AVM: Application-Level Virtual Memory

Dawson R. Engler Sandeep K. Gupta M. Frans Kaashoek
{engler, skgupta, kaashoek}@lcs.mit.edu
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

Virtual memory (VM) is a notoriously complicated abstraction to implement, and is hard to change, specialize, or replace. Although a certain degree of flexibility is achieved by user-level pagers, the control they provide is limited: they leave much of the VM system fixed in the kernel, unreachable by the application. As applications become more diverse and the opportunity cost of bad memory policies grows, it is essential for applications to have more control over the VM abstraction. In this position paper, we motivate and describe a VM system that is implemented completely at the application level. To the best of our knowlege this system is the first complete example of application-level virtual memory (AVM). AVM allows applications to easily specialize, modify, or even replace the VM abstractions offered. For example, on architectures with software TLB management, applications can even select their own page-table structures. In addition, AVM simplifies the OS kernel, since the kernel only multiplexes and does not abstract physical memory. A prototype AVM system is implemented for Aegis, an experimental exokernel.

1 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 kernel, a number of microkernels [1, 11, 25] 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 is fixed) and the page-servers are often complex and have superuser privilege, making them hard to modify or replace. Other microkernels [12, 17, 21] 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 [2, 11, 14].

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.

To support AVM, kernels have to be written differently: instead of abstracting physical memory and TLBs, kernels should safely export and multiplex VM resources directly to application level. We call such a kernel an *exokernel* [9, 10] (to be precise, exokernels strive to export all resources, not just those necessary to support AVM). The AVM system we describe in this paper is built on top of our prototype exokernel, Aegis. Using this system, applications can choose among multiple AVM implementations, select their own policies, modify the VM system, or even craft their own VM abstractions. This thorough control over the VM system enables operations not possible on traditional OSs.

In this paper, we describe the design, implementation, and usage of AVM. Section 2 motivates AVM, and Section 3 describes general design issues for its

This work was supported in part by the Advanced Research Projects Agency under contracts N00014-94-1-0985 and by a NSF National Young Investigator Award.

0-8186-7081-9/95 $04.00 © 1995 IEEE

implementation, and provides specific examples from our implementation. In Section 4 we present measurements of this prototype; related work is discussed in Section 5 and we conclude in Section 6.

2 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.

In this section, we give eleven motivating examples and abstractions for AVM that can be built on top of an exokernel. While many of these could be implemented without AVM, their lack in traditional OSs after three decades of virtual memory research suggests that AVM is required if applications are to acheive any consistent degree of flexibility.

2.1 Examples

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 [18] or to trace address streams [22].

Accurate "in core" information. AVM systems have total control over virtual memory mappings. Their accurate knowledge of which pages are resident in memory can be useful to many types of applications. For example, a scientific program manipulating large matrices could work on those pieces that are "in core", while prefetching others. Garbage collectors can use this information as well, by reclaiming only the storage which is resident. Finally, this information is critical to real-time applications, which must meet deadlines and cannot be subject to the vagaries of current VM systems.

Page replacement policy. Since the AVM system controls paging, it can implement an application-specific paging policy (e.g., LRU, MRU, by priorities). Furthermore, it can decide what to do with pages that are reclaimed. For example, garbage collectors do not need to page out scanned pages, since these pages only contain garbage. Unfortunately, since current OS implementations do not support this optimization, garbage-collection results in a "flurry" of I/O activity, because the OS VM system does not realize that while the page has indeed been modified, it contains garbage and so does not need to be stored to disk [7].

Specific page allocation. An application's ability to request specific physical pages enables a large number of optimizations. For example, control over the physical page-numbers allows cache-conscious layout of data to be done in physically mapped caches through "page-coloring" techniques [5]. This technique is further enabled by the ability of AVM to approximate the working set using TLB snapshots [18]. Additionally, data and text that exhibit poor locality can be mapped to pages of the same color, restricting their pollution of the cache to a specific segment. Applications can also use this functionality to construct contiguous regions of physical memory that are larger than the hardware page size, improving the efficiency of DMA operations [8]. It also allows simpler DMA hardware to be used, since the hardware does not have to support "scatter-gather" functionality.

Control of page-size. On machines that support variable page-size mappings [19, 20], AVM can exploit application specific knowlege to determine appropriate page sizes (e.g., many of the applications discussed in Appel et al. [2] benefit from smaller page sizes).

DMA. Avoidance of the memory subsystem during bulk data transfers can improve performance by eliminating the effects of both cache and TLB misses, and pollution. This optimization can aid many operations such as networking and garbage-collection, and more common operations such as `memcpy()`.

Fine-grain control over virtual memory attributes. Giving applications access to the full complement of hardware facilities allows precise control of page information (e.g., reference bits, page-size, caching attributes). Reference bits can be used to track writes to memory pages (useful for garbage-collectors [2]). Application-controlled caching can be used to reduce cache pollution by disabling caching for memory that exhibits poor locality. For example, a log that absorbs many writes before being flushed to disk should not be cached, since it needlessly evicts cache entries.

Fast exception propagation. Fast memory protection traps aid DSM and garbage collection systems that use page-protection to detect references [2].

2.2 Radical New Structures

AVM enables many radical structures to be built. We examine a few here.

Different address space sizes. With the advent of 64-bit machines, this consideration becomes increasingly important. The page-table mechanisms and structure used to map a sparse 64-bit address space (i.e., inverted page-tables) are different from those appropriate for a dense 32-bit address space (e.g., hierarchical page-tables). Current OSs fix a general-purpose page-table implementation, penalizing applications not fitting within the parameters used to derive it. For example, an OS on a 64-bit machine will likely choose a page-table structure that trades space in mapping large address spaces for speed in mapping small ones. This tradeoff is needless, since if page-table implementations can be isolated in libraries, applications can select (or have selected for them) any one of an array of page-table structures. For instance, a sparse 64-bit address space could use inverted page-tables, a 32-bit address space could use hierarchical page-tables, and a protected object comprised solely of two or three pages can use a simple linear vector. A single-address space OS subsystem may use a completely radical structure; so too might a persistent object store.

Better static fault-isolation techniques. Control of their address space layout allows applications to place sensitive state in arbitrary locations. This technique can be used for improved fault isolation by reducing the chance that a write or read can access this state; in a sense, the virtual address is a capability [24]. Such control can be used to allow applications to safely import untrusted code (or to guard against their own buggy algorithms).

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. This can be used to replace sandboxing or to implement a single-address space operating system at application-level.

Further motivation and examples can be found in [2, 11, 14].

3 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. We restrict our discussion to the mechanics of allocation, bootstrapping and revocation.

Allocation. The first requirement is that the AVM system should be able to allocate physical memory. An important feature of this allocation is that the AVM system should be able to ask for specific physical pages. Aegis supports specific requests for physical pages; to allow AVM systems to enumerate the available pages, the kernel's bitmap of free pages can be mapped into the AVM system's address space as read-only. Implicit in allocation is the ability to track ownership. Aegis uses self-authenticating capabilities for access control. At allocation time, Aegis records the owning process and the capabilities it has selected. The owner of a resource has the power both to change its capabilities and to free it. At all usage or binding points (e.g., insertion of a virtual to physical mapping into the TLB) the AVM must present a capability for the physical memory it is attempting to use. The exokernel checks that this capability is the required one, and if so allows the operation to continue.

Bootstrapping. An exokernel must provide support for bootstrapping the virtual naming system (i.e., supporting translation exceptions on both application page-tables and exception code). Bootstrapping can be performed in a number of ways. The simplest is to have the operating system map the code and page-tables. A more ambitious and (potentially) more flexible mechanism is to allow the AVM system to do the bootstrapping itself by downloading code and data into supervisor mode, where it can access unmapped physical memory and hence either construct a page-table to map the application's page-table and exception code, or simply access all of this using unmapped memory. Finally, since accesses to mapped page-tables can incur TLB misses, nested TLB miss exceptions must be handled.

Aegis provides a simple bootstrapping mechanism through the use of *guaranteed mappings*. An application's virtual address space is partitioned into two segments. The first segment holds normal application data and code. The second segment is used to hold exception handling code, page-tables, etc. The exokernel allows mappings in the second segment to be "pinned" through guaranteed mappings. A miss on a guaranteed mapping will be handled by Aegis. This frees the application from dealing with the intricacies of bootstrapping the TLB and exception handlers that can take TLB misses.

On a TLB miss, the following four actions occur. First, Aegis checks which segment the virtual address resides in. If it is in the standard user segment, the exception is forwarded directly to the application. If it is in the second region, Aegis first checks to see if it is a guaranteed mapping: if so, it installs the TLB entry and continues, otherwise it forwards it to the application. Second, the AVM system looks up the virtual address in its page-table structure, and if the access is not allowed raises the appropriate exception (e.g., "segmentation fault"). If the mapping is valid, the application constructs the appropriate TLB entry and its associated capability and invokes the appropriate exokernel system routine. Third, Aegis checks that the given capability corresponds to the access rights requested by the application. If so, the mapping is installed; control is then returned to the application. Otherwise an error is returned. Finally, The application performs cleanup and resumes execution.

The obvious challenge in supporting AVM is making it fast. The primary bottleneck that must be overcome is the cost of TLB refills. Their overhead can be reduced either by downloading the AVM system's TLB refill code into the kernel (this code can be "sandboxed" to ensure fault-isolation [23]) or by reducing the number of TLB misses that the AVM system must handle. Aegis uses the latter approach: it overlays the hardware TLB with a large software TLB (STLB) to absorb capacity misses [3, 13]. On a TLB miss, Aegis first checks to see whether the required mapping is in the STLB; if so, Aegis installs it and resumes execution. Otherwise, the miss is forwarded to the application.

Currently we use a unified STLB. It is a direct-mapped, resides in unmapped physical memory, and on an STLB "hit", replaces the desired mapping in 18 instructions. The STLB is mapped using a well-known capability, allowing applications to efficiently probe for entries, etc.

Revocation. When physical memory is scarce, the exokernel selects an AVM system to revoke pages from and sends it a revocation interrupt. The AVM system then selects pages to deallocate (perhaps by using LRU, MRU or some other scheme) and returns them to the exokernel (possibly after writing them to disk). To prevent future uses, the exokernel then changes the pages' associated capabilities, and flushes all TLB mappings and any queued DMA requests. In practice, these operations are deferred until the resource is actually reallocated.

To protect against malicious or buggy AVM systems, the exokernel must revoke pages if an AVM system exceeds the time limit it was given to return physical memory. The system's *abort protocol* is used to determine what action to take if this time-limit bound has been exceeded. To allow a usable system, some care must be taken in the design of the abort protocol. For example, the kernel must avoid deallocating the pages holding the AVM system's exception and page-table code without warning and must ensure that translations involving the physical page can be updated, etc. For space reasons, we elide further discussion; a thorough exploration can be found in [10].

4 Experiments

Our AVM system is approximately 1000 lines of heavily commented code. 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 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 Aegis and our AVM system to Ultrix across seven virtual memory experiments, based upon those listed in [2]. The experiments are done within the DECstation/MIPS family. All times are measured using the "wall-clock." All benchmarks were compiled using the same compiler and flags, and run in "single-user" mode. A more complete discussion of the methodology and experiments can be found in [10]. 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 (this benchmark is "prot1+trap+unprot" in Appel et al. [2]). **appel2**: Time to protect 100 pages, access each page in a random sequence and, in the fault-handler, unprotect the faulting page (this benchmark is "protN+trap+unprot" in Appel et al. [2]).

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

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

structure more tuned to efficient lookup (e.g., a hashtable). 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 exokernel-based 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. Two obvious methods would be to use a more sophisticated page-table and to code the operations in assembly (the current use of a high-level language requires that time be wasted saving and restoring registers at exception time).

5 Related Work

It is interesting to contrast our AVM with the approach taken by some of the other current extensible operating system projects. In SPIN [4], for example, applications achieve greater control over the VM systems by downloading spindles, which are fragments of user-code written in a pointer-safe language. On events, such as a page-fault, these spindles are invoked and undertake application-specific operations. This approach offers more application control than user-level pagers do, but is more limited than AVM. For example, the application cannot replace the existing VM abstractions; it can only specialize existing implementations. In addition, this approach makes the kernel more complicated, while AVM simplifies the kernel.

Many of the ideas we discuss here could be implemented on top of the Caching Kernel [6]. However, the Caching Kernel is implemented on the Motorola 68040, which has hardware page-tables, limiting the flexibility with which page-table structures can be experimented. Additionally, it is unclear to what degree applications can control page attributes (e.g., specific page requests or caching attributes) or fully exploit the functionality of the memory subsystem (e.g., DMA). From a more global perspective, the Caching Kernel is *server-based* rather than *application-based*, limiting the per-application specialization. For example, the kernel only supports 16 application kernels simultaneously. Finally, the Caching Kernel has an alarmist view of downloading code into the kernel, limiting the flexibility and efficiency of their approach.

User-level pagers allow a rudimentary control over the VM system by allowing decisions about which pages to swap [16]. However, their interface and power is very limited: they do not allow control over page-table structure, allocation of specific physical pages, or even access to many page attributes (e.g., pagesize or uncached) [7].

The Bridge project attempts to move the virtual memory decisions into the compiler [15]. This is possibly another way to implement many of the features of AVM. However, not enough details have been provided to do a proper comparison of Bridge and AVM.

6 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] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic virtual memory management for operating system kernels. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 123–36, December 1989.

[2] A.W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on ASPLOS*, pages 96–107, Santa Clara, CA, April 1991.

[3] K. Bala, M.F. Kaashoek, and W.E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the First Symposium on OSDI*, pages 243–253, June 1994.

[4] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. SPIN - an extensible microkernel for application-specific operating system services. TR 94-03-03, Univ. of Washington, February 1994.

[5] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct mapped caches. In *Proceedings of the Sixth International Conference on ASPLOS*, pages 158–170, 1994.

[6] D. Cheriton and K. Duda. A caching model of operating system kernel functionality. In *Proceedings of the Sixth SIGOPS European Workshop*, September 1994.

[7] Eric Cooper, Robert Harper, and Peter Lee. The Fox project: Advanced development of systems software. Technical Report CMU-CS-91-178, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

[8] Peter Druschel, Larry L. Peterson, and Bruce S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *SIGCOMM'94*, pages 2–13, 1994.

[9] 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.

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

[11] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on ASPLOS*, pages 187–199, October 1992.

[12] D. Hildebrand. An architectural overview of QNX. In *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April 1992.

[13] J. Huck and J. Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 19th International Symposium on Computer Architecture*, 1992.

[14] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for development of application-specific virtual memory management. In *Proceedings of OOPSLA*, pages 48–64, October 1993.

[15] Steven Lucco. High-performance microkernel systems (abstract). In *Proc. of the first Symp. on OSDI*, November 1994.

[16] Dylan McNamee and Katherine Armstrong. Extending the mach external pager interface to accommodate user-level page replacement policies. In *Mach Workshop Conference Proceedings*, pages 17–30, Burlington, VT, October 4-5 1990. USENIX.

[17] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: a distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[18] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the First Symposium on OSDI*, pages 255–266, June 1994.

[19] M. Talluri, S. Kong, M.D. Hill, and D.A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 415–424, May 1992.

[20] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the Sixth International Conference on ASPLOS*, 1994.

[21] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S.J. Mullender, A. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[22] Richard Uhlig, David Nagle, Trevor Mudge, and Stuart Schrest. Trap-driven simulation with tapeworm II. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 132–144, October 1994.

[23] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, 1993.

[24] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *Proceedings of the Summer 1993 USENIX Conference*, June 1993.

[25] M. Young, A. Tevenian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. Duality of memory and communication in the implementation of a multiprocessor. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 63-67, Austin, TX, Nov. 1987.