current memory-protection methods for small embedded systems. Moreover, in small embedded systems, it is very important to improve the efficiency of memory usage and reduce the memory requirements. So, it is necessary to use and share thread stacks efficiently among processes in a multiple threads support environment. If it is possible to enlarge or shrink data or stack area dynamically on demand, one can also improve the efficiency of memory usage.

[1] PowerPC is a trademark of Motorola, Inc.
[2] Intel386 is a trademark of Intel Corporation.
[3] Chorus is a registered trademark of Chorus systèmes.
[4] LynxOS is a trademark of Lynx Real-Time Systems, Inc.
[5] VRTX is a registered trademark of Microtec Research.

In this paper, we focus on the feasibility of memory protection for small embedded systems, using the following metrics:

- efficiency of memory usage
  - efficient threads support
  - data and stack region enlargement
- memory consumption
- processing overhead

The rest of this paper is organized as follows. In Section 2, we give a brief description of the current memory-protection methods for large systems, compare them qualitatively, and state the problems associated with them. Then, we describe a new memory-protection method for small embedded systems. In Section 3, we evaluate the feasibility of each memory-protection method, and in Section 4, we describe the effect of page size and logical-address-space size on memory-protection methods. In Section 5, we describe the MMU features needed to support memory-protection methods. Finally, we conclude the paper with Section 6.

We will use the following parameter values for our evaluation:

- The page size is set to 128 bytes. Page is the smallest unit for memory protection, and this page size – which can have 32 four-byte data – is feasible for data or stack areas of small embedded applications.
- Logical address space is 4 MBytes. This size is enough for small embedded system.

In general, the smaller the page size or the larger the logical address space becomes, the more memory consumed by the memory-protection method. We will discuss this in Section 4.

2. Memory-Protection Methods

Memory-protection methods are either physical-address-based or logical-address-based. Bit Map and Linked List belong to the former, while Paging, Segmentation belong to the latter. Described below are Bit Map, Multi-level Paging, Segmentation, Paged Segmentation, and Short- Circuit Segment Tree as currently available memory-protection methods, as well as a new protection method we have developed.

2.1. Classification of Protection Methods

Currently-available protection methods can be classified into the following five types.

(1) Bit Map: divides physical memory into pages for memory protection. Each process checks the protection information of the corresponding page whenever it accesses memory. For example, the memory management information per page could be constructed with just the 2-bit-size read/write protection information.

(2) Segmentation: divides the logical address space into several segments each of which is mapped onto the physical address space. This method requires a memory management table (segment table) which contains segment table entries (STEs) managing their own segments.

(3) Multi-level Paging: divides the logical address space into fixed-size pages and each page is mapped onto the physical address space. The memory management tables (page tables) are constructed in several levels of hierarchy in order to reduce the total size of page tables. Figure 1 shows the page table structure of 3-level Paging.

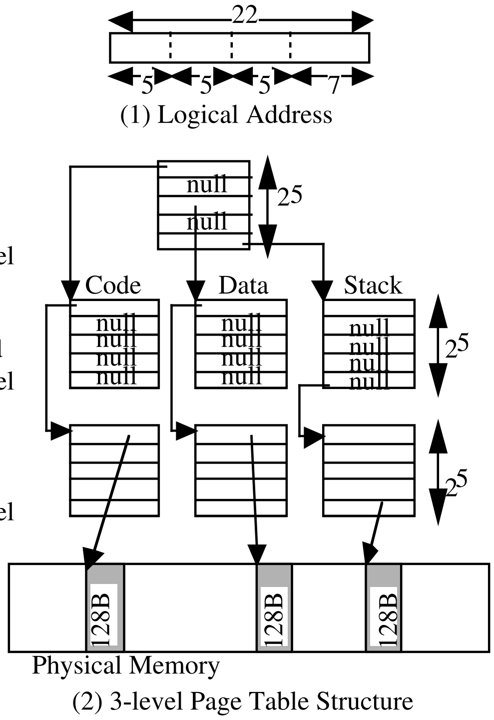

Figure 1. Multi-level Paging (3-level)

(4) Paged Segmentation: divides each segment of the logical address space into pages, and each page is mapped onto the physical address space. Memory is managed using a segment table and several page tables.

(5) Short-Circuit Segment Tree: of EROS (the Extremely Reliable Operating System) of the University of Pennsylvania [11] is based on Multi-level Paging. It skips page tables in which only the first page table entries (PTEs) is used, resulting in a decrease of both the processing overhead and the total table size as compared to Multi-level Paging.

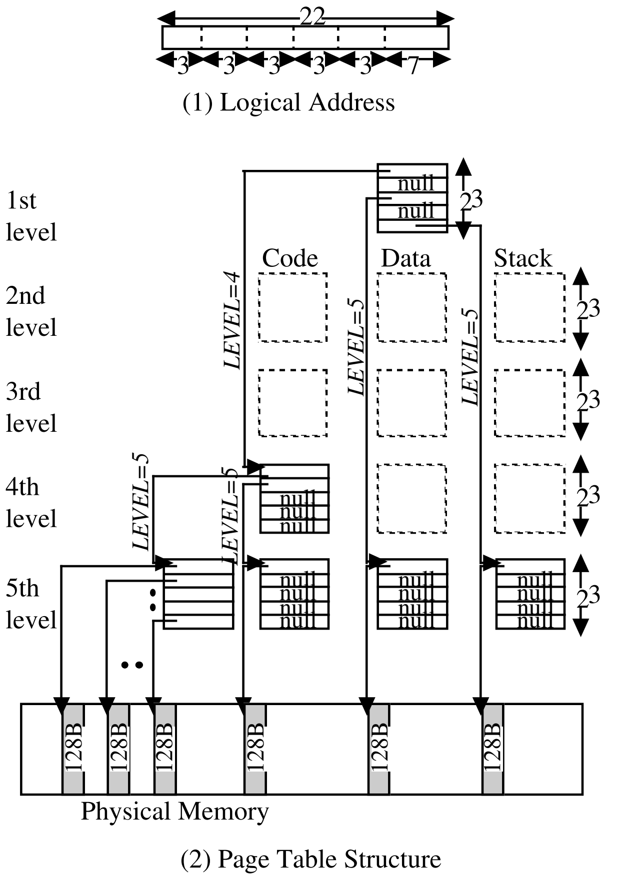

Figure 2. Short-Circuit Segment Tree (modified a little from the original in order to fit the small embedded system)

Figure 2 shows the page table structure of Short-Circuit Segment Tree. For the logical address format of 5-level Paging as shown in Figure 2 (1), we show an example of the page table structure of this method in Figure 2 (2). This example shows that the second- to the fourth-level page tables of data and stack area are skipped and the first-level PTE links directly to the fifth (bottom)-level page tables. The PTE contains the LEVEL information in the figure which indicates the level of the page table to be linked and makes it possible to omit intermediate-level page tables.

2.2. Efficiency of Memory Usage

It is important to improve the efficiency of memory usage in cost-sensitive small embedded systems. In a multiple-thread support environment, it is necessary to use thread-stack areas efficiently among processes by attaching and detaching them to each process on demand basis. Also, it is necessary to enlarge or shrink data or stack area dynamically on demand.

Efficient threads support: In case of Bit Map which is one of the physical-address-based memory-protection methods, we can use thread-stack areas efficiently among processes by dividing part of the memory *a priori* into a set of thread stacks, and making a pool of free thread stacks. In case of logical-address-based memory protection, the memory manager handles thread stacks with either segments or pages. In Segmentation, it is possible to use the segment table to support multiple threads by mapping one segment onto one thread stack, but this requires many STEs in segment table which increases the memory requirements. In paging-based management (Multi-level Paging, Paged Segmentation and Short-Circuit Segment Tree), the memory manager can handle thread stacks flexibly per page by getting one from the pool of free physical pages (page frames), making it easy to use thread-stack areas among processes. So we see that Bit Map and paging-based managements are suitable for multiple-thread support.

Region enlargement: In Bit Map and Segmentation, it is difficult to enlarge regions (data or stack) dynamically because regions must be mapped contiguously in memory and there is no guarantee of free space next to them. In paging-based management, the memory manager gets a new physical page from the pool of free physical pages and enlarges the region and can manage it flexibly. So, paging-based managements are suitable for region enlargement.

2.3. New Memory-Protection Method for Small Embedded Systems

To improve the efficiency of memory usage mentioned above, we present a new memory-protection method for small embedded systems.

Intermediate-level Skip Multi-Size Paging: skips the intermediate-level page tables of Multi-level Paging like Short-Circuit Segment Tree. The major differences between the two are that Intermediate-level Skip Multi-Size Paging supports multiple page sizes by enabling a PTE in each level page table to map onto the physical address space, and it can also skip page tables in which only one PTE (not just the first PTE) is used. The page size of a PTE in a level is the total size of the address space covered by the next level. This method supports as many page sizes as the number of levels of Paging. This feature makes it possible to omit intermediate-level page tables and to use larger-size pages, and decreases both the processing overhead and the total table size as Short-Circuit Segment Tree does.

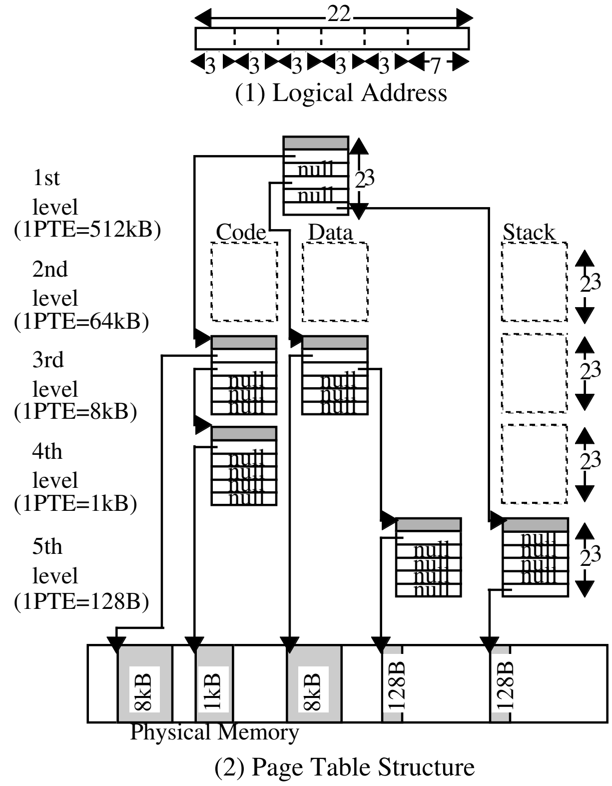

Figure 3. Intermediate-level Skip Multi-Size Paging

Figure 3 shows the page table structure of Intermediate-level Skip Multi-Size Paging. In the case of logical address format of 5-level Paging as shown in Figure 3 (1), we show an example of the page table structure of this method in Figure 3 (2). This example shows that the second through fourth-level page tables of the stack area are skipped and the first (top)-level PTE links directly to the fifth (bottom)-level page table.

Figure 4 (1) shows the page table of each level. Each

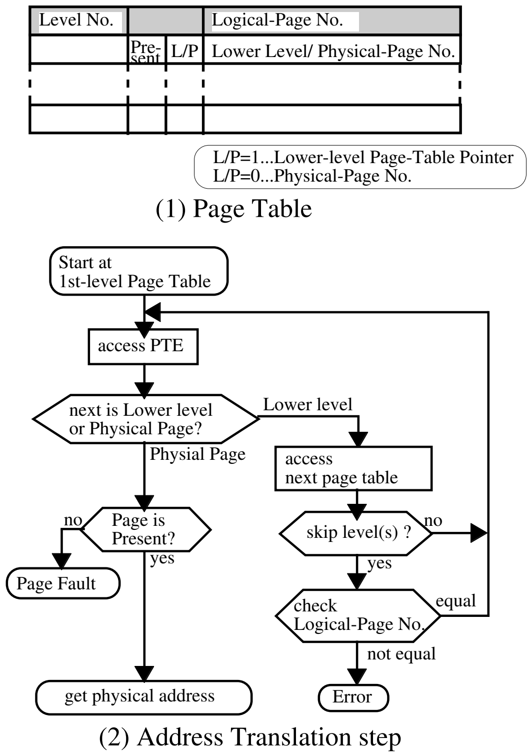

Figure 4. Page Table and Address Translation Step of Intermediate-level Skip Multi-Size Paging

table contains its level number and its logical page number. Each PTE is almost same as that of PTE of Multi-level Paging except for including one bit (L/P bit in Figure 4 (1)) to indicate which type of information is stored: a pointer to the lower-level page table or a physical page number.

Figure 4 (2) shows the procedure of address translation in Intermediate-level Skip Multi-Size Paging. The memory manager accesses the PTE corresponding to the logical address and checks if the entry has either the pointer to the lower-level page table or its physical page number with L/P bit. If the physical page number is found, it derives the physical address from it. If the pointer to the lower-level page table is found, it accesses the page table. The level number of the page table is first checked to see if one or more levels have been skipped. In case of skipping level(s), it checks if the logical page number in the table equals that of the logical address - this enables Intermediate-level Skip Multi-Size Paging to skip page tables in which only one PTE (not just the first PTE) is used - then repeats the same procedure.

3. Evaluation

In this section, we evaluate the memory consumption and processing overhead of each memory-protection method. We have written evaluation programs which emulate each of the memory-protection methods, and compared their code size and processing overhead. We evaluated Bit Map, Segmentation, Multi-level Paging (3- , 5- and 7-level), Paged Segmentation, Short-Circuit Segment Tree (5- and 7-level) and Intermediate-level Skip Multi-Size Paging (5- and 7-level) mentioned above.

Memory consumption: We compared the memory consumption of memory- protection methods (the code size of a memory-protection procedure + the total table size) under the following conditions:

- Logical address space is 4 MBytes and page size is 128 bytes,
- The code size of kernel is 16 KBytes, the data size is 8 KBytes, and the stack size is 256 bytes,
- The code size of each process is 2 KBytes, the data size is 128 bytes, and the stack size is 128 bytes,
- No code or data sharing between processes,
- Memory size is 128 KBytes and the number of processes is 40.

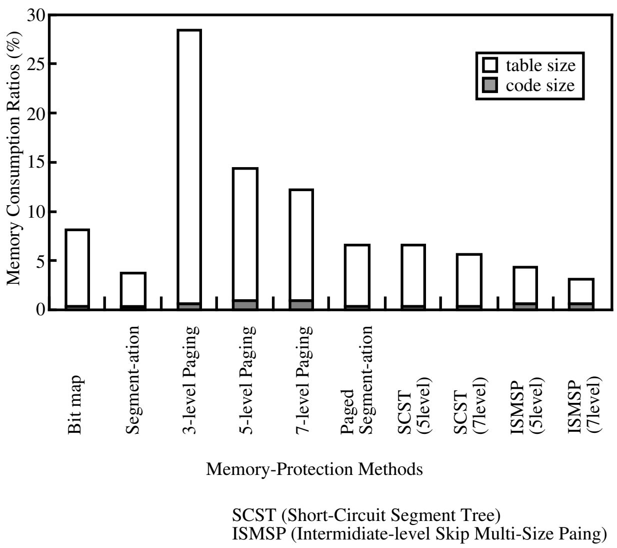

Figure 5. Comparison of the memory consumption

Figure 5 shows the memory consumption of the memory- protection methods. 7-level Intermediate-level Skip Multi-Size Paging consumes the least amount of memory (3.0% of the memory), Segmentation is the second (3.9%), 5-level Intermediate-level Skip Multi-Size Paging is the third (4.5%), 7-level Short-Circuit Segment Tree is the fourth (5.7%), 5-level Short-Circuit Segment Tree is the fifth (6.6%), Paged Segmentation is the sixth (6.7%), and Bitmap is the seventh (8.0%) respectively. Multi-level Paging is the worst (12-28%).

In this evaluation, the code size of each memory-protection procedure is about 0.2-1% of the memory and very small compared with the total table size. The code size could become smaller and negligible in this evaluation if the procedure is optimized and written in assembly language.

Processing overheads: In both Intermediate-level Skip Multi-Size Paging and Short-Circuit Segment Tree, only the top-level and the bottom-level of page tables are accessed (2-level access) in the best-case, while the worst-case overhead is much longer than the best-case overhead because page tables at all levels are accessed, resulting in a variable processing overhead, which is undesirable in real-time systems. In Short-Circuit Segment Tree, with the lowest and the second lowest-level Page Tables, 8-KByte areas (5-level) and 2-KByte areas (7-level) are supported. This is the case of 3-level access (the second best-case overhead). With 3-level access, almost all areas in processes in small embedded systems can be managed. So, the average overhead is almost same as the second best-case overhead. In case of Intermediate-level Skip Multi-Size Paging, by managing areas of processes with almost identical page sizes, almost all areas can be accessed with 2-level access (the best-case overhead). So, the average overhead becomes almost same as the best-case overhead.

Figure 6 shows the processing overheads of the memory-protection methods. The overheads are normalized with respect to the overhead of 3-level Paging which is viewed as a general memory management method. The overhead of Bit Map is the smallest (0.2), Segmentation is the second (0.4), Paged Segmentation is the third (0.7), 3-level Paging is the fourth (1), Intermediate-level Skip Multi-Size Paging (the best-case overhead) is the fifth (1.3), Short-Circuit Segment Tree (the second best-case overhead) is the sixth (1.4), 5-level Paging is the seventh (1.5), and 7-level Paging is the worst (2.0). These overheads are about tens to hundreds of machine cycles, but they could be made smaller if the procedure is optimized and written in assembly language.

Table 1 summarizes the results of our evaluations from which we can make the following conclusions:

- Despite the problem Paged Segmentation and Short-Circuit Segment Tree have regarding the memory consumption and the problem Intermediate- level Skip Multi-Size Paging and Short-Circuit Segment Tree have regarding processing overhead, these memory-protection methods can support multiple threads and region enlargement efficiently and improve the efficiency of memory usage, and are feasible for small embedded systems.
- Bit Map is suitable for small memory, but it is not good for region enlargement and is inferior to the above three methods.
- Segmentation is not good for multiple-thread support and region enlargement.
- Multi-level Paging cannot be used because it consumes too much memory space.

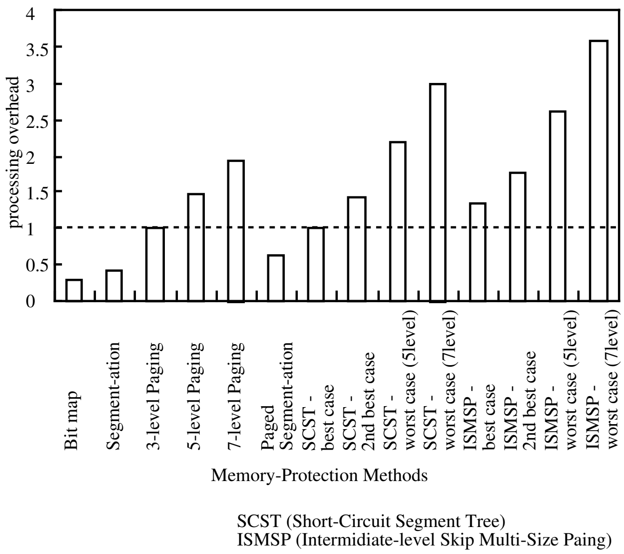

Figure 6. Comparison of the processing overheads

4. Logical address space and page size

So far, we have evaluated the feasibility of several memory-protection methods for small embedded systems assuming a 128- byte page size under a 4-MByte logical address space. In general, the size of logical address space influences the total table size only in paging-based management, not in segmentation-based management. Here, we will focus on paging-based management and describe its effect on memory consumption for different logical address space and page sizes.

Logical address space size:

Items	Bit Map	Segment -ation	Multi-level Paging	Paged Segment-ation	Short-Circuit Segment Tree	Intermidiate-level Skip Multi-Size Paging
Memory Consump -tion	Proportio-nal to memory size. Small in small memory, large in large memory.	Small.	Large to very large.	slightly large.	slightly large.	Small.
Process-ing Overhead	Small.	Small.	Moderate to large.	Small.	Slightly large. Most regions can be accessed in the second best-case overhead.	Slightly large. Most regions can be accessed in the best- case overhead.
Efficient Threads Support	Difficult.	Difficult.	Easy.	Easy.	Easy.	Easy.
Region Enlarge -ment	Difficult.	Difficult.	Easy.	Easy.	Easy.	Easy.
Total Evalua -tion	Not good for region enlarge-ment. Suitable for small memory.	Not good for threads support and region enlarge-ment.	Consumes too much memory. Cannot be used.	Suitable for small embedded systems.	Suitable for small embedded systems.	Suitable for small embedded systems.

Table 1. Comparison of feasibility between memory protection methods

The larger the logical address space, the more memory the protection method consumes. In case of Short-Circuit Segment Tree and Intermediate-level Skip Multi-Size Paging skipping the intermediate- level page tables of Multi-level Paging, we can keep memory consumption small by making the page table size of the layer always skipped (e.g., the second level page table) very large.

By making the logical address space much smaller, we can reduce memory consumption to some extent. Table 2 shows the minimum total table size per process in 1- to 7-level Paging. It also shows the table size both in 4-MByte (22 bits) logical address space and 128- KByte (17 bits) logical address space. Table 2 shows that the effect of the logical address space shrinkage to decrease the total table size is rather large on a small number of levels, but it becomes small in case of a large number of levels. The shrinkage of the logical address space also limits the size of physical memory to be used.

Page size:

In general, increasing page size decreases TLB miss ratio and the memory space occupied by page tables, and improves the processing speed, but the efficiency of memory usage gets worse because of internal fragmentation. On the other hand, a small page size increases memory consumption and TLB miss ratio, and decreases the processing speed, but the efficiency of memory usage improves. The page size is inversely proportional to the total size of PTEs necessary for directly managing all of the physical memory. For example, for page size of 128 bytes, the total PTEs consume 3% of physical memory. On the other hand, for page size of 32 bytes they consume 12%, while for page size of 1 KBytes they consume just 0.4%. The best page size depends on the application-program size, memory size and the cost/performance requirements.

Logical Address Space	1-level	2-level	3-level	4-level	5-level	6-level	7-level
4MBytes	131072	2560	896	608	416	368	320
128KBytes	4096	512	320	224	208	208	208

(unit: bytes)

Table 2. The total page size per process in Multi-level Pagings

Almost all of the above evaluation results should apply to the case when the ratio of page size to logical address space is the same as that of 128-byte page size to the 4-MByte address space considered in this paper except for the physical memory space occupied by PTEs.

5. Hardware implementations

Today, many microprocessors support hardware- managed TLBs, while some micro-processors like MIPS R3000[6] support software-managed TLBs generating a trap to the operating system when a TLB miss occurs. In general, the processing overhead of memory- protection procedure is up to ten machine cycles in the former, while tens to hundreds of machine cycles in the latter. But in order to implement MMUs in microprocessors for small embedded systems, it is desirable to support the latter because it can simplify the hardware, keep its cost low, and also provide the flexibility to support paging-based managements mentioned in this paper. Design tradeoffs for software-managed TLBs are studied by the authors of [12].

With a MMU supporting 128-byte pages, it is possible to implement Paged Segmentation and Short-Circuit Seg-

[6] R3000 is a registered trademark of MIPS Technologies, Inc.

ment Tree. For Intermediate-level Skip Multi-Size Paging, it can be implemented by calculating 128-byte physical page numbers in the memory-protection procedure supported by the operating system. The processing speed of an application program is affected not only by the processing overhead of the memory-protection method but also by the frequency of TLB misses (which depends on the access locality of the program), the number of TLB entries and TLB entry caching method. It is difficult to estimate processing speed precisely and is beyond the scope of this paper.

Next, we study the structure of MMU which is suitable for each memory-protection method. For segmentation-based memory- protection methods (Paged Segmentation), the MMU must support the function to check if an accessing address is within segments, making the MMU hardware complex and costly. For Intermediate-level Skip Multi- Size Paging, if the TLB supports all page sizes of this memory-protection method, the frequency of TLB misses may be rather small and this can increase the processing speed dramatically. For example, if 1-KByte page size as well as 128-byte page size are supported, only one TLB miss occurs while executing 1-KByte code. On the other hand, if only a 128- byte page size is supported, many TLB misses may occur. For example, when the code is executed sequentially from lower address to higher address, 8 TLB misses occur. This simple example indicates the importance of supporting all page sizes of this memory-protection method. The authors of [13] observed the fact that supporting multiple page sizes will generally improve processing speed, especially with a fully- associative TLB. However, the MMU hardware supporting multiple page sizes with a fully-associative TLB is complex and costly, making it difficult to use for cost-sensitive embedded systems. But this feature may be cost-effective if the number of TLB entries in the MMU is decreased.

6. Conclusion

In this paper, we studied memory-protection methods for small embedded systems. Paged Segmentation, Intermediate-level Skip Multi-Size Paging and Short-Circuit Segment Tree are most suitable in terms of multiple threads support, region enlargement, processing overhead, and memory consumption. The feasibility of Intermediate-level Skip Multi-Size Paging can also be improved with a MMU which supports several page sizes.

Future work includes study of more detailed functions of the MMU and evaluation of each memory-protection method by measuring the improvement in the performance of a microprocessor.

7. Acknowledgment

The authors would like to thank Khawar Zuberi for his valuable comments and suggestions on an earlier version of this paper.

References

[1] K. M. Zuberi and K. G. Shin, "EMERALDS: A Microkernel for Embedded Real-Time Systems", *Proceedings of IEEE Real-Time Technology and Applications Symposium*, pages 241-249, June 1996.

[2] *M68040 User's Manual*, Motorola, 1993.

[3] *PowerPC 602 RISC Microprocessor User's Manual*, Motorola, 1995.

[4] S. Gorman, Overview of the Protected Mode Operation of the Intel Architecture, *Intel Architecture-Paper*, Available via http://www.intel.com/design/intarch/papers/EXC_IA.PDF.

[5] P. J. Denning, "Working Sets Past and Present", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, pages 64-84, January 1980.

[6] P. J. Denning, "Virtual Memory", *ACM Computing Surveys*, vol. 2, no. 3, pages 153-189, September 1970.

[7] F. Mueller, V. Rustagi, T. P. Baker, "MiThOS — A Real-Time Micro-Kernel Threads Operating System", *Proceedings of the IEEE Real-Time Systems Symposium*, pages 49-53, December 1995.

[8] V. Abrossimov, M. Rozier, M. Shapiro, "Generic Virtual Memory Management for Operating System Kernels", *Proceedings of the 12th ACM Symposium on Operating System Principles* (SOSP '89), December 1989.

[9] LynxOS — Hard Real-Time OS Features and Capabilities, Available via the Lynx home page at http://www.lynx.com.

[10] P. Resenfeld, Using 80x86 Segmentation to Improve Software Reliability, Available via the Microtec home page at http://www.mri.com.

[11] J. S. Shapiro, A Programmer's Introduction to EROS, Available via the EROS home page at http://www.cis.upenn.edu/˜eros.

[12] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, R. Brown, "Design Tradeoffs for Software-Managed TLBs", *Proceedings of IEEE/ACM 20th Annual International Symposium on Computer Architecture*, pages 27-38, May 1993.

[13] M. Talluri, S. Kong, M. D. Hill and D. A. Patterson, "Trade-Offs in Supporting Two Page Sizes", Sun Microsystems Laboratories Technical Report, February 1993.

[14] A. S. Tanenbaum, Modern Operating Systems, chapter 3, Memory Management, Prentice-Hall, 1992.