Single shared memory space architecture for less power

M. Bos, P.J.M. Havinga, G.J.M. Smit
University of Twente
Department of Computer Science
PO Box 217, 7500 AE Enschede, the Netherlands
phone: +31 53 4893730; fax: +31 53 4894590
E-mail: m.bos@cs.utwente.nl

Abstract— **Virtual Memory and Hardware Memory Protection are so common now that they are even used in handheld devices that do not include any secondary storage to swap to. As most handhelds depend on batteries for their power supply, it seems worth investigating an architecture without these expensive pieces of hardware. New type safe languages may allow dropping the memory protection hardware as well. This paper describes the first investigations done in this direction. Both benefits and drawbacks are described and ways to minimize the drawbacks are investigated.**

Keywords—**Memory Management Unit, low-power, Single Memory Space.**

I. Introduction

DUE to historical reasons, most current microprocessors include a *Memory Management Unit* to support virtual memory, memory protection, paging, segmentation, and the like. According to Silberschatz and Galvin [1], all memory-management strategies have the same goal: to keep many processes in memory simultaneously to allow multiprogramming. Memory Management Units and mechanisms like swapping have been introduced to allow the total memory used by all the processes to be more than the size of physical memory.

One observation in the current world of notebook and handheld machines is, when processes regularly outgrow main memory, more memory is bought and put in the machine. Active swapping rarely occurs, simply because performance degradation is so drastic compared to the cost of more memory.

The mechanisms needed for virtual memory and memory protection seem quite expensive when it comes to energy efficiency. Every address used must be translated to a physical memory location. As the mapping from virtual to physical memory addresses must be flexible, memory is often divided into pages and mapping is on a page basis. Page tables are then needed to track the pages. Often these *page tables* are again paged with use of *directory tables*. Main memory access is quite excessive in this architecture, a directory table access, a page table access and finally the operand access for each reference.

To lower main memory load a cache for the translations is added, called the *Translation Lookaside Buffer*. This Translation Lookaside Buffer caches the most recent translations from virtual page numbers to physical page numbers. A fully associative cache is often used. Only when the cache misses, access

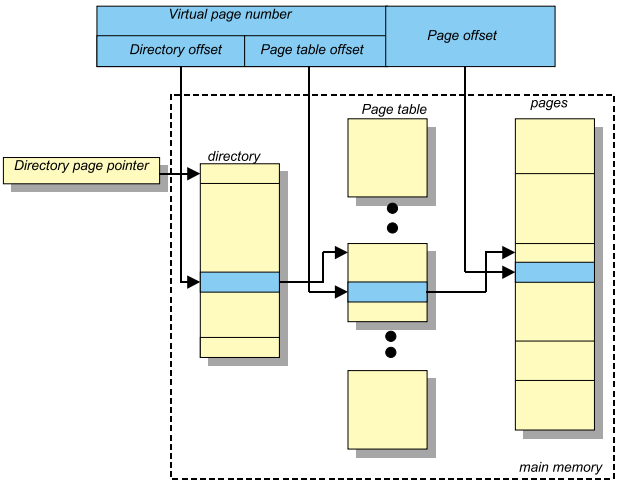

Fig. 1. Example of page table structure [2]

to main memory for the directory table entry and the page table entry is needed.

Besides these mechanisms for virtual memory, mechanisms for memory protection are used. Pages include privilege bits and structures exists to track processes' actual privileges. Registers inside the microprocessor are needed to keep track of all these structures and to allow easy reference to them.

The Operating System must include routines to cope with protection violations and also with *page faults*.

This paper describes the benefits, in terms of possible energy reduction, of leaving these mechanisms out going back to a single shared memory architecture. The main goal is reducing overall energy consumption as focus is on mobile systems.

II. Architecture

An architecture with a single shared memory space, without paging, segmentation and virtual memory is proposed. The main idea why they are not needed is the lack of secondary storage on handhelds. Main memory is simply all there is. Flash might be identified as secondary storage, but when reading flash it can be addressed just like normal memory at about the same speed.

In the proposed architecture there is no need for a complicated Memory Management Unit As there is no need for virtual to physical address translation, no expensive Translation Lookaside Buffer is needed. In common architectures this Transla-

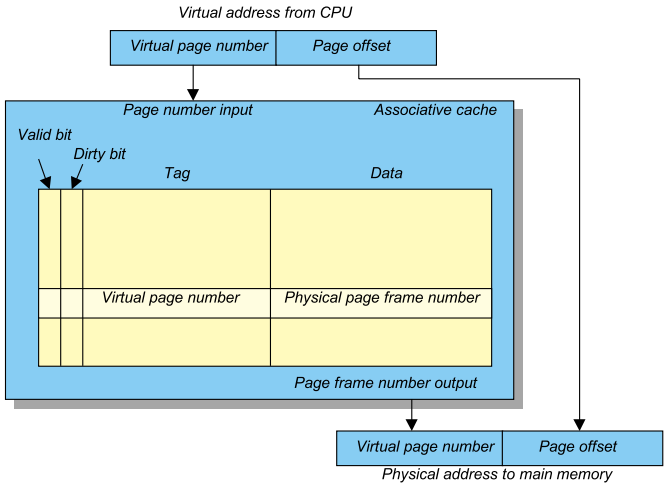

Fig. 2. Example of Translation Lookaside Buffer [2]

tion Lookaside Buffer must provide a virtual to physical address translation for each memory reference, so about every clock cycle. In the proposed architecture only cache-control is necessary, and thus a relative simple Memory Management Unit remains.

Several registers are normally there to allow use and management of mechanisms like paging and segmentation. They are also no longer needed, thus cutting down task switch costs.

A single shared memory space has some more advantages. One is that in Direct Memory Access from peripherals like the Network Interface no worries exist on whether the buffer to write to or read from still exists in main memory or not. Normally there is a chance that the buffer-space is currently swapped to disk. Solutions to overcome this problem, like mechanisms to allow pinning of memory pages or using a buffer in system memory from which data is copied to the user process' memory, all introduce at least a little overhead. Such difficulties like those that arose in implementing the Single-Copy Protocol stack described in [3], simply don't exist in a single shared memory space. Avoiding frequent copying of buffers from one place to another has a significant impact on energy efficiency as data transport is an expensive operation.

Yet another advantage is the ease to share memory, both for code and data, between user processes, without the overhead of the processes each using their own virtual memory space. Normally the Operating System has to keep track of shared parts of memory and map them to the same physical pages for all parties involved in the sharing.

III. Overcoming the drawbacks

Using a single shared memory space introduces some drawbacks too. First of all, how about memory protection? And don't forget memory fragmentation either.

First note that, in a perfect world, no memory protection is needed. If all applications would be bug-free, the memory protection mechanism would never have to take action. That said, and with the knowledge that programs are almost never perfect, a memory protection mechanism that provides enough security, but as simple as possible, must be found.

Current developments in new languages like Java and Limbo [4], offer type safety, which narrows down the possible misbehavior of code quite a bit. One operating system called Inferno [5] only allows programs written in a safe language and compiled with a certified compiler, to be executed. As such, Inferno can run without a Memory Management Unit. Due to the use of a virtual machine in Inferno (called DIS) for code compatibility, performance is somewhat downgraded. User programs are compiled into virtual machine instructions and either interpreted or compiled into native code during runtime. But there is no fundamental reason for compilers to compile languages like Java and Limbo into virtual machine bytecode. Java to native-code compilers already exist [6].

Type safe languages can filter out most possible memory protection violations during compile time. What remains for runtime are bounds checking Taking care of most possible memory protection violations during compile time allows for a more energy-efficient runtime system. Certificates bound together with code to prove origin and compliance are already getting more accepted through continuous development of the Internet [7]. This principal can be extended towards the Operating System, which can deny running of any program that is not certified by a proven compiler.

Type safe languages with runtime bounds checking actually provide more protection than regular systems with Hardware Memory Protection do. In a regular system, processes are protected from each other by running them in separate memory spaces. If a program tries to access memory outside its memory space, the Memory Protection Hardware signals the Operating System. This mechanism does not trap programs that exceed structure boundaries as long as the memory referenced is claimed by other structures of the same program. A type safe language with runtime boundary checking provides more fine grain protection as each single structure within a program is protected.

The proposed application software environment does not provide protection for device drivers and other native code written for optimal performance. Device drivers usually work in kernel mode thus they are able to bypass memory protection in a normal architecture also. Drivers are of vital essence to make a device work. If their code would be erroneous, the system wouldn't be able to provide full functionality anyway.

One of the benefits of memory protection used on current machines is that, when applications malfunction, only the malfunctioning application crashes and other programs continue as usual. The same happens in the proposed architecture when runtime checks generated by the compiler find problems. Also, the number of possible errors is smaller as program are written in a type safe language.

In the case that bugs exist in native libraries, which may be still used for optimal performance, the whole system might now be affected. Apart from the fact that on a handheld only one or a few applications will run at once, this pitfall is alleviated by the fact that it can only occur when bugs exist in the few carefully handcrafted native libraries.

IV. PRACTICE

Overhead due to runtime checks needed to ensure programs behave correctly has been measured in Inferno. The two most important checks done at runtime are reference counting and array-boundary checking.

In Inferno, all boundary checking is done within only a few of the virtual machine instructions. These are the special instructions (one for each basic type, and one for structures) that get the address of an entry inside an array. To allow measurements, the kernel has been patched to provide counting of all different instructions executed per process. These are logged over the network together with a tree that shows the parent-child relations of all processes that have existed during the uptime of the machine. By accumulation of the counters for all child processes of the main process under test, a feeling for the amount of overhead due to boundary checking can be found for different types of programs.

Measurements have been taken for a number of different programs. A software implementation of the GSM 06.10 speech transcoder algorithm [8], an AVI video player, the Brutus text editor and common shell commands have been profiled.

TABLE I
BOUNDARY CHECKING PROFILES

Application	total instr.	bounds instr.	%
GSM Transcoder	2102223	296675	14
Shell commands	2763528	321229	12
AVI player	1210020	6413364	19
Brutus (Text Editor)	104432	6791	7

From table I it is clear that a significant amount of the instructions executed include bounds checking. To get a better picture of the overhead, the internals of these instructions must be examined. To get the address of an element at index *i* of an array, *i* times the size of each element must be added to the base address of the array. Two memory references are needed to get the array and *i*. In Inferno, the base address is indirect so must be fetched from memory also.

Thus, to get the address, a multiplication, an addition and three memory fetches are needed.

The additional bounds checking consists of checking the address of the array to be non-zero and of checking the (unsigned) index *i* to be smaller than the array-length *len*. The array address is already fetched from memory so can be checked from a register, but *len* must be fetched from memory. Thus, overhead consists of one more memory fetch and two comparisons.

As comparisions are generally cheaper than multiplications and additions, at most 40 percent of the work in instructions that do array indexing consists of bounds checking.

If all virtual machine instructions are treated equal, this leads to the bounds checking overhead numbers in table II. From this table, as far as only bounds checking is considered, leaving out the Memory Management Unit is always more efficient, only if, this unit accounts for more than 8 percent of the total microprocessor's energy consumption.

This number is rather rough of course, as not all virtual machine instructions have the same code size and time is spend in the kernel code as well, and kernel code does not include any bounds checking. But the 8 percent is a lower bound to start with. 8 percent seems rather high, so it seems worthwhile to look into possible optimizations, which is done in the next section.

TABLE II
BOUNDARY CHECKING OVERHEAD

Application	%
GSM Transcoder	6
Shell commands	5
AVI player	8
Brutus (Text Editor)	3

A. GSM Speech transcoder code in depth

After close inspection of the code for the GSM Speech transcoder, it showed that for this particular implementation most array references are within for-loops and the used indexes are as simple as *[i]* or *[i+a]* where *a* is set outside the for-loop. This leaves room to optimize the compiler to create code that checks for possible boundary crossing only once every time before entering the for-loop, and if possible, generates an optimized for-loop without checks for each separate reference.

```
code:
for (k = 0; k <= 119; k++)
    drp[ k ] = drp[ 40 + k ];

generates:
for (k = 0; k <= 119; k++)
{
    check drp's bounds to include 40+k;
    get drp[ 40 + k];
    check drp's bounds to include k;
    assign value to drp[ k ];
}

more optimal:
check drp's bounds to include 159;
for (k = 0; k <= 119; k++)
{
    get drp[ 40 + k];
    assign value to drp[ k ];
}
```

Fig. 3. Small for-loop from GSM speech transcoder source

The code shown in figure 3 is a piece of the GSM speech transcoder which is executed for every sample. From this code we can easily determine that the bounds of the array *drp* will not be exceeded if the array has a minimum size of 160 elements. If for this particular piece of code the array *drp* is passed as an argument, and the calling function dynamically allocates the array, its size is not known at compile time. Still, if the compiler would add a statement to check *drp*'s size just before entering the for-loop, there would be no need to do runtime array-bounds checking on any of the 240 references to *drp* inside the for-loop.

The GSM speech transcoder contains a lot of for-loops that index arrays in the same manner. Most of these for-loops are executed one or several times for each sample that stands for 20 milliseconds of speech. Actually, for all array indexing in the few routines that account for over 80 percent of the total instructions executed by the transcoder the arrays are allocated statically and it seems possible to prove at compile time that bounds will never be exceeded. Thus, for the GSM speech transcoder bounds checking overhead could be brought back from 6 percent to about only 1 percent.

By a look at its code, the AVI player program is expected to benefit as much from these optimizations as the GSM Speech Transcoder does. The code of the AVI player is a bit more complicated though, so actual implementation of the compiler optimizations will have to prove this.

To allow for these optimizations, the DIS virtual machine must be extended with instructions that allow array indexing without bounds checking. The compiler should generate these instructions only when it can be proved that bounds cannot be exceeded, or together with appropriate explicit bounds checking instructions outside loops. The addition of these instructions to the virtual machine does not make the system less safe, as in Inferno the Virtual Machine is not safe without cooperation of the compiler in the first place. This was done for added code efficiency. Safety is guaranteed with compiler signatures in the generated code.

V. Conclusions

A combination of a well known architecture from the past and recent developments in programming languages has been studied for energy efficiency. The possible gain seems high enough to continue on this track. Dropping Virtual Memory and Memory Protection gives a simpler architecture and also simplifies interaction between peripherals and the software system. The Operating System complexity is reduced as well.

Overhead is introduced by placing protection in software instead of hardware, but it seems possible to get rid of most of this overhead during compile time. The optimizations used are expected to not only benefit energy efficiency, but may also benefit execution speed of general systems that use type safe languages.

References

[1] A. Silberschatz and P.B. Galvin, *Operating System Concepts*, Addison Wesley, 1994.

[2] M. Morris Mano and Charles R. Kime, *Logic and Computer Design Fundamentals*, Prentice Hall, 1997.

[3] P. Steenkiste, "Design, implementation and evaluation of a single-copy protocol stack.", *Software practice and experience*, 1998.

[4] Lucent Technologies Inc, *The Limbo Programming Language*, 1997.

[5] Sean Dorward, Rob Pike, David Leo Presotto, Dennis Ritchie, Howard Trickey, and Phil Winterbottom, "Inferno", in *Proceedings of the IEEE Compcon 97 Conference*, San Jose, 1997, pp. 241–244.

[6] Diab Data. Inc., "Fastj", 1998, http://www.ddi.com.

[7] Microsoft, "Authenticode white paper", 1998, http://msdn. microsoft.com/workshop/security/authcode/authwp.asp.

[8] ETSI, "Digital cellular telecommunications system (phase 2+); full rate speech; transcoding (gsm 06.10 version 5.1.1)", Tech. Rep., European Telecommunications Standards Institute, May 1998.