AVM: Application-Level Virtual Memory

Dawson R. Engler Sandeep K. Gupta *
{engler,skgupta}@lcs.mit.edu
MIT Laboratory for Computer Science

Introduction

Virtual memory is an important, hard-to-implement, and performance-critical operating system abstraction. Many operating systems implement this abstraction completely in the kernel. These systems are typically large, complex, and inflexible, resulting in bad performance and poor reliability.

To allow some degree of customization and to simplify the kernel, a number of microkernels have put the policy decisions for page-replacement in user-level servers. Unfortunately, many policies in these systems are still hard-coded (e.g., the page-table structure) and the page-servers are often complex and have superuser privilege, making them hard to modify or replace. Other microkernels have chosen to implement a very restricted virtual memory system in order to make the system fast and simple. These systems do not support VM techniques such as copy-on-write, which can improve application-performance by an order of magnitude.

In summary, current VM systems are either complicated and unwieldy, or naive and rudimentary; both approaches penalize applications. To remedy the situation, this paper proposes a novel organization of the VM system, *application-level VM* (AVM), which allows applications to set policies by choosing among different supplied implementations, or even by implementing their own VM system. AVM moves the entire VM system out of the kernel into libraries, simplifying the kernel implementation. Furthermore, the VM system itself is simpler since it does not have to multiplex multiple entities with a wide range of needs, but can instead be specialized for particular domains.

Motivation

The reasons why AVM is important can be classified according to three general principles. First, OS-enforced abstractions of hardware resources hide valuable information. In a system that supports AVM, all information is readily accessible (e.g., TLB exceptions can be monitored). Second, OSs have made poor trade-offs in the past. Since many trade-offs are application-dependent, this leads to poor performance (an obvious example is LRU or MRU page replacement policies). Third, the OS has to be more general and thus more expensive than a specialized and tuned implementation. For example, support for copy-on-write complicates and slows down the entire memory system; applications are penalized by decreased reliability and performance whether they use this functionality or not. AVM allows multiple specialized implementations to co-exist.

Many motivating examples and abstractions for AVM are discussed in [3]. A few of them follow:

Fine-grain monitoring. AVM (by necessity) gives very precise information about and control over the TLB. This information can be used to derive working sets or to trace address streams.

Accurate "in core" information. AVM systems have total control over virtual memory mappings. This information is critical to real-time applications, which must meet deadlines and cannot be subject to the vagaries of current VM systems.

Fast exception propagation. Fast memory protection traps aid DSM and garbage collection systems that use page-protection to detect references.

More efficient dynamic fault isolation. Since context identifiers are available to AVM systems, applications can create light-weight fault isolation domains within their address space. The address space could be split up into protection domains for each identifier, and a protection domain switch would only involve a context identifier switch.

Issues in Designing an AVM

To support AVM, the OS must provide the following functionality: allocation of physical memory, bootstrapping of virtual memory machinery (i.e., TLB miss code and page-tables), efficient exception propogation, secure modification of the mapping hardware (e.g., TLB) and revocation of physical memory. These issues are discussed in [3].

Experiments

Our AVM system is approximately 1000 lines of heavily commented code and runs on Aegis [1], an experimental Exokernel. Its two main limitations are that it does not handle swapping and that page-tables are implemented as a linear vector (address tranlations are looked up in this structure using binary search). Barring these two implementation constraints, its interface is richer than other virtual memory systems we know

*Dawson Engler and Sandeep Gupta are advised by Frans Kaashoek and are supported in part by the Advanced Research Projects Agency under contract N00014-94-1-0985, and by an NSF National Young Investigator Award to Prof. Kaashoek. Sandeep Gupta is also supported in part by an ONR Graduate Fellowship. Address: Room 520, 545 Technology Square, Cambridge MA 02139. Web: http://www.pdos.lcs.mit.edu/~{engler,skgupta}/

Machine	OS	dirty	(un)prot1	prot100	unprot100	trap	appel1	appel2
DEC2100	Ultrix4.2	n/a	51.6	175.	175.	297.	438.	392.
DEC2100	Aegis	17.5	32.5	213.	275.	13.9	74.4	45.9
DEC3100	Ultrix4.2	n/a	47.8	140.	140.	240.	370.	325.
DEC3100	Aegis	13.1	24.4	156.	206.	10.1	55.	34.

Figure 1: Virtual memory benchmarks; times are in micro-seconds

of: it provides flexible support for aliasing, sharing, disabling and enabling of caching on a per-page basis, specific page-allocation, DMA, etc.

We compare our AVM system to Ultrix across seven virtual memory experiments. A complete discussion of the methodology and experiments can be found in [1]. In general, the low-level nature of Aegis allows extremely efficient tuning of system primitives: for example, it performs IPC and exception forwarding 10-100 times faster than Ultrix.

We perform seven experiments. `dirty` measures the time to query whether a page is "dirty" or not. Since it does not require examination of the TLB, this measurement is used to test the base cost of looking up a virtual address in the AVM system's page-table structure. This operation is not provided by Ultrix. `(un)prot1` measures the time required to change the page protection of a single page. `prot100` measures the time required to "read-protect" 100 pages. `unprot100` measures the time required to remove read-protections on 100 pages. `trap` measures the time to take a page-protection trap. `appel1` measures the time to access a random protected page and in the fault-handler, protect some other page and unprotect the faulting page. `appel2` measures the time to protect 100 pages, access each page in a random sequence and, in the fault-handler, unprotect the faulting page.

`dirty` measures the average time to parse the page-table for a random entry. If we compare the time required for `dirty` to the time required to perform `(un)prot1`, over half the time in `(un)prot1` is due to the overhead of parsing the page-table. This overhead can be directly eliminated through the use of a data structure more tuned to efficient lookup (e.g., a hash-table). Even with this penalty, our system performs these operations close to two times more efficiently than Ultrix. The likely reason for this difference is that Aegis dispatches system calls an order of magnitude more efficiently than Ultrix.

In general, our AVM system performs well on this set of benchmarks. The sole exceptions are `prot100` and `unprot100`. Ultrix is extremely efficient in protecting and unprotecting contiguous ranges of virtual addresses: it performs 20% to 60% more efficiently than Aegis in these operations. Part of this difference is a direct result of our immature implementation; another appears to be due to the fact that, on Aegis, changing page-protections requires access to two data structures (Aegis' STLB and the AVM system's page-table). We anticipate these times improving as we tune the system. Fortunately, even with poor performance on these two operations, the benchmark that depends on this operation, `appel2`, is close to an order of magnitude more efficient on Aegis than on Ultrix.

`trap` is another area where the exokernel system performs extremely well (i.e., 21 to 24 times faster than Ultrix). This performance differential is achieved even though the `trap` benchmark on Aegis is implemented with standard signal semantics: for example, all caller-saved registers are saved. If these semantics were violated, the performance difference would become even larger. Finally, the higher-level benchmarks, `appel1` and `appel2`, also show impressive speedup: up to an order of magnitude in some cases and never less than a factor of five.

As these experiments show, an exokernel structure coupled with an AVM system can have impressive performance in even simple operations. At first blush the micro-cost of AVM would seem to add a large overhead to basic memory operations; these benchmarks show that this assumption is wrong. Additionally, these numbers are for an immature system and can be substantially improved.

Conclusions

Application-level VM is a novel virtual memory organization which offers applications complete control over the VM system. Applications can extend existing VM abstractions, specialize them, or even replace them. In addition, the AVM system is less complicated than an OS provided VM system, as it does not have to multithread potentially malicious entities. Finally, the OS kernel can be simplified, because the VM system has been excised from it: the kernel's sole responsibility is to securely multiplex physical memory.

References

[1] D. R. Engler. The design and implementation of a prototype exokernel operating system. Master's thesis, MIT, 545 Technology Square, Boston MA 02139, February 1995.

[2] D. R. Engler, M. F. Kaashoek, and J. O'Toole. The exokernel approach to extensibility (abstract). In *Proceedings of the First Symposium on OSDI*, November 1994.

[3] D.R. Engler, S.K. Gupta, and M. Frans Kaashoek. AVM: Application-level virtual memory. *Proceedings of HOTOS-V*, pages 72–77, May 1995.