The SawMill Framework for Virtual Memory Diversity

Mohit Aron
Department of Computer Science
Rice University

Yoonho Park and Trent Jaeger
IBM T.J. Watson Research Center

Jochen Liedtke and Kevin Elphinstone
System Architecture Group
University of Karlsruhe

Luke Deller
School of Computer Science and Engineering
University of New South Wales, Sydney

Abstract

We present a framework that allows applications to build and customize VM services on the L4 microkernel. While the L4 microkernel's abstractions are quite powerful, using these abstractions effectively requires higher-level paradigms. We propose the dataspace *paradigm which provides a modular VM framework. The modularity introduced by the dataspace paradigm facilitates implementation and permits dynamic configurability. Initial performance results from a prototype are promising.*

1 Introduction

We argue that a virtual-memory system (VM) designed to support a wide variety of applications should have the following features:

- *VM diversity:* Applications should be able to build and customize the VM according to their needs. They should have complete control over the VM policies. The alternative that frequently occurs in practice is that application programmers settle for a policy in the kernel that comes closest to providing the right level of service.
- *Dynamic extensibility:* Applications should be able to dynamically extend the VM system. This calls for a modular design that provides the following benefits: (1) code reuse which facilitates implementation of VM policies, (2) enables VM policies to be easily tuned to applications needs, and (3) dynamic configurability.
- *Performance:* Increased functionality should not be performance limited. Consider Mach user-level pagers. Mach applications can use pagers to control how data is moved between physical memory and backing store. This increased functionality is policy-limited — i.e., pagers cannot control replacement policy — and performance-limited by the high cost of Mach IPCs. Backing store is usually slow and hides the high cost of Mach IPCs in page-fault handling performance. If backing store is not slow, page-fault handling performance will be limited by the cost of Mach IPCs.

In this paper, we present the SawMill VM framework in the context of the L4 microkernel [16, 17]. The SawMill project itself aims to develop highly-configurable operating-system technology to address the complexity of building and maintaining a variety of custom operating systems [8]. The SawMill VM framework reconciles the conflicting goals of functionality and performance in order to provide flexibility to applications. The design provides a modular and dynamically extensible framework that enables applications to (1) build application-specific VM services from modular components and (2) dynamically plug these services into the existing framework. Initial performance results with a prototype implementation are promising.

The rest of the paper is organized as follows. Section 2 describes the L4 microkernel and its abstractions for supporting VM. It describes the L4 *pagers* that are responsible for handling page faults and describes their use in the *hierarchical* management of address spaces. Section 3 introduces the concept of a *dataspace* — an unstructured container of data — and shows how it augments L4 abstractions to provide a dynamically extensible VM framework. In Section 4 we discuss the types of components that we envision for the VM framework. In Section 5 we evaluate the performance of our VM framework prototype using popular benchmarks. Section 6 presents related work, and Section 7 summarizes.

2 L4 Microkernel

In this section, we briefly present the abstractions and primitives provided by the L4 microkernel. By means of an example pager, we illustrate how the primitives can be used, and how powerful as well as flexible they are. A detailed discussion of the L4 microkernel's API can be found elsewhere [17].

2.1 VM primitives

The L4 microkernel provides two abstractions: *threads* and *address spaces*. The thread is a unit of execution and is associated to a unique address space. An address space and the threads associated with it are collectively referred to as a L4 task. The L4 threads can communicate with each other using the IPC operations provided by the microkernel [16, 10]. At the hardware level, an address space is a mapping that associates each virtual page with a physical page frame or marks it non-accessible. An address space defines the virtual memory of the threads associated with it.

For the purpose of page-fault handling, the microkernel supports the notion of per-thread *pagers*. A pager in L4 is a thread running in the same or different address space as the faulting thread. The page fault is reflected to a pager thread as an IPC, the reply for which is used by the pager to map a page into the faulting thread's address space.

The L4 microkernel permits hierarchical management of its address spaces. In other words, a pager's address space might itself be managed by another address space. For such a hierarchical scheme to work, we need one intial address space. This address space, called σ_0, is created at system start time and is idempotent to the physical memory of the machine. Management of other address spaces is enabled by means of the following microkernel operations:

Grant. A thread associated with an address space can *grant* any of its pages to another space, provided the recipient agrees. The granted page is removed from the granter's address space and included into the grantee's address space. The important restriction is that instead of physical page frames, the granter can only grant pages that are already accessible to itself.

Map. A thread in an address space can *map* any of its pages into another address space, provided the recipient agrees. Afterwards, the page can be accessed in both address spaces. In contrast to granting, the page is not removed from the mapper's address space. Comparable to the granting case, the mapper can only map pages that it itself can access.

Unmap. A thread in an address space can *unmap* any of its pages. The unmapped page remains accessible in the unmapper's address space, but is removed from all other address spaces that had received the page directly or indirectly from the unmapper. Although explicit consent of the affected address-space owners is not required, the operation is safe, since it is restricted to ones own pages. The users of these pages already agreed to accept a potential unmapping, when they received the pages by mapping or granting.

Page-fault handling is done outside the microkernel and only the *grant*, *map* and *unmap* operations are done inside. The microkernel only reflects the page fault to the corresponding pager thread by means of an IPC. The actual page-fault handling is left upto the pager thread.

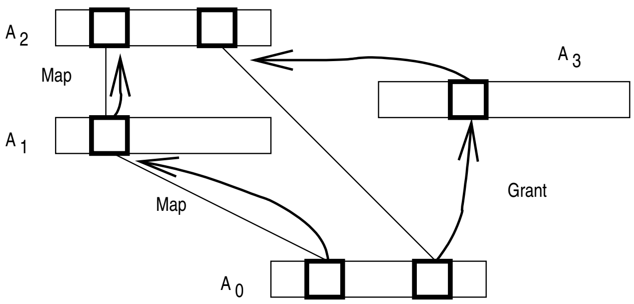

Figure 1. L4 microkernel map and grant operations.

Figure 1 shows example uses of the above mentioned microkernel operations. Address space A_0 maps a page into address space A_1 which further maps it into address space A_2 . The resulting mapping is shown as a thin line from A_0 to A_1 and then to A_2. If A_0 were to unmap this page in its address space, then the mapping would disappear in A_1 as well as A_2. However, if A_1 unmaps the mapping in his address space, the mapping will disappear in A_2's address space, but is retained in A_0's address space. The figure also shows a *grant* operation. Address space A_0 maps a page into address space A_3 which grants it into address space A_2. As a result, the mapping disappears in A_3. The resulting mapping is indicated as a thin line from A_0 to A_2.

2.2 Constructing a Simple Example Pager

As an example to illustrate the mechanisms decribed above, we describe the construction of a simple pager. It manages only a single address space and uses a contiguous partition of a disk for swapping. To make things a little bit more complicated, the physical video frame buffer is mapped one-to-one into the address space and is, of course, not subject to swapping.

As shown in Section 2.1, a pager uses the pages in its own address space and maps them into the faulting thread's address space. We will refer to the pages in the pager's address space as *PagerPages*. For easier understanding, the reader might provisionally think of the *PagerPages* as physical pages in this context. However, it must be noted that, in fact, they are *virtual pages* (of the pager's address space). Therefore, we do not use the term *physical pages*.

Figure 2 shows the pseudo-code used by our simple example pager to implement page-fault handling for its client address space. *ClientPage* refers to the faulting virtual page in the client's address space and *SelectedPagerPage* refers to the *PagerPage* that the pager selects for mapping into the *ClientPage*. *FaultType* indicates whether the page fault is on a read or a write.

```
do
   wait for page-fault ipc from ClientSpace ;
   if FaultAddress within frame buffer
      then  select corresponding frame buffer PagerPage
   elif a PagerPage is associated with FaultAddress
      then  select FaultAddress-associated PagerPage
      else  select a PagerPage to be replaced ;
            if SelectedPagerPage in use
               then  unmap (SelectedPagerPage) ;
                     write back (SelectedPagerPage)
            fi ;
            load required page into SelectedPagerPage
   fi ;
   touch (SelectedPagerPage) ;        /* see Section 2.3 */
   map (SelectedPagerPage, FaultAddress, ClientSpace)
od .
```

Figure 2. A simple example pager.

As mentioned in Section 2.1, the microkernel notifies the pager about the occurrence of a page fault in the client's address space using an IPC. This IPC also contains information about the faulting page in the client's address space as well as whether the fault is upon a read or a write. If there is currently no *PagerPage* associated to the faulting client page, the pager selects an arbitrary *PagerPage* for replacement, writes it to swap if necessary, loads the required contents into the selected *PagerPage*, and then maps it to the faulting *ClientPage* in the client's address space. Just before mapping the *SelectedPagerPage*, the pager touches it so as to ensure that it is mapped in its own address space in case any underlying pager had unmapped it. Section 2.3 discusses this aspect in more detail.

The reader should note that a *PagerPage* is always unmapped before its contents are replaced. The corresponding microkernel primitive unmaps it from all address spaces the pager had mapped it, i.e. in our example from the client's space, but leaves it mapped inside the pager's address space. This ensures that the client does not inadvertently access incorrect data if the *PagerPage* in question had been mapped into the client's address space earlier. As an effect, the pager can access the page while the client will incur a page fault if it tries to do the same.

An advantage afforded by our example pager is that information regarding the association between a *ClientPage* and the corresponding *PagerPage* is kept internal to the pager in a page table data structure. The pager can then choose any suitable page table organization.

2.3 Hierarchical Pagers

We use the example pager shown in Section 2.2 to show how two such pagers can be stacked. This is possible because our example pager did not use physical pages. Its *PagerPages* were virtual pages of the pager's address space. Thus stacking two such pagers is easily possible.

As shown in Figure 3, the upper pager P_1 is then the client of the underlying pager P_0. The *PagerPages* of P_1 are the virtual *ClientPages* for P_0. Stacking one example pager on top of another one doesn't really extend the functionality of the first pager. Nevetheless, it is a simple and good example for illustrating how hierarchical pagers work and how they must be constructed to be independent of their potentially underlying pagers. Later in this paper (Section 4), we present really meaningful, but also more complicated, hierarchical pagers that extend each others semantics significantly.

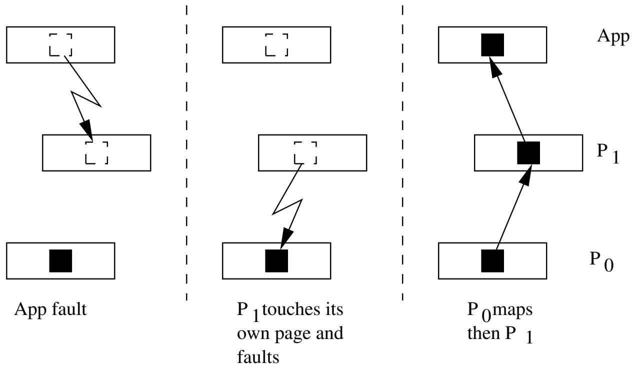

Figure 3. Nested pagers.

As long as P_1's *PagerPages* are all mapped into P_1's address space, P_0 does obviously not affect P_1. The fact that P_1's pager pages may have different physical than virtual addresses is transparent to P_1 since it never sees the physical addresses.

The situation becomes more complicated when some of P_1's pager pages are not mapped, i.e. have not been used before or have been unmapped by P_0 for some reason. Assume that P_1 receives a page fault, finds an appropriate

PagerPage, and maps it into its client. What happens if that *PagerPage* itself was not mapped into P_1's address space?

To answer the question, we must understand that L4's map primitive passes existing access rights from the mapper to the mappee. The mapper can narrow the access rights (e.g. to read-only) but he can never grant the mappee more rights than the mapper possesses itself on the page. Consequently, mapping an unmapped page (no access rights at all) unmaps the page in the mappee's address space as well.

To ensure that its *PagerPage* is mapped into its own address space when mapping it into the client's address space, the example pager always touches its own *PagerPage* prior to mapping it into the client; if P_1's *PagerPage* is currently unmapped, touching raises a page fault in P_1. The lower-level pager P_0 will handle it transparently to P_1, potentially swapping in a page from the P_0-disk. Afterwards, P_1's *PagerPage* is mapped into P_1 so that the subsequent map operation of P_1 into its client space will succeed with a high probability. If the just mapped *PagerPage* is unmapped by P_0 a second time between touching and mapping (unlikely but possible), we will see another page fault at the client, touching and mapping by P_1 , and so forth. Depending on P_0's qualities, the game will terminate sooner or later. Without touching, it would never terminate.

3 An Extensible VM Framework

Pagers as described above, are a low-level and strongly microkernel-related concept. Using the concept effectively requires more elaborate higher-level paradigms and concepts. Any such framework should (a) pass all the flexibility and power of the underlying pager concept to its applications, (b) ensure extensibility by means of different instances and types of those "higher-level pagers," (c) ensure interoperability between them, and (d) offer customized semantics for a wide range of application types.

As an instance of such a framework, we present SawMill's *dataspace* paradigm, the design and implementation of an according framework, and some examples. The system described is a true user-level system, located outside the microkernel, and uses the low-level operations described in Section 2 to provide virtual memory to applications.

3.1 The Concept of Dataspaces

The virtual-memory framework is based on dataspaces[1]. A dataspace is an unstructured data container. In other words, the term dataspace abstracts any system entity that contains data. Examples for dataspaces are files, anonymous memory, frame buffers, etc.

[1]The term "Dataspace" was coined by L3 developers [4].

Dataspaces can be *attached* to *regions* of an address space. Accessing the virtual memory of a region thus effectively accesses the dataspace associated with the region. In Figure 4, address space A_1 has two regions to which the dataspaces ds_1 and ds_2 are attached. ds_2 is additionally attached to a region of A_2.

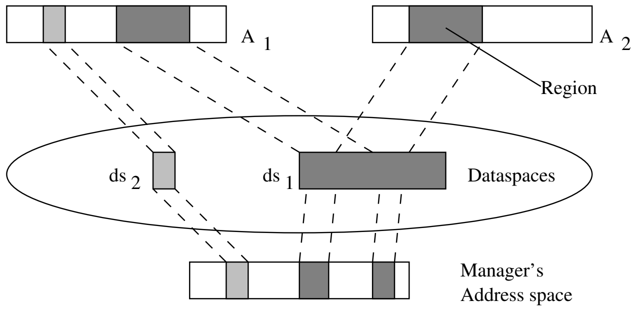

Figure 4. Relationship between address spaces, dataspaces, and regions.

The *region map* is a per-address space object that keeps track of the attached dataspaces and translates any virtual address to a 3-tuple (dataspace manager, dataspace id, offset). Every page fault is captured by the region map. It translates the faulting address and then forwards the page fault, including dataspace id and offset, by means of IPC to the dataspace manager that corresponds to the faulting address.

Dataspace managers implement dataspaces. Each such manager determines the semantics of the dataspaces that it offers. For instance, one manager might offer physical frame buffers as dataspaces, another one anonymous paged memory, a third one Unix files, a fourth one MSDOS files, a fifth one distributed shared memory dataspaces. By attaching those dataspaces to address space regions, the managers also define the semantics of the address space regions to which their dataspaces are currently attached.

In L4 microkernel terms, dataspace managers are pagers. Typically, they cache the contents of dataspaces in their own virtual address-spaces as shown in Figure 4 and use the microkernel's VM operations (presented in Section 2) to satisfy application page faults.

As discussed in Section 2, the L4 microkernel binds pagers to threads. The pager is then responsible for servicing page faults generated by the thread. In our framework, the dataspace manager is the pager and is bound to a region. The region map is declared the thread's pager, but the dataspace manager services the page fault. Binding pagers to regions is more conventional but less flexible than binding pagers to threads. Binding pagers to threads allows the construction of simple pagers such as the pager presented in Section 2.2 without any region bookkeeping overhead.

It is to be noted that the dataspace concept is a higher-level concept in that the microkernel is unaware of dataspaces. Attaching a dataspace is a logical operation that provides access to the dataspace through the virtual memory of the application. The actual series of steps leading from the attachment of a dataspace to the actual accessing of its content by the application can be enumerated as follows:

1. An application attaches a dataspace to a virtual-memory region.
2. The application accesses a page in the virtual-memory region — this generates a page fault since the virtual memory is as of yet unmapped.
3. The region mapping is notified of the page fault by an IPC.
4. The region mapping translates the faulting address to (dataspace manager,dataspace id, offset).
5. The region mapping forwards the page fault including dataspace id and offset to the appropriate dataspace manager by another IPC.
6. The dataspace manager caches the contents of the dataspace in a mapped virtual-memory page in its own address space.
7. This VM page is mapped into the applications address space using the microkernel's map or grant operations.

3.2 Operations On Dataspaces

All dataspace managers must support *identify, attach/open, detach/ close*, and *interrogate* described below. *Share, copy, transfer, create*, and *delete* are optional operations.

Identify: Request for a dataspace id. The manager returns a dataspace id. The information provided in the request is manager-dependent. For example, the request can contain an id such as a file descriptor or a path name.

Attach/Open: Opens a dataspace for access and attaches it to a region. The request can also contain open methods such as read-only or read-write. The request can either work on a dataspace id that was priorily delivered by an *identify* operation, or it can work on, e.g., a file name and implicitly *identify* the dataspace. After an attach, the region mapping forwards all page faults in that region to the dataspace manager. The dataspace manager resolves the faults with *map* or *grant* operations.

Detach/Close: Removes a region-dataspace mapping. Any mapped pages are unmapped. The dataspace is no longer accessible, but the dataspace id remains valid and can be used, e.g., for another open.

Interrogate: Request to determine which operations and flavors of operations the manager supports. For example, it is unlikely a dataspace manager that provides a video frame buffer will support the copy operation.

Share: Request to allow a dataspace to be shared with another task. After the share operation, the other task is allowed to open the dataspace. The sharing semantics are manager-dependent. For example, the manager can offer read-only sharing requests.

Copy: Request to create a copy of an existing dataspace in the same or different manager. The manager returns the dataspace id of the copy. Managers are free to define the semantics of the copy. For example, copying can be performed lazily.

Transfer: Request to transfer ownership of a dataspace to another task. The dataspace is detached and closed in the transferring task, and can then be opened and attached in the target task.

Create: Request to create a dataspace. The request contains an id such as a file descriptor or path name. There is no effect on the requestor's address space.

Delete: Request to delete a dataspace specified by a dataspace id. All attachments are invalidated.

3.3 Extensibility of Dataspace Semantics

Existing dataspace semantics can be modified in two ways. The *transfer* operation allows applications to dynamically move dataspaces. This allows applications to layer additional semantics on existing dataspaces by stacking dataspace managers. Applications can also replace semantics of existing dataspaces.

Figure 5 shows the dynamic stacking of dataspace managers. Before the managers are stacked, the application uses a dataspace from dataspace manager 1. To stack the managers, the dataspace is transferred from the application to dataspace manager 2 (not shown). The application then re-{attaches/opens} the dataspace from dataspace manager 2. Dataspace manager 2 is now free to extend the semantics of the dataspace. For example, the manager can increase/decrease accessibility, add lazy copying, or add persistence.

Figure 6 shows how an application can replace the semantics of existing dataspaces. Suppose dataspace manager 1 provides swappable, anonymous memory and dataspace

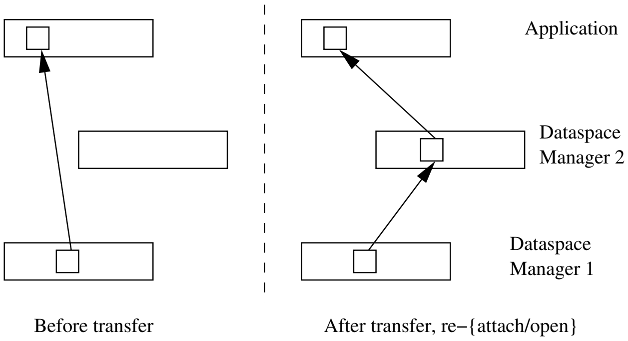

Figure 5. Extending the semantics of a dataspace by dynamically stacking managers.

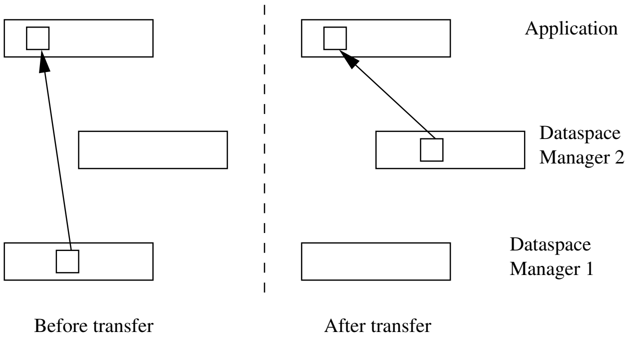

Figure 6. Modifying the semantics of a dataspace by moving the dataspace between managers.

manager 2 provides non-swappable, anonymous memory. Moving a dataspace from 1 to 2 is a pinning operation. Moving a dataspace from 2 to 1 is an unpinning operation.

L^4Linux [10], a port of Linux to the L4 microkernel, provides an interesting opportunity to experiment with a large system. L^4Linux acts as a pager for L^4Linux applications. Placing L^4Linux within our framework would allow L^4Linux applications to arbitrarily extend L^4Linux VM semantics. This has several interesting possibilities. One possibility is that existing semantics could be preserved and additional semantics such as persistence could then be layered. Another possibility is that existing semantics could be stripped to a minimum and applications could then layer only the semantics it deems necessary.

4 VM Diversity

Different instances and types of dataspace managers will ensure the extensibility of the VM framework. At this time, we envision three types of memories provided by dataspace managers — basic memories, paged memories, and specialized memories.

Basic memories represent physical memories and include main memory, colored memory, and device memory. A main memory manager can be thought of as a σ_0 dataspace manager. It controls the allocation of main memory by exporting dataspaces that map directly to main memory. A dataspace provided by a colored memory manager represents physical page-frames which select the same cache bank. Such dataspaces provide a user-level cache-partitioning mechanism. Cache-partitioning is useful in real-time systems where task switches disrupt cache working sets, making execution times unpredictable [18]. A video frame buffer is an example of device memory. The primary responsibility of a device memory manager is access mediation.

Paged memories include anonymous memory, file systems, and compressed memory. Anonymous memory is zero-filled memory backed by secondary storage, usually disk. Compressed memory is useful in memory-constrained devices and can be used with a wide variety of other memories including anonymous memory and file systems.

Specialized memories include pinned memory and persistent memory. Traditional pinning is static. Pages are pinned forever or until the user unpins the pages. Quotas are used to control static pinning. In [19] we propose dynamic pinning. In this scheme, pages are pinned for short periods of time. When and how many pages are pinned is determined dynamically. Both static and dynamic pinning can be implemented by a dataspace manager. Within our framework, adding persistence to a dataspace is just a matter of inserting a persistent dataspace below the dataspace as in Figure 5. The manager can periodically write the dataspace to secondary storage or employ an incremental strategy.

5 Performance Analysis

We have implemented a prototype that consists of a main memory manager and an anonymous memory manager. The managers implement *identify, attach/open, detach/close*, and *interrogate*.

Table 1 compares the time to execute the Appel-Li VM primitives [2]. The times for OSF/1, Mach and SPIN taken from [3] were measured on a 133MHz DEC Alpha. The numbers for SawMill were measured on a 100MHz Pentium which is roughly comparable to a 133MHz DEC Alpha. The times for SPIN correspond to kernel extensions invoking the virtual-memory system and would be 4μs more expensive had they been invoked from user-level. The times for SawMill on the other hand correspond to VM usage by user-level applications.

The *Prot* primitive measures the time to increase the protection of a single page. The *Prot100* and *Unprot100* mea-

Table 1. VM primitive performance. Times shown are in μs.

	OSF/1	Mach	SPIN	SawMill
Prot	45	106	16	22
Trap	260	185	7	22
Fault	329	415	29	22
Prot100	1041	1792	213	51
Unprot100	1016	302	214	25
Appel1	382	819	39	22
Appel2	351	608	29	25

sure the time to increase and decrease the protection respectively over a range of 100 pages. *Trap* measures the latency between a page fault and the time when a user-specified handler executes. *Fault* measures the total latency perceived by the application in receiving a page fault, enabling access to the page within the handler and resuming the faulting thread. The *Appel1* and *Appel2* primitives measure a combination of trap and protection changes. *Appel1* measures the time to fault on a protected page, enable access to the page within a handler and protect another page within the handler. *Appel2* measures the time to protect 100 pages and faulting on each one, unprotecting the page in the handler. Table 1 shows the average cost per page for *Appel2*.

The results in Table 1 indicate the SawMill VM framework is capable of achieving virtual memory performance at user-level that is comparable to SPIN's performance with kernel extensions. The significantly lower cost of the SawMill *Prot100* and *Unprot100* is attributed to the L4 microkernel's capability of being able to change protection at the granularity of superpages. The protection of 100 contiguous virtual pages is accomplished by changing the protection of just three superpages of 64, 32, and 4 machine-size pages.

6 Related Work

Numerous efforts have been made to provide application-specific VM. Mach [25] user-level pagers allow applications to control how data is transferred between physical memory and backing store. User-level pagers were later incoporated by Chorus [1] and Spring [13]. Premo pagers [20] and extensible object-oriented virtual-memory [14] extended Mach pagers by allowing pagers to implement replacement policies. HiPEC [15] allows applications to control replacement policies by downloading policies written in a restricted language to the kernel. Sechrest [24] and V++ page-cache managers [11] extended pagers even further by allowing pagers to implement replacement and placement policies. Sechrest and V++ page-cache managers also moved allocation policies out of the kernel.

SPIN [3] attempts to provide application-specific VM through kernel extensions. However, downloading untrusted user code into the kernel safely remains a difficult and unavoidable problem [5]. Exokernel [7, 12] takes a radically different approach in that its kernel exports the underlying hardware safely to applications that can implement VM in user-level libraries. AVM [6] shares some of the same goals as our work. However, AVM does not provide any framework to facilitate the construction and extension of VM services.

Grasshopper [22] is the only other operating system we know that permits hierarchical address space mappings. Grasshopper provides containers which are similar to dataspaces, however, container managers do not have control over all VM policies and lack well-defined interfaces. While our work focuses on VM, it is very similar in philosophy to the extensible network protocol framework proposed by O'Malley [21] and file system stacking research [9, 23]. Both bodies of research demonstrated that a modular framework promotes code reuse and allows the application programmers to successfully configure services to their needs. We are hoping to demonstrate similar points within the context of VM.

7 Summary

We have presented a virtual-memory framework that is capable of providing application specific policies that can be dynamically extended to suit application needs. Benchmark results from a prototype implementation indicate that our that the proposed framework is capable of affording performance that contemporary research implementations have only been able to achieve through kernel extensions.

References

[1] V. Abrossimov, M. Rozier, and M. Gien. Virtual memory management in Chorus. In *Workshop on Progress in Distributed Systems and Distributed Systems Management*. Springer-Verlag, April 1989.

[2] A. Appel and K. Li. Virtual memory primitives for user programs. In *ASPLOS*. ACM, April 1991.

[3] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*. ACM, December 1995.

[4] U. Beyer, D. Heinrichs, and J. Liedtke. Dataspaces in L3. In *Mini and Microcomputers and Their Applications*. The International Society for Mini and Microcomputers, 1988.

[5] P. Druschel, V. Pai, and W. Zwaenepoel. Extensible kernels are leading OS research astray. In *HotOS-VI*. IEEE, May 1997.

[6] D. Engler, S. Gupta, and M. Kaashoek. AVM: Application-level virtual memory. In *HotOS-V*. IEEE, May 1995.

[7] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *SOSP*. ACM, December 1995.

[8] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon Tidswell, Luke Deller, and Lars Reuther. The sawmill multiserver approach. In *SIGOPS European Workshop*. ACM, September 2000.

[9] R. Guy, J. Heidemann, W. Mak, T. Page Jr., and G. Popek. Implementation of the Ficus replicated file system. In *USENIX*. USENIX, June 1990.

[10] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *SOSP*. ACM, October 1997.

[11] K. Harty and D. Cheriton. Application-controlled physical memory using external page-cache management. In *ASPLOS*. ACM, October 1992.

[12] M. Kaashoek, D. Engler, G. Ganger, H. Briceño, R. Hunt, D. Maziéres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *SOSP*. ACM, October 1997.

[13] Y. Khalidi and M. Nelson. The Spring virtual memory system. Technical Report SMLI TR-93-09, Sun Labs, February 1993.

[14] K. Krueger, D. Loftesness, A. Vahdat, and D. Anderson. Tool for the developement of application-specific virtual memory. In *OOPSLA*. ACM, October 1993.

[15] C. Lee, M. Chen, and R. Chang. HiPEC: High performance external virtual memory caching. In *OSDI*. USENIX, November 1994.

[16] J. Liedtke. Improving IPC by kernel design. In *SOSP*. ACM, December 1993.

[17] J. Liedtke. On μ-kernel construction. In *SOSP*. ACM, December 1995.

[18] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *Real-time Technology and Applications Symposium*. IEEE, May 1997.

[19] J. Liedtke, V. Uhlig, and O. Hers. How to schedule unlimited memory pinning of untrusted processes or provisional ideas about service neutrality, December 1998. Submitted to HotOS-VII.

[20] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accomodate user-level page replacement policies. In *Mach Workshop*. USENIX, October 1990.

[21] S. O'Malley and L. Peterson. A dynamic network architecture. *TOCS*, 10(2), May 1992.

[22] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris. Operating system support for persistent and recoverable computations. *CACM*, 39(9), September 1981.

[23] D. Rosenthal. Evolving the vnode interface. In *USENIX*. USENIX, June 1985.

[24] S. Sechrest and Y. Park. User-level physical memory management for Mach. In *Mach Symposium*. USENIX, November 1991.

[25] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *SOSP*. ACM, November 1987.