Software Virtual Memory Management for MMU-less Embedded Systems

Siddharth Choudhuri

Tony Givargis

Technical Report CECS-05-16

November 6, 2005

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-3425, USA

(949) 824-8168

{sid, givargis}@cecs.uci.edu

Abstract

For an embedded system designer, the rise in processing speeds of embedded processors and microcontroller evolution has lead to the possibility of running computation and data intensive applications on small embedded devices that earlier only ran on desktop-class systems. From a memory stand point, there is a similar need for running larger and more data intensive applications on embedded devices. However, support for large memory adadress spaces, specifically, virtual memory, for MMU-less embedded systems is lacking. In this paper, we present a software virtual memory scheme for MMU-less systems based on an application level virtual memory library and a virtual memory aware assembler. Our virtual memory support is transparent to the programmer, can be tuned for a specific application, correct by construction, and fully automated. Our experiements validate the feasibility of virtual memory for MMU-less embedded systems using benchmark programs.

Contents

List of Figures

Software Virtual Memory Management for MMU-less Embedded Systems

Siddharth Choudhuri, Tony Givargis

Center for Embedded Computer Systems

University of California, Irvine

Irvine, CA 92697-3425, USA

{sid,givargis}@cecs.uci.edu

http://www.cecs.uci.edu

Abstract

For an embedded system designer, the rise in processing speeds of embedded processors and microcontroller evolution has lead to the possibility of running computation and data intensive applications on small embedded devices that earlier only ran on desktop-class systems. From a memory stand point, there is a similar need for running larger and more data intensive applications on embedded devices. However, support for large memory adadress spaces, specifically, virtual memory, for MMU-less embedded systems is lacking. In this paper, we present a software virtual memory scheme for MMU-less systems based on an application level virtual memory library and a virtual memory aware assembler. Our virtual memory support is transparent to the programmer, can be tuned for a specific application, correct by construction, and fully automated. Our experiements validate the feasibility of virtual memory for MMU-less embedded systems using benchmark programs.

1 Introduction

Embedded ubiquitous devices have revolutionized our day to day life. Most of these devices are based on microcontrollers and low end processors. The rise in use and variety of such embedded systems is driven by the fact that microcontrollers and low end embedded processors are cheap, easy to program using well defined programming models, and provide great flexibility in design and function of embedded systems.

According to the semiconductor industry association [19], the global microcontroller market, driven by consumer and automotive applications, will reach $13.5 billion by 2007. It is estimated that in five years, the number of processors in the average home could grow from 40 to 280 and the number of embedded chips sold to support increasingly intelligent devices could grow to over nine billion [19]. In fact, microcontrollers account for more than 97% of total processors sold [6].

While microcontrollers were initially designed for small control based applications, driven by Moore's law, this class of processors have evolved from tiny 4-bit systems running at a few KHz to more complex 8/16-bit systems running at a few MHz, as well as some 32-bit systems running at 100s of MHz [18].

For an embedded system designer, this steady rise in processor speed has lead to the possibility of running complex and computationally intensive applications on small embedded devices that could earlier run only on desktop-class systems. From a memory stand point, there is a similar need for running larger and more data intensive applications on embedded devices. However, support for large memory address spaces, specifically, virtual memory, for low end embedded processors and microcontroller-based systems is lacking [4].

Virtual memory is a scheme that provides to the software layer an illusion of a very large contiguous memory address space (i.e., one that is as large as the secondary storage) at an ideal speed of accessing RAM. Virtual memory, in addition to providing applications larger-than-RAM memory, enables abstractions such as multitasking (e.g., multiple processes accessing disjoint and large address spaces) and protection (e.g., prohibiting one process from accessing another process's data). In most cases, virtual memory is implemented as an operating system (OS) feature [7][20].

The implementation of virtual memory in an OS is heavily based on the underlying processor architecture, specifically, the memory management unit (MMU). MMU is a hardware unit associated with the core processor that provides support for virtual memory addressing. However, unlike high-end processors, microcontroller and a large class of low end embedded processors are MMU-less. Table 1 lists a few such MMU-less embedded processors [25, 8, 2, 14, 1]. As a result, OS supported virtual memory for such processors, for the most part, is absent [4]. Thus having processing power does not necessarily translate into the ability of running data-intensive programs unless process address space issues, specifically, virtual memory is taken care of.

In order to support data-intensive programs that do not fit into available RAM, embedded programmers

end up reinventing hand coded paging techniques. Such adhoc techniques are time consuming, complex and non-portable at best. An incorrect implementation can lead to serious bugs that are difficult to track in the process of address translation.

In this paper, we present software virtual memory management schemes for MMU-less processors. Our proposed approach presents the programmer with an abstraction of virtual memory with little or no modifications required to the program. Our approach gives the programmer a view of contiguous memory that is as large as secondary storage. We present three schemes that differ from each other in terms of address space virtualized, performance overhead, and transparent access to the virtual memory system by the high level application. Using our scheme, the application program is compiled and passed as input to *vm-assembler* (virtual memory aware assembler). The vm-assembler, in addition to converting assembly to binary, inserts additional code that implements the virtual to physical address translation at runtime. Thus, the binary executable has a self-contained software implementation of a traditional MMU. Besides giving the programmer a view of large available memory, virtual memory also lays foundation on which schemes like protection, multitasking, dynamic memory allocation and sharing can be built. We show the efficacy of our work by running a set of benchmark applications targeted for a MIPS processor, compiled using mips-gcc compiler, and the proposed vm-assembler. Our scheme can be tuned to be application specific, it is correct by construction and automated.

Table 1: MMU-less Processors

Processor	**Manufacturer**	**Specs**
MPC52XX (Coldfire)	Freescale	40-60 (MHz)
MCF52XXX	Freescale	16-166 (MHz)
MPC5200	Freescale	266-400 (MHz)
Microblaze	Xilinx	125 MIPS
i960	Intel	16-40 (MHz)
ARM7TDMI	ARM	60 (MHz)
NEC V850	NEC	20-100 (MHz)
PIC18XXX	Microchip	30 (MHz)
ADSP-BFXXX	Analog Deices	200-750 (MHz)

2 Related Work

Compiler directed virtual memory for low end embedded systems has been proposed by [15]. The proposed approach can only be used for code section of the program. In a lot of cases, the data segment of a program

is larger than code segment, attributed to size of data-structues. Also, code segment in low end embedded systems is sometimes hardwired (in ROM/EPROM) and hence cannot be paged. Another drawback in [15], is that the amount of program paged depends on the function size. This could lead to varying page sizes that depend on function body leading to an inefficient implementation. In our work, we have proposed techniques to page data segment of a program, which has a greater need to be paged, than the code segment which is fixed and known apriori. The granularity of page size is determined by the programmer and is not fixed. Thus the programmer can customize the page size that works best for the given application in hand.

Softvm is an approach proposed by [9] that provides virtual memory support for MMU-less systems. However, the work requires presence of a virtual cache, a hardware mechanism supported by the processor to invoke a "software cache miss handler" and a "mapped load" instruction supported by the processor that converts virtual address to physical address. Our approach does not make any assumption about the underyling processor architecture.

Application level virtual memory (AVM) has been proposed in [5]. While providing the virtual memory abstraction at user level, this work is based in the context of extensible operating systems and assumes the presence of an underlying MMU. Virtual memory support in embedded operating systems is lagging for MMU-less systems. QNX[16] and WindowsCE[23], provide limited virtual memory support for processors equipped with MMU. uClinux[22], a Linux derivative for MMU-less processors does not provide any support for virtual memory[4]. While eCOS[17], an embedded OS, can be compiled for MMU-less processors, doing so disables virtual memory support. Other popular OSs, namely, VxWorks[24], Hard Hat Linux[13], and LynxOS[10] have support for virtual memory but can only be ported to processors with MMU support.

3 Technical Approach

3.1 System Architecture

Figure 1 shows the overall system architecture. Figure 1(a) depicts the target system consisting of a MMU-less MIPS R3000 processor simulator, having a fixed amount of RAM. The chracterstics of RAM (or local memory) is that it is fast (in terms of access time) and small (in terms of size). The processor is connected to a secondary storage device using an I/O interface. Two types of I/O interfaces are possible, namely, serial and parallel. The characteristics of secondary storage is that it is slow (in terms of access time) and large (in

terms of size). For our experimental results, we have considered two kinds of secondary storage, namely, EEPROM and Flash. The target system can have different architectures depending on the combination of I/O interface and the type of secondary storage. Figure 1(b) shows our design flow. The inputs to the design flow are:

1. The application source code. Note that, application's view of address space is as large as the secondary storage i.e., the virtual address space.

2. The virtual memory library. This library consists of an implementation of virtual to physical address translation (`vm.c`). It also includes a header file (`vm.h`) with configurable parameters (page size, ram size), details of which are explained later in this section.

We use mips-gcc (version 3.2.1) to compile the application, along with virtual memory library, and generate an assembly output. The generated assembly code, serves as the input to the vm-assembler. The vm-assembler inserts additional code that is responsible for runtime virtual to physical translation. The generated binary, directly runs on our MIPS instruction set simulator (i.e., without the support of any underlying operating systems).

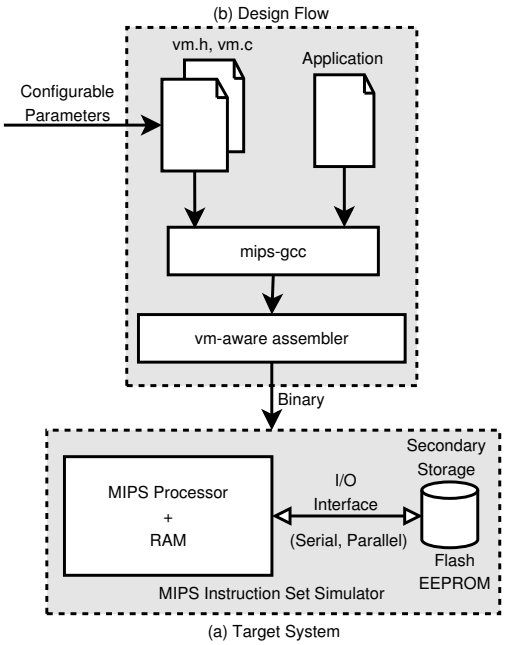

Figure 1: Overall Design Flow

The virtual memory sub-system has two configurable parameters, namely, RAM size *S* (i.e., the amount of RAM available for caching of virtual memory pages) and page size *P* which determines the size of an indivisible block of data that is stored/loaded from secondary storage at any given time.

3.2 Methodology

The underlying idea behind any virtual memory scheme is that compilers generate code with memory operations (loads and stores) for a virtual address space. A virtual address, generated out of the processor during runtime, is translated to a valid, physical address. In systems that have an MMU, this translation is done by the MMU at runtime. In our scheme, we provide this translation in software. The vm-assembler, described in the previous section, intercepts memory operations in the assembly code (loads and stores) and replaces them by a call to a *virtual-to-physical* translation function (from vm library), invoked during runtime. Before describing the translation algorithm, we present the system memory organization in Figure 2.

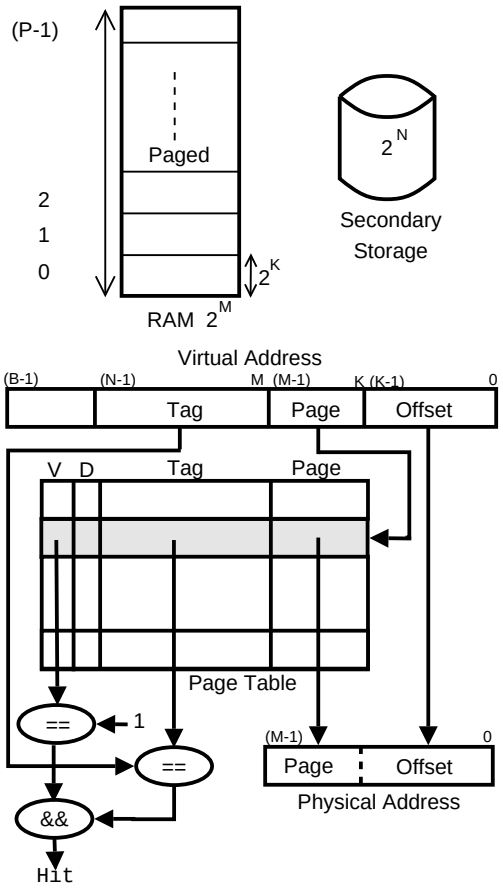

Figure 2: Memory Layout

Consider a system with secondary storage equal to 2^N bytes and 2^M bytes in RAM that is paged. A page size is equal to 2^K bytes. Thus, we have $P = 2^M/2^K$ pages of RAM available for paging. We use a page table for address translation lookup. With our scheme, a page can map only to one unique entry in page table (direct mapped). Thus, we have P entries in the page table. Each page table entry (PTE), has the following information:

1. $Page = log_2(M-K)$ bits, points to a unique page in RAM i.e., $Page \in \{0,1,2,...,P-1\}$.
2. $Tag = log_2(N-M)$ bits, is used for comparing tag generated out of the virtual address.
3. V the valid bit, if set, indicates that the PTE information is valid.
4. D the dirty bit, if set, indicates that the page pointed to by the PTE, has data that has been modified (written).

The processor generated virtual address consists of B-bits (If the secondary storage is large enough, we could have $B = N$). This address is broken down into offset(K least significant bits), page ($M-K$ bits) and tag ($N-M$ bits). The page bits of virtual address point to an entry in the PTE. The virtual address tag bits are compared to the tag field of this PTE. If the valid bit is set and the tag bits match, we have a hit (i.e., the physical address is in main memory). The actual physical address is computed by concatenating offset bits of virtual address with "page" entry of the PTE.

There can be cases other than a hit, depending on the result of comparing tags, valid bit and dirty bit. These are described in the virtual-to-physical translation algorithm. This algorithm is similar to a direct mapped address translation used in traditional operating systems [7][20]. Note that the virtual memory library, mentioned in previous section, has the implementation of 1.

3.3 Virtual Memory Approach

We now present three approaches to providing virtual memory. The approaches differ from each other in terms of (1) extent of memory mapped to virtual address space, (2) performance trade-offs (in terms of execution cycles) and (3) transparency provided to the programmer by the virtual memory subsystem. Experimental results from the three approaches are provided in the next section.

Algorithm 1 Virtual to Physical Translation

```
1: function virtual-to-physical
2: Input va : virtual address, wr : 1 ⟹ write, 0 ⟹ read
3: Output pa : physical address
4: Decompose va into ⟨tag, page, offset⟩
5: pte ← PTE[page]
6: if pte.tag = va.tag and pte.valid = 1 then
7:     if wr = 1 then
8:         pte.dirty ← 1
9:     end if
10:    return pa = page× sizeof(page) + offset
11: end if
12: if pte.valid = 1 and pte.dirty = 1 then
13:    write RAM[page] to secondary store
14: end if
15: Read newpage|va ∈ newpage from secondary store
16: Update pte.tag, pte.page, pte.valid
17: if wr = 1 then
18:    pte.dirty ← 1
19: end if
20: return pa = page× sizeof(page) + offset
```

3.3.1 Approach 1 - Pure VM

The first approach mimics a system with hardware MMU. Every memory access in the application is in a virtual address space that is translated to physical address during runtime. This approach is transparent to the application. The drawback however, is that, every memory access is virtualized resulting in a call to the virtual-to-physical function as many times as there are memory (load/store) instructions in the program.

3.3.2 Approach 2 - Fixed Address VM

In this approach, a region of the memory is marked as virtualized. Any memory access (load/store) that belongs to this marked region is translated. This approach requires the programmer to indicate to the vm-assembler the region marked as virtual. As opposed to the previous approach, in this case, the overhead of translation from virtual to physical address is reduced to only the memory region marked as virtual. This however, requires a runtime check to be made at every load/store to determine if the address is virtualized. This is achieved by modifying the vm-assembler so that it inserts code that does runtime check on every memory access and translates only those addresses that are virtualized. In our experiments, we tested this approach by marking all the data region belonging to global variables as belonging to virtual address space.

3.3.3 Approach 3 - Selective VM

Selective VM is similar to the previous approach, but is more fine-grained in terms of memory that is virtualized. Note that in the previous approach, a runtime check was required on every memory access to determine if the address is virtualized. Selective VM avoids this runtime check overhead by annotating data structures at source level. It requires the programmer to tag individual data structures as belonging to virtual address space (as opposed to an entire region). This annotation is done at variable declaration, using a `#pragma` directive. Any use or def of annotated data structure in the source is modified to a call to the virtual-to-physical function. This approach significantly reduces the runtime overhead by restricting the translation only to large data structures that can reap benefit out of virtualization. It gives the embedded programmer more control on what is virtualized. However, this approach is the least transparent to the application programmer compared to the other two approaches.

4 Experiments

In this section, we describe our experimental setup followed by results from the approaches described in previous section.

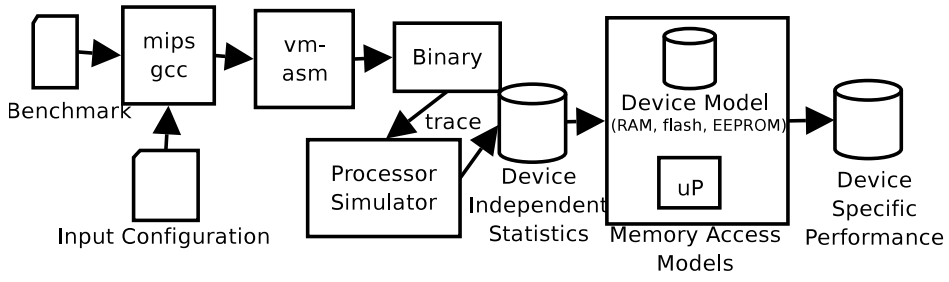

Figure 3: Experimental Setup

4.1 Experimental Setup

Figure 3 depicts our experimental setup and flow. Our experimental setup consists of a virtual memory aware assembler, the target processor simulator, the RAM and the secondary storage device (i.e., Flash and EEPROM) simulators. Four architectures of the target system are simulated. The first two architectures consist of a processor connected to either Flash or EEPROM secondary storage devices over a serial interface.

The next two architectures consist of a processor connected to either Flash or EEPROM secondary storage device over a parallel interface.

The inputs to the system is a C source file (benchmark) and virtual memory configuration parameters, namely, page size *P* and RAM size *S*. In our experiments, we have considered page size vlaues of $P = (32, 64...512)$ bytes. Further, in our experiments, we have considered RAM size (amount of main memory available for caching secondary storage) $S = (256, 512...16384)$ bytes. The program is compiled into assembly and then passed through our vm-assembler. The output of vm-assembler is executed in the context of a MIPS instruction set simulator. The device independent statistics (reads, writes, misses, hits) are then passed through our memory access models to obtain device speicific performance values, namely, the *average memory access time* for application. Our memory access models are based on datasheet values [3][21][12] for clock speed, bus speed, and secondary storage access time (Table 2)[1].

Table 2: Access Times

Device	Read (ns/byte)	Write (ns/byte)
SRAM	70	70
EEPROM (Serial)	800	39062
EEPROM (Parallel)	260	78125
Flash (Serial)	25	42968
Flash (Parallel)	23	390

We used benchmark programs from powerstone [11] suite for our experiments. Specifically, the benchmark programs include: *adpcm*, a 16 bit PCM to 4 bit ADPCM coder; *bcnt* a bit shifting and anding program; *crc*, program that performs cyclic redundancy check; *des* a standard data encryption algorithm; *fir* integer finite impulse response filter; *g3fax*, a group three fax decoder; *jpeg*, an implementation of the JPEG image decompression standard; *pocsag* paging communication protocols and *v42*, a modem encoding/decoding algorithm. Table 3 summarizes the total number of bytes marked for virtual memory in case of selective VM (approach 3), as described in section §4.3.3.

4.2 Experimental Results

Ideally, a program will have the fastest execution time if there is enough RAM available to fit the runtime needs of the program. This case, although possible, is not practical due to the high cost requirements of having large amounts of RAM. However, we use this case as a *lower bound* for our purposes (i.e., any

[1]The values shown are averaged for per byte access time

Table 3: Benchmark Memory Access Pattern

Benchmark	Read	Write
adpcm	1028	400
bcnt	8448	0
crc	1280	1280
des	4096	128
fir	140	136
g3fax	4096	1728
jpeg	154801	154201
pocasg	256	656
v42	23938	8192

program running in the context of virtual memory cannot be faster than a program having sufficient RAM available to fit its run-time requirements). Similarly, the case of having to run a program entirely off the secondary storage is not feasible and results in the worst case execution time. We use this case as our *upper bound*. Virtual memory implementation tries to achieve a performance that is as close as possible to the lower bound, while not imposing the limitations of having large amounts of main memory to fit a program's run-time requirements.

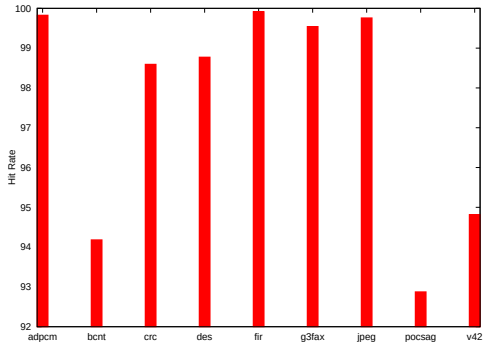

Figure 4: Hit Rate - Pure VM

Figure 4 shows the memory access hit rate of benchmark programs using pure VM (approach 1), as described in section §4.3.1. The hit rates are averaged over all combinations of ram size S, and page size P (i.e., $S \times P$). Similarly, Figure 5 shows the memory access hit rate of benchmark programs using fixed address VM (approach 2), as described in section §4.3.2. In case of fixed address VM, we observe that not every program results in a high hit rate. Note that, hit rate is calculated as ratio of number of hits to number of accesses. Thus, in case of fixed address VM, having large number of accesses to data that resides in virtual address space would help increase the hit rate. However, for benchmark programs like bcnt, this is

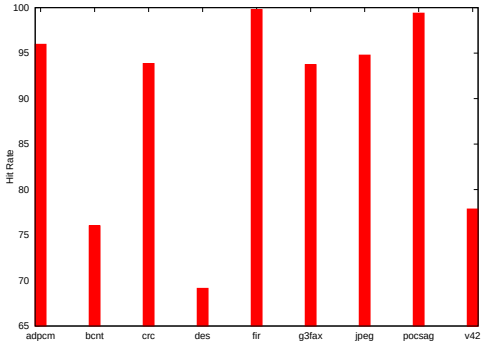

Figure 5: Hit Rate - Fixed Address VM

not the case. It has variables in global memory that are accessed infrequently, leading to a relatively low hit rate. Thus, fixed address VM is highly application specific and depends on application memory access pattern. Not all programs perform well using the fixed address VM approach.

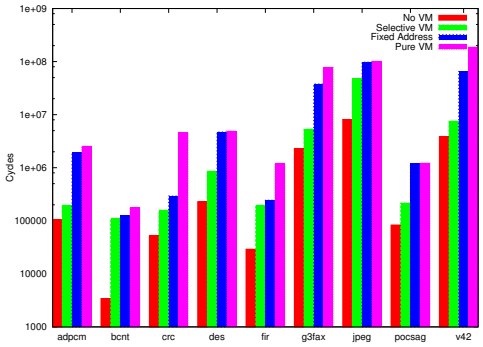

Figure 6: Execution Time Comparision

Figure 6 compares the average execution time of benchmark programs (over all combinations of S and R). The comparision is made against the lower bound case of having a RAM size large enough to fit the runtime requirements of program (labelled as No VM). It can be seen that the pure VM and the fixed address VM approaches lead to execution cycles that is an order of magnitude larger than the selective VM approach. This can be attributed to the fact that in case of pure VM, every memory access results in a call to the virtual-to-physical function. In case of fixed address VM, execution cycles is less than pure VM, but there is additional overhead in terms of checking every memory access to see if the address lies in virtual or physical address space, contributing to the overhead in execution cycles. Out of the three approaches, selective VM results in execution cycles that is closest to the lower bound. Hence for futher results, we only concentrate on selective VM approach, as the other two approaches may not be feasible due to the

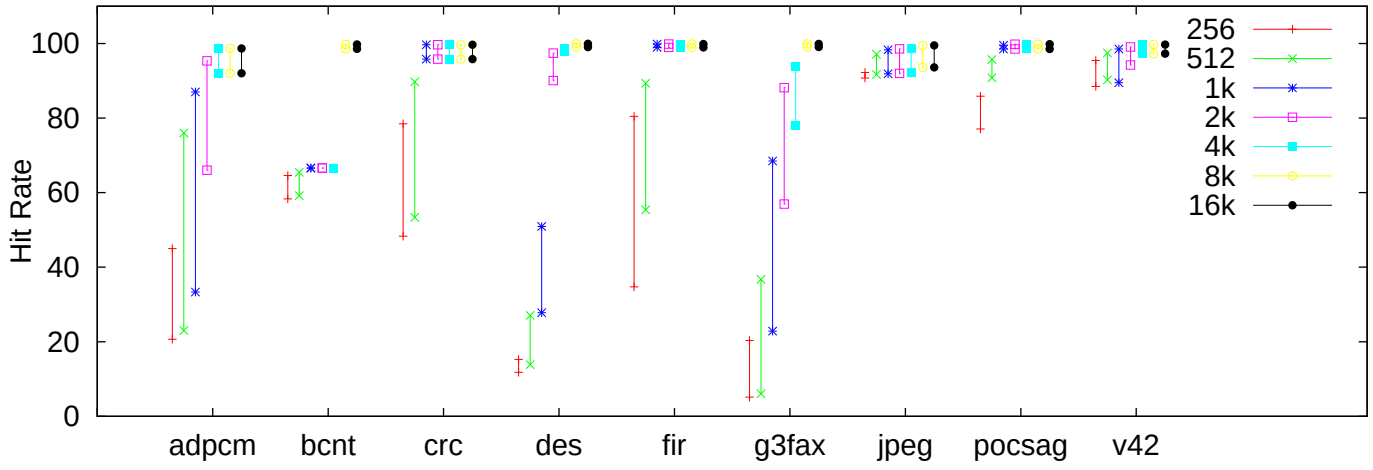

Figure 7: Hit Rate Variation for Benchmarks

overhead in execution cycles. Note that, even though in some cases pure VM has considerable overhead in execution cycles compared to the lower bound; the comparision is being made against a case of having RAM that is large enough to fit the entire program runtime requirements. This may not be possible in case of embedded systems with limited memory and cost constraints (of having large RAM); more so when dealing with programs that have large runtime data memory requirements. The selective VM approach is a trade off between providing virtual memory with reasonable overhead in execution cycles. Apart from providing the programmer with an abstraction of virtual memory, the other goal of this work was to explore various combinations of page size P and RAM size S and study how these configurations affect the performance in terms of hit rate and average memory access time. Figure 7 shows the variation in hit rate for each benchmark program, with all possible page sizes P and all possible RAM sizes S. For each benchmark, multiple vertical bars, each corresponding to one of the possible RAM sizes S, capture the hit rate as a function of each of the possible page sizes P. For example in case of *g3fax* and $S = 2k$, varying page size P yields a hit rate between 56% to 88%. We note that, for some benchmarks, the variation is smaller (e.g., *jpeg* and *v42*) and for other benchmarks this is much more pronounced (e.g., *adpcm* and *g3fax*). Thus, programs can be profiled to come up with an optimal page size. Furthermore, as the amount of RAM size S increases beyond a certain threshold, the variation disappears. Likewise, as the RAM size S increases, we reach a point of *diminishing returns* after which increase in the RAM size S does not result in any significant improvements in hit rate. This is because we reach a point where there are enough free pages available to cache most of the data that is virtualized.

Table 4: Optimal Configurations

	RAM Size S							
	256	512	1K	2K	4K	8K	16k	
adpcm	32	32	32	64	512	512	512	Page Size P
bcnt	32	32	128	128	128	512	512	
crc	32	32	512	512	512	512	512	
des	32	32	32	32	128	512	512	
fir	32	32	512	512	512	512	512	
g3fax	32	32	32	32	32	512	512	
jpeg	128	256	512	512	512	512	512	
pocsag	32	64	128	512	512	512	512	
v42	32	32	64	128	128	128	128	

Table 4 shows the best case configurations corresponding to each of the possible RAM sizes S. These configurations correspond to the points that have maximum hit rate in Figure 7. It can be seen that, though a large page size often yields high hit rates, this case is not always common. This could be due to the fact that, having a large page size does not necessarily mean an entire page worth of data is used to cache (internal fragmentation). In such a case, having smaller, fuller, page sizes can be more useful. As the page size P increases, the penalty of accessing secondary storage also increases, creating a trade-off between improved hit rate versus miss penalty. This trade-off requires device specific exploration of all possible confiugrations which is described next.

Figure 8 shows the average memory access time for each benchmark and each RAM size S with the optimal page size configuration P (i.e., from Table 4). Figure 8 captures this information for four different architectures obtained by using serial vs. parall and EEPROM vs. Flash combinations. Figure 8 also captures the lower bound (i.e., all data in RAM) and worst case (i.e., all data in secondary memory), shown as the two continous plots. The lower and upper bounds presented are not feasible solutions, but they serve as bounds to evaluate our experiments. The average memory access time also shows a trend of *diminishing returns* as seen in case of hit rate. For example in case of *crc*, the average memory access time drops close to the lower bound for $S = 1K$.

Thus, from an embedded systems programmers point of view, applications can be profiled to obtain the right configuration of page size and ram size that leads to maximum benefit.

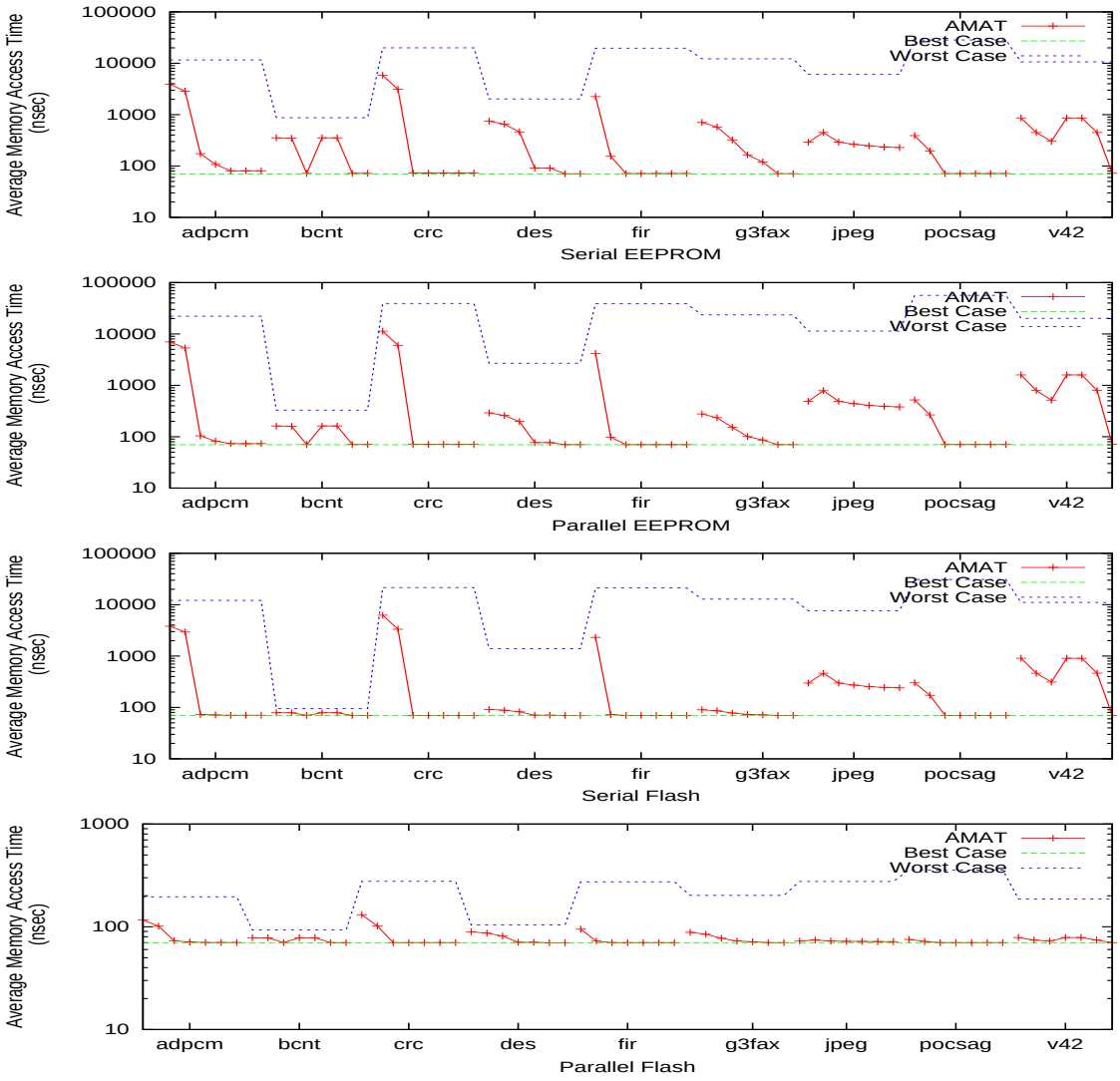

Figure 8: Average Memory Access Time for Benchmarks

5 Conclusion

We have presented a software virtual memory scheme for MMU-less embedded systems. This is achieved using a vm-aware assembler and a virtual memory library. Our approach provides a view of larger than RAM memory to the programmer, tuned for a specific application, robust and automated. The virtual memory system that we presented can be tuned by adjusting two configuration parameters, namely, RAM size and page size. In our experiments, we have explored different configurations for applications drawn from the powerstone benchmark. Our results show that the ideal configuration of a virtual memory system are application dependent. Our experiments validate the feasibility of virtual memory for MMU-less embedded

systems. We also presented three different approaches each with its advantages and disadvantages and found out that having a user defined virtual memory is the right trade-off between achieving virtual memory and performance overhead.

Our future work will focus on optimizing the virtual to physical translation that can lead to reduction in execution time cycles. We also plan to focus on considering additional caching techniques, such as assciative schemes.

References

[1] Blackfin Processor. http://www.analog.com/processors/processors/blackfin/.

[2] ARM. ARM7TDMI. http://www.arm.com/products/CPUs/ARM7TDMI.html.

[3] Atmel. Atmel Corporation. http://www.atmel.com.

[4] M. David. uClinux for Linux Programmers. In *Linux Journal*, July 2000.

[5] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266. ACM Press, 1995.

[6] Gartner. Gartner research. http://www3.gartner.com.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, second edition.* Morgan Kaufmann, San Francisco, California, USA, 1995.

[8] Intel. Intel i960 Processor Overview. http://developer.intel.com/design/i960/family.htm.

[9] B. L. Jacob and T. N. Mudge. Uniprocessor Virtual Memory without TLBs. In *IEEE Transactions on Computers*, volume 50, pages 482 – 499, May 2001.

[10] Lynuxworks. Lynx OS. http://www.lynxworks.com/.

[11] A. Malik, B. Moyer, and D. Cermak. A Lower Power Unified Cache Architecture Providing Power and Performance Flexibility. In *Internationl Symposium on Low Power Electronics and Design*, 2000.

[12] Microchip. PIC18F4320 Device Datasheet. http://www.microchip.com.

[13] Montavista. Hard Hat Linux. http://www.mvista.com.

[14] NEC Electronics. NEC V850E Product Overview. http://www.necel.com/micro/english/v850/.

[15] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min. Compiler-assisted demand paging for embedded systems with flash memory. In *EMSOFT '04: Proceedings of the fourth ACM international conference on Embedded software*, pages 114–124. ACM Press, 2004.

[16] QNX. QNX Software Systems. http://www.qnx.com.

[17] RedHat. ecos. http://sources.redhat.com/ecos/.

[18] W. Schwartz. Enhancing performance using an arm microcontroller with zero wait-state flash. In *Information Quarterly*, Volume 3, Number 2, 2000.

[19] SIA Press Release. Growth for 2004 global semiconductor sales. `http://www.semichips.org/pre_release.cfm?ID=321`.

[20] A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts, sixth edition*. John Wiley and Sons, Inc., 2003.

[21] ST. Stmicroelectronics. `http://www.st.com`.

[22] uClinux. uclinux. `http://www.uclinux.com`.

[23] Windows CE. Microsoft Windows Embedded. `http://www.microsoft.com/windowsce`.

[24] Windriver. VxWorks Real Time OS. `http://www.windriver.com`.

[25] Xilinx. Microblaze Soft Core. `http://www.xilinx.com/microblaze/`.