The Pebble Component-Based Operating System

Eran Gabber, Christopher Small, John Bruno†, José Brustoloni and Avi Silberschatz

Information Sciences Research Center
Lucent Technologies—Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974
{eran, chris, jbruno, jcb, avi}@research.bell-labs.com

†*Also affiliated with the University of California at Santa Barbara*

Abstract

Pebble is a new operating system designed with the goals of flexibility, safety, and performance. Its architecture combines a set of features heretofore not found in a single system, including (a) a minimal privileged mode nucleus, responsible for switching between protection domains, (b) implementation of all system services by replaceable user-level components with minimal privileges (including the scheduler and all device drivers) that run in separate protection domains enforced by hardware memory protection, and (c) generation of code specialized for each possible cross-domain transfer. The combination of these techniques results in a system with extremely inexpensive cross-domain calls that makes it well-suited for both efficiently specializing the operating system on a per-application basis and supporting modern component-based applications.

1 Introduction

A new operating system project should address a real problem that is not currently being addressed; constructing yet another general purpose POSIX- or Windows32-compliant system that runs standard applications is not a worthwhile goal in and of itself. The Pebble operating system was designed with the goal of providing flexibility, safety, and high performance to applications in ways that are not addressed by standard desktop operating systems.

Flexibility is important for specialized systems, often referred to as embedded systems. The term is a misnomer, however, as embedded systems run not just on microcontrollers in cars and microwaves, but also on high-performance general purpose processors found in routers, laser printers, and hand-held computing devices.

Safety is important when living in today's world of mobile code and component-based applications. Although safe languages such as Java [Gosling96] and Limbo [Dorward97] can be used for many applications, hardware memory protection is important when code is written in unsafe languages such as C and C++.

High performance cannot be sacrificed to provide safety and flexibility. History has shown us that systems are chosen primarily for their performance characteristics; safety and flexibility almost always come in second place. Any system structure added to support flexibility and safety cannot come at a significant decrease in performance; if possible, a new system should offer better performance than existing systems.

Early in the project, the designers of Pebble decided that to maximize system flexibility Pebble would run as little code as possible in its privileged mode nucleus. If a piece of functionality could be run at user level, it was removed from the nucleus. This approach makes it easy to replace, layer, and offer alternative versions of operating system services.

Each user-level component runs in its own *protection domain*, isolated by means of hardware memory protection. All communication between protection domains is done by means of a generalization of interrupt handlers, termed *portals*. Only if a portal exists between protection domain A and protection domain B can A invoke a service offered by B. Because each protection domain has its own *portal table*, by restricting the set of portals available to a protection domain, threads in that domain are efficiently isolated from services to which they should not have access.

Portals are not only the basis for flexibility and safety in Pebble, they are also the key to its high performance. Specialized, tamper-proof code can be generated for each portal, using a simple interface definition language. Portal code can thus be optimized for its portal, saving and restoring the minimum necessary state, or encapsulating and compiling out demultiplexing decisions and run-time checks.

The remainder of this paper is structured as follows. In Section 2 we discuss related work. In Section 3 we describe the architecture of Pebble, and in Section 4 we discuss the portal mechanism and its uses in more detail. Section 5 covers several key implementation issues of Pebble. Section 6 introduces the idea of implementing a protected, application-transparent "sandbox" via portal interposition, and shows the performance overhead of such a sandbox. Section 7 compares the performance of Pebble and OpenBSD on our test hardware, a MIPS R5000 processor. Section 8 reviews the current status of Pebble and discusses our plans for future work. We summarize in Section 9, and include a short code example that implements the sandbox discussed in Section 6.

2 Related Work

Pebble has the same general structure as classical microkernel operating systems such as Mach [Acetta86], Chorus [Rozer88], and Windows NT [Custer92], consisting of a privileged mode kernel and a collection of user level servers. Pebble's protected mode nucleus is much smaller and has fewer responsibilities than the kernels of these systems, and in that way is much more like the L4 microkernel [Liedtke95]. L4 and Pebble share a common philosophy of running as little code in privileged mode as possible. Where L4 implements IPC and minimal virtual memory management in privileged mode, Pebble's nucleus includes only code to transfer threads from one protection domain to another and a small number of support functions that require kernel mode.

Mach provides a facility to intercept system calls and service them at user level [Golub90]. Pebble's portal mechanism, which was designed for high-performance cross-protection-domain transfer, can be used in a similar way, taking an existing application component and interposing one or more components between the application component and the services it uses.

Pebble's architecture is closer in spirit to the nested process architecture of Fluke [Ford96]. Fluke provides an architecture in which virtual operating systems can be layered, with each layer only affecting the performance of the subset of the operating system interface it implements. For example, the presence of multiple virtual memory management "nesters" (*e.g.*, to provide demand paging, distributed shared memory, and persistence) would have no effect on the cost of invoking file system operations such as `read` and `write`. The Fluke model requires that system functionality be replaced in groups; a memory management nester must implement all of the functions in the virtual memory interface specification. Pebble portals can be replaced piecemeal, which permits finer-grained extensibility.

The Exokernel model [Engler95, Kaashoek97] attempts to "exterminate all OS abstractions," with the privileged mode kernel in charge of protecting resources, but leaving resource abstraction to user level application code. As with the Exokernel approach, Pebble moves the implementation of resource abstractions to user level, but unlike the Exokernel, Pebble provides a set of abstractions, implemented by user-level operating system components. Pebble OS components can be added or replaced, allowing alternate OS abstractions to coexist or override the default set.

Pebble can use the interposition technique discussed in Section 6 to wrap a "sandbox" around untrusted code. Several extensible operating system projects have studied the use of software techniques, such as safe languages (*e.g.*, Spin [Bershad95]) and software fault isolation (*e.g.*, VINO [Seltzer96]), for this purpose. Where software techniques require faith in the safety of a compiler, interpreter, or software fault isolation tool, a sandbox implemented by portal interposition and hardware memory protection provides isolation at the hardware level, which may be simpler to verify than software techniques.

Philosophically, the Pebble approach to sandboxing is akin to that provided by the Plan 9 operating system [Pike90]. In Plan 9, nearly all resources are modeled as files, and each process has its own file name space. By restricting the namespace of a process, it can be effectively isolated from resources to which it should not have access. In contrast with Plan 9, Pebble can restrict access to any service, not just those represented by files.

Pebble applies techniques developed by Bershad et al. [Bershad89], Massalin [Massalin92], and Pu et al. [Pu95] to improve the performance of IPC. Bershad's results showed that IPC data size tends to be very small (which fits into registers) or large (which is passed by sharing memory pages). Massalin's work on the Synthesis project, and, more recently, work by Pu et al. on the Synthetix project, studied the use of generating specialized code to improve performance.

Pebble was inspired by the SPACE project [Probert91]. Many of the concepts and much of the terminology of the project come from Probert's work; *e.g.*, SPACE provided us with the idea of cross-domain communication as a generalization of interrupt handling.

The Spring kernel [Mitchell94] provided cross-protection domain calls via doors, which are similar to Pebble's portals. However, Spring's doors are used only for implementing operations on objects, and do not include general purpose parameter manipulations.

The Kea system [Veitch96] is very similar to Pebble. It provides protection domains, inter-domain calls via portals and portal remapping. However, Kea's portals do not perform general parameter manipulations like Pebble. Parameter manipulations, such as sharing memory pages, are essential for efficient communication between components.

The MMLite system [Helander98] is a component-based system that provides a wide selection of object-oriented components that are assembled into an application system. MMLite's components are space efficient. However, MMLite does not use any memory protection, and all components execute in the same protection domain.

Like Dijkstra's THE system [Dijkstra68], Pebble hides the details of interrupts from higher level components and uses only semaphores for synchronization.

Some CISC processors provide a single instruction that performs a full context switch. A notable example is the Intel x86 task switch via a call gate [Intel94]. However, this instruction takes more than 100 machine cycles.

3 Philosophy and Architecture

The Pebble philosophy consists of the following four key ideas.

The privileged-mode nucleus is as small as possible. If something can be run at user level, it is.

The privileged-mode nucleus is only responsible for switching between protection domains. In a perfect world, Pebble would include only one privileged-mode instruction, which would transfer control from one protection domain to the next. By minimizing the work done in privileged mode, we reduce both the amount of privileged code and the time needed to perform essential privileged mode services.

The operating system is built from fine-grained replaceable components, isolated through the use of hardware memory protection.

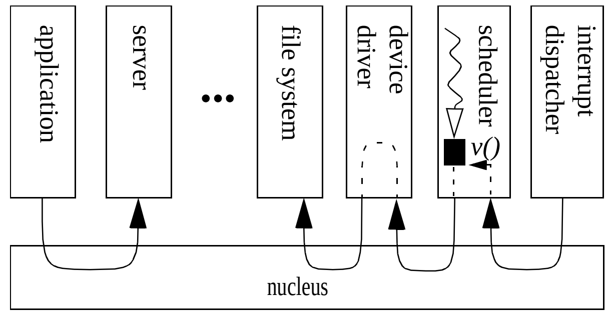


Figure 1. Pebble architecture. Arrows denote portal traversals. On the right, an interrupt causes a device driver's semaphore to be incremented, unblocking the device driver's thread (see Section).

The functionality of the operating system is implemented by trusted user-level components. The components can be replaced, augmented, or layered.

The architecture of Pebble is based around the availability of hardware memory protection; Pebble, as described here, requires a memory management unit.

The cost of transferring a thread from one protection domain to another should be small enough that there is no performance-related reason to co-locate services.

It has been demonstrated that the cost of using hardware memory protection on the Intel x86 can be made extremely small [Liedtke97], and we believe that if it can be done on the x86, it could be done anywhere. Our results bear us out—Pebble can perform a one-way IPC in 114 machine cycles on a MIPS R5000 processor (see Section 7 for details).

Transferring a thread between protection domains is done by a generalization of hardware interrupt handling, termed portal traversal. Portal code is generated dynamically and performs portal-specific actions.

Hardware interrupts, IPC, and the Pebble equivalent of system calls are all handled by the portal mechanism. Pebble generates specialized code for each portal to improve run-time efficiency. Portals are discussed in more detail in the following section.

3.1 Protection Domains, Portals and Threads

Each component runs in its own protection domain (*PD*). A protection domain consists of a set of pages, represented by a page table, and a set of portals, which are generalized interrupt handlers, stored in the protection domain's portal table. A protection domain may share both pages and portals with other protection domains. Figure 1 illustrates the Pebble architecture.

Portals are used to handle both hardware interrupts and software traps and exceptions. The existence of a portal from PD_A to PD_B means that a thread running in PD_A can invoke a specific entry point of PD_B (and then return). Associated with each portal is code to transfer a thread from the invoking domain to the invoked domain. Portal code copies arguments, changes stacks, and maps pages shared between the domains. Portal code is specific to its portal, which allows several important optimizations to be performed (described below).

Portals are usually generated in pairs. The call portal transfers control from domain PD_A to PD_B, and the return portal allows PD_B to return to PD_A. In the following discussion we will omit the return portal for brevity.

Portals are generated when certain resources are created (*e.g.* semaphores) and when clients connect to servers (*e.g.* when files are opened). Some portals are created at the system initialization time (*e.g.* interrupt and exception handling portals).

A scheduling priority, a stack, and a machine context are associated with each Pebble thread. When a thread traverses a portal, no scheduling decision is made; the thread continues to run, with the same priority, in the invoked protection domain. Once the thread executes in the invoked domain, it may access all of the resources available in the invoked domain, while it can no longer access the resources of the invoking domain. Several threads may execute in the same protection domain at the same time, which means that they share the same portal table and all other resources.

As part of a portal traversal, the portal code can manipulate the page tables of the invoking and/or invoked protection domains. This most commonly occurs when a thread wishes to map, for the duration of the IPC, a region of memory belonging to the invoking protection domain into the virtual address space of the invoked protection domain; this gives the thread a window into the address space of the invoking protection domain while running in the invoked protection domain. When the thread returns, the window is closed.

Such a memory window can be used to save the cost of copying data between protection domains. Variations include windows that remain open (to share pages between protection domains), windows that transfer pages from the invoking domain to the invoked domain (to implement tear-away write) and windows that transfer pages from the invoked domain to the invoker (to implement tear-away read).

Note that although the portal code may modify VM data structures, only the VM manager and the portal manager (which generates portal code) share the knowledge about these data structures. The Pebble nucleus itself is oblivious to those data structures.

3.2 Safety

Pebble implements a safe execution environment by a combination of hardware memory protection that prevents access to memory outside the protection domain, and by limiting the access to the domain's portal table. An protection domain may access only the portals it inherited from its parent and new portals that were generated on its behalf by the portal manager. The portal manager may restrict access to new portals in conjunction with the name server. A protection domain cannot transfer a portal it has in its portal table to an unrelated domain. Moreover, the parent domain may intercept all of its child portal calls, including calls that indirectly manipulate the child's portal table, as described in Section 6.

3.3 Server Components

As part of the Pebble philosophy, system services are provided by operating system server components, which run in user mode protection domains. Unlike applications, server components are trusted, so they may be granted limited privileges not afforded to application components. For example, the scheduler runs with interrupts disabled, device drivers have device registers mapped into their memory region, and the portal manager may add portals to protection domains (a protection domain cannot modify its portal table directly).

There are many advantages of implementing services at user level. First, from a software engineering standpoint, we are guaranteed that a server component will use only the exported interface of other components. Second, because each server component is only given the privileges that it needs to do its job, a programming error in one component will not directly affect other components. If a critical component fails (*e.g.*, VM) the system as a whole will be affected—but a bug in console device driver will not overwrite page tables.

Additionally, as user-level servers can be interrupted at any time, this approach has the possibility of offering lower interrupt latency time. Given that server components run at user level (including interrupt-driven threads), they can use blocking synchronization primitives, which simplifies their design. This is in contrast with handlers that run at interrupt level, which must not block, and require careful coding to synchronize with the upper parts of device drivers.

3.4 The Portal Manager

The Portal Manager is the operating system component responsible for instantiating and managing portals. It is privileged in that it is the only component that is permitted to modify portal tables.

Portal instantiation is a two-step process. First, the server (which can be a Pebble system component or an application component) registers the portal with the portal manager, specifying the entrypoint, the interface definition, and the name of the portal. Second, a client component requests that a portal with a given name be opened. The portal manager may call the name server to identify the portal and to verify that the client is permitted to open the portal. If the name server approves the access, the portal manger generates the code for the portal, and installs the portal in the client's portal table. The portal number of the newly generated portal is returned to the client. A client may also inherit a portal from its parent as the result of a `domain_fork()`, as described in Section 4.5**.**

To invoke the portal, a thread running in the client loads the portal number into a register and traps to the nucleus. The trap handler uses the portal number as an index into the portal table and jumps to the code associated with the portal. The portal code transfers the thread from the invoking protection domain to the invoked protection domain and returns to user level. As stated above, a portal transfer does not involve the scheduler in any way. (Section 5.4 describes the only exception to this rule.)

Portal interfaces are written using a (tiny) interface definition language, as described in Section 4.4. Each portal argument may be processed or transformed by portal code. The argument transformation may involve a function of the nucleus state, such as inserting the identity of the calling thread or the current time. The argument transformation may also involve other servers. For example, a portal argument may specify the address of a memory window to be mapped into the receiver's address space. This transformation requires the manipulation of data structures in the virtual memory server.

The design of the portal mechanism presents the following conflict: on one hand, in order to be efficient, the argument transformation code in the portal may need to have access to private data structures of a trusted server (*e.g.*, the virtual memory system); on the other hand, trusted servers should be allowed to keep their internal data representations private.

The solution we advocate is to allow trusted servers, such as the virtual memory manager, to register argument transformation code templates with the portal manager. (Portals registered by untrusted services would be required to use the standard argument types.) When the portal manager instantiates a portal that uses such an argument, the appropriate type-specific code is generated as part of the portal. This technique allows portal code to be both efficient (by inlining code that transforms arguments) and encapsulated (by allowing servers to keep their internal representations private). Although portal code that runs in kernel mode has access to server-specific data structures, these data structures cannot be accessed by other servers. The portal manager currently supports argument transformation code of a single trusted server, the virtual memory server.

3.5 Scheduling and Synchronization

Because inter-thread synchronization is intrinsically a scheduling activity, synchronization is managed entirely by the user-level scheduler. When a thread creates a semaphore, two portals (for *P* and *V*) are added to its portal table that transfer control to the scheduler. When a thread in the domain invokes *P*, the thread is transferred to the scheduler; if the *P* succeeds, the scheduler returns. If the *P* fails, the scheduler marks the thread as blocked and schedules another thread. A *V* operation works analogously; if the operation unblocks a thread that has higher priority than the invoker, the scheduler can block the invoking thread and run the newly-awakened one.

3.6 Device Drivers and Interrupt Handling

Each hardware device in the system has an associated semaphore used to communicate between the interrupt dispatcher component and the device driver component for the specific device.

In the portal table of each protection domain there are entries for the portals that corresponds to the machine's hardware interrupts. The Pebble nucleus includes a short trampoline function that handles all exceptions and interrupts. This code first determines the portal table of the current thread and then transfers control to the address that is taken from the corresponding entry in this portal table. The nucleus is oblivious to the specific semantics of the portal that is being invoked. The portal that handles the interrupt starts by saving the processor state on the invocation stack (see Section 5.1), then it switches to the interrupt stack and jumps to the interrupt dispatcher. In other words, this mechanism converts interrupts to portal calls.

The interrupt dispatcher determines which device generated the interrupt and performs a *V* operation on the device's semaphore. Typically, the device driver would have left a thread blocked on that semaphore. The *V* operation unblocks this thread, and if the now-runnable thread has higher priority than the currently running thread, it gains control of the CPU, and the interrupt is handled immediately. Typically, the priority of the interrupt handling threads corresponds to the hardware interrupt priority in order to support nested interrupts. The priority of the interrupt handling threads is higher than all other threads to ensure short handling latencies. In this way, Pebble unifies interrupt priority with thread priority, and handles both in the scheduler. A pictorial example of this process is found in Figure 1.

Note that Pebble invokes the interrupt dispatcher promptly for all interrupts, including low priority ones. However, the interrupt handling thread is scheduled only if its priority is higher than the currently running thread.

Only a small portion of Pebble runs with interrupts disabled, namely portal code, the interrupt dispatcher, and the scheduler. This is necessary to avoid race conditions due to nested exceptions.

3.7 Low and Consistent Interrupt Latency

Pebble provides low and consistent interrupt latency by design, since most servers (except the interrupt dispatcher and the scheduler) run with interrupts enabled. The interrupt-disabled execution path in Pebble is short, since portal code contain no loops, and the interrupt dispatcher and the scheduler are optimized for speed. User code cannot increase the length of the longest interrupt-disabled path, and thus cannot increase the interrupt latency. In previous work we included details on the interrupt handling mechanism in Pebble, along with measurements of the interrupt latency on machines with differering memory hierarchies [Bruno99]. In particular, the interrupt latency on the MIPS R5000 processor that is used in this paper is typically 1200-1300 cycles from the exception until the scheduling of the user-level handling thread.

3.8 Non-Stop Systems

Non-stop (or high-availability) systems are characterized by the ability to run continuously over extended periods of time and support dynamic updates. For example, some systems, such as telephone switches, are expected to run for years without unscheduled down time. Pebble is especially suited for these systems, since most system functionality may be replaced dynamically by loading new servers and modifying portal tables. The only component that cannot be replaced is the nucleus, which provides only minimal functionality.

4 Portals and Their Uses

Portals are used for multiple purposes in Pebble. In this section, we describe a few of their applications.

4.1 Interposition and Layering

One technique for building flexible system is to factor it into components with orthogonal functionality that can be composed in arbitrary ways. For example, distributed shared memory or persistent virtual memory can be implemented as a layer on top of a standard virtual memory service. Or, altered semantics can be offered by layering: the binary interface of one operating system can be emulated on another operating system by intercepting system calls made by an application written for the emulated system and implementing them through the use of native system calls.

The portal mechanism supports this development methodology very nicely. Because the portal mechanism is used uniformly throughout the system, and a portal performs a user-level to user-level transfer, service components can be designed to both accept and use the same set of portals.

For example, the primary task of a virtual memory manager is to accept requests for pages from its clients and service them by obtaining the pages from the backing store. When a client requests a page, the virtual memory manager would read the page from the backing store and return it to the client via a memory window operation. A standard virtual memory service implementation would support just this protocol, and would typically be configured with a user application as its client and the file system as its backing store server.

However, the backing store could be replaced with a distributed shared memory (DSM) server, which would have the same interface as the virtual memory manager: it would accept page requests from its client, obtain the pages from its backing store (although in this case the backing store for a page might be the local disk or another remote DSM server) and return the page to its client via a memory window operation. By implementing the DSM server using the standard virtual memory interface, it can be layered between the VM and the file system. Other services, such as persistent virtual memory and transactional memory, can be added this way as well.

When a page fault takes place, the faulting address is used to determine which portal to invoke. Typically a single VM fault handler is registered for the entire range of an application's heap, but this need not be the case. For example, a fault on a page in a shared memory region should be handled differently than a fault on a page in a private memory region. By assigning different portals to subranges of a protection domain's address space, different virtual memory semantics can be supported for each range.

4.2 Portals Can Encapsulate State

Because portal code is trusted, is specific to its portal, and can have private data, portal code can encapsulate state associated with the portal that need not be exposed to either endpoint. The state of the invoking thread is a trivial example of this: portal code saves the thread's registers on the invocation stack (see Section 5.1), and restores them when the thread returns. On the flip side, data used only by the invoked protection domain can be embedded in the portal where the invoker cannot view or manipulate it. Because the portal code cannot be modified by the invoking protection domain, the invoked protection domain is ensured that the values passed to it are valid. This technique frequently allows run-time demultiplexing and data validation code to be removed from the code path.

As an example, in Pebble, portals take the place of file descriptors. An `open()` call creates four portals in the invoking protection domain, one each for reading, writing, seeking and closing. The code for each portal has embedded in it a pointer to the control block for the file. To read the file, the client domain invokes the `read` portal; the portal code loads the control block pointer into a register and transfers control directly to the specific routine for reading the underlying object (disk file, socket, etc.). No file handle verification needs to be done, as the client is never given a file handle; nor does any demultiplexing or branching based on the type of the underlying object need to be done, as the appropriate read routine for the underlying object is invoked directly by the portal code. In this way, portals permit run-time checks to be "compiled out," shortening the code path.

To be more concrete, the `open()` call generates four consecutive portals in the caller's portal table. `Open()` returns a file descriptor, which corresponds to the index of the first of the four portals. The `read()`, `write()`,

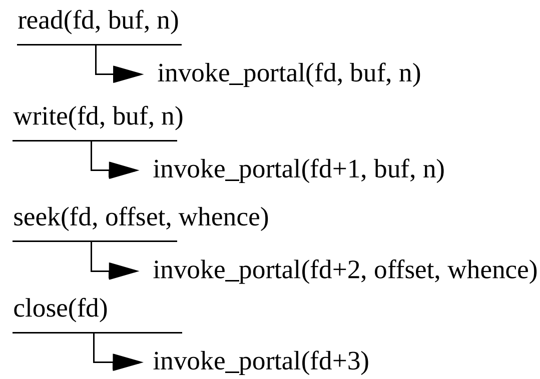


Figure 2. Implementing file descriptors with portals

`seek()` and `close()` calls are implemented by library routines, which invoke the appropriate portals, as seen in Figure 2. `invoke_portal()` invokes the portal that is specified in its first argument. (Note that the portal code of `read` and `write` may map the buffer argument in a memory window to avoid data copying.)

4.3 Short-Circuit Portals

In some cases the amount of work done by portal traversal to a server is so small that the portal code itself can implement the service. A short-circuit portal is one that does not actually transfer the invoking thread to a new protection domain, but instead performs the requested action inline, in the portal code. Examples include simple "system calls" to get the current thread's ID and read the high resolution cycle counter. The TLB miss handler (which is in software on the MIPS architecture, the current platform for Pebble) is also implemented as a short-circuit portal.

Currently, semaphore synchronization primitives are implemented by the scheduler and necessitate portal traversals even if the operation does not block. However, these primitives are good candidates for implementation as hybrid portals. When a *P* operation is done, if the semaphore's value is positive (and thus the invoking thread will not block), the only work done is to decrement the semaphore, and so there is no need for the thread to transfer to the scheduler. The portal code could decrement the semaphore directly, and then return. Only in the case where the semaphore's value is zero and the thread will block does the calling thread need to transfer to the scheduler. Similarly, a *V* operation on a semaphore with a non-negative value (*i.e.*, no threads are blocked waiting for the semaphore) could be performed in a handful of instructions in the portal code itself.

Although these optimizations are small ones (domain transfer takes only a few hundred cycles), operations that are on the critical path can benefit from even these small savings.

4.4 Portal Specification

The portal specification is a string that describes the behavior of the portal. It controls the generation of portal code by the portal manager. The portal specification includes the calling conventions of the portal, which registers are saved, whether the invoking domain shares a stack with the invoked domain, and how each arguments is processed.

The first character in the specification encodes the portal's stack manipulation. For example, `"s"` denotes that the invoking domain shares its stack with the invoked domain. `"n"` denotes that the invoked domain allocated a new stack. The second character specifies the amount of processor state that is saved or restored. For example, `"m"` denotes that only minimal state is saved, and that the invoking domain trusts the invoked domain to obey the C calling convention. `"p"` denotes that partial state is saved, and that the invoking domain does not trust the invoked domain to retain the values of the registers required by the C calling convention. The rest of the specification contains a sequence of single character function codes, that specify handling of the corresponding parameters. For example, the template `"smcwi"` specifies a shared stack, saving minimal state, passing a constant in the first parameter, passing a one-page memory window in the second parameter, and passing a word without transformation in the third parameter. This template is used by the `read` and `write` portals.

4.5 Portal Manipulations

As described earlier, portals are referred to by their index in the local portal table. A portal that is available in a particular portal table cannot be exported to other protection domains using this index. A protection domain may access only the portals in its portal table. These properties are the basis for Pebble safety. When a thread calls `fork()`, it creates a new thread that executes in the same protection domain as the parent. When a thread calls `domain_fork()`, it creates a new protection domain that has a copy of the parent domain's portal table. The parent may modify the child's portal table to allow portal interposition, which is described in Section 6.

5 Implementation Issues

In this section we discuss some of the more interesting implementation details of Pebble.

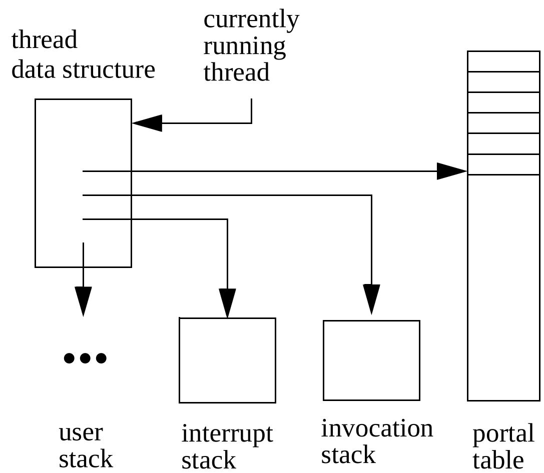


Figure 3. Pebble nucleus data structures

5.1 Nucleus Data Structures

The Pebble nucleus maintains only a handful of data structures, which are illustrated in Figure 3. Each thread is associated with a `Thread` data structure. It contains pointer to the thread's current portal table, user stack, interrupt stack and invocation stack. The user stack is the normal stack that is used by user mode code. The interrupt stack is used whenever an interrupt or exception occurs while the thread is executing. The interrupt portal switches to the interrupt stack, saves state on the invocation stack and calls the interrupt dispatcher server.

The invocation stack keeps track of portal traversals and processor state. The portal call code saves the invoking domain's state on this stack. It also saves the address of the corresponding return portal on the invocation stack. The portal return code restores the state from this stack.

The portal table pointer in the `Thread` data structure is portal table of the domain that the thread is currently executing in. It is changed by the portal call and restored by the portal return.

5.2 Virtual Memory and Cache

The virtual memory manager is responsible for maintaining the page tables, which are accessed by the TLB miss handler and by the memory window manipulation code in portals. The virtual memory manager is the only component that has access to the entire physical memory. The current implementation of Pebble does not support demand-paged virtual memory.

Pebble implementation takes advantage of the MIPS tagged memory architecture. Each protection domain is allocated a unique ASID (address space identifier), which avoids TLB and cache flushes during context switches. Portal calls and returns also load the mapping of the current stack into TLB entry 0 to avoid a certain TLB miss.

On the flip side, Pebble components run in separate protection domains in user mode, which necessitates careful memory allocation and cache flushes whenever a component must commit values to physical memory. For example, the portal manager must generate portal code so that it is placed in contiguous physical memory.

5.3 Memory Windows

The portal code that opens a memory window updates an access data structure that contains a vector of counters, one counter for each protection domain in the system. The vector is addressed by the ASID of the corresponding domain. The counter keeps track of the number of portal traversals into the corresponding domain that passed this page in a memory window. This counter is incremented by one for each portal call, and is decremented by one for each portal return. The page is accessible if the counter that corresponds with the domain is greater than zero. We must use counters and not bit values for maintaining page access rights, since the same page may be handed to the same domain by multiple concurrent threads.

The page table contains a pointer to the corresponding access data structure, if any. Only shared pages have a dedicated access data structure.

The portal code does not load the TLB with the mapping of the memory window page. Rather, the TLB miss handler consults this counter vector in order to verify the access rights to this page. This arrangement saves time if the shared window is passed to another domain without being touched by the current domain. The portal return code must remove the corresponding TLB entry when the counter reaches zero.

5.4 Stack Manipulations

The portal call may implement stack sharing, which does not require any stack manipulations. The invoked domain just uses the current thread's stack.

If the portal call requires a new stack, it obtains one from the invoked domain's stack queue. In this case, the invoked protection domain must pre-allocate one or more stacks and notify the portal manger to place them in the domain's stack queue. The portal call dequeues a new stack from the invoked domain's stack queue. If the stacks queue is empty, the portal calls the scheduler and waits until a stack becomes available. The portal return enqueues the released stack back in the stack queue. If there are any threads waiting for the stack, the portal return calls the scheduler to pick the first waiting thread and allow it to proceed in its portal code.

The portal that calls the interrupt dispatcher after an interrupt switches the stack to the interrupt stack, which is always available in every thread.

5.5 Footprint

The Pebble nucleus and the essential components (interrupt dispatcher, scheduler, portal manager, real-time clock, console driver and the idle task) can fit into about 70 pages (8KB each). Pebble does not support shared libraries yet, which cause code duplication among components. Each user thread has three stacks (user, interrupt and invocation) which require three pages, although the interrupt and invocation stacks could be placed on the same page to reduce memory consumption. In addition, fixed size pages inherently waste memory. This could be alleviated on segmented architectures.

6 Portal Interposition

An important aspect of component-based system is the ability to *interpose* code between any client and its servers. The interposed code can modify the operation of the server, enforce safety policies, enable logging and error recovery services, or even implement protocol stacks and other layered system services.

Pebble implements low-overhead interposition by modifying the portal table of the controlled domain. Since all interactions between the domain and its surroundings are implemented by portal traversals, it is possible to place the controlled domain in a comprehensive sandbox by replacing the domain's portal table. All of the original portals are replaced with portal stubs, which transfer to the interposed controlling domain. The controlling domain intercepts each portal traversal that takes place, performs whatever actions it deems necessary, and then calls the original portal. Portal stubs pass their parameters in the same way as the original portals, which is necessary to maintain the semantics of the parameter passing (*e.g.* windows). Actually, portal stubs are regular portals that pass the corresponding portal index in their first argument. The controlling domain does not have to be aware of the particular semantics of the intercepted portals; it can implement a transparent sandbox by passing portal parameters verbatim.

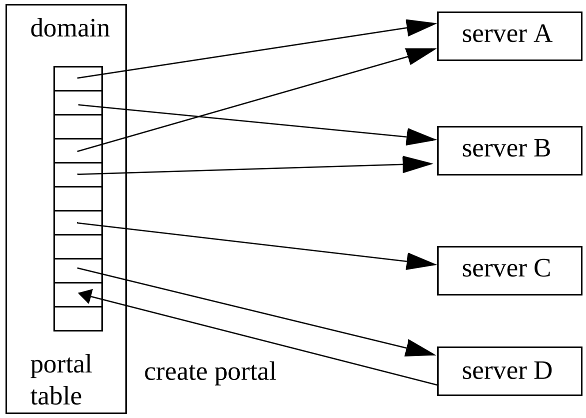
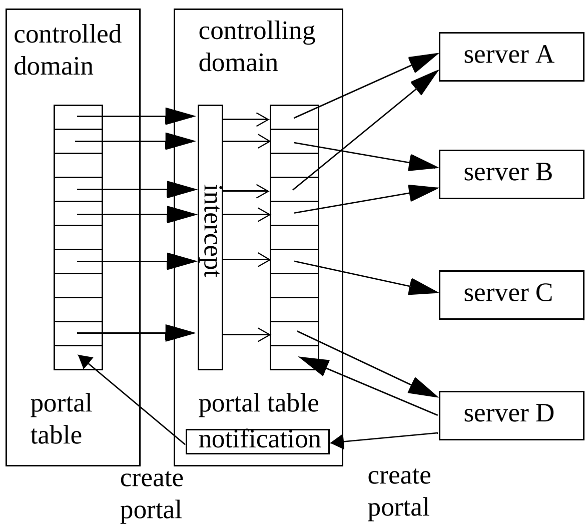


Figure 4. Original portal configuration (above) and with portal interposition (below)



The top diagram of Figure 4 illustrates the configuration of the original portal table without interposition, where the domain calls its servers directly. The bottom diagram shows the operation of portal interposition. In this case, all of the portals in the controlled domain call the controlling domain, which makes the calls to the servers.

However, one-time modification of the controlled domain's portal table is not enough. Many servers create new portals dynamically in their client's portal table, and then return an index to the newly created portal back to the client. Since the controlling domain calls the server, the server creates new portals in the controlling domain's table. The controlling domain is notified by the portal manager that a new portal was created in its portal table. The notification portal completes the process by creating a portal stub in the controlled domain's table with the same index as in controlling domain table. The portal stub calls the controlling domain and passes the parameters in the same way as the original portal. In this way, the controlling domain implements a robust sandbox around the controlled domain, without actually understanding the semantics of the controlled domain portals.

There are a few comments about this interposition mechanism. First, the controlled domain cannot detect that its portals are diverted nor can it thwart the interposition in any way. This mechanism is similar to the Unix I/O redirection, in which a child process accesses standard file descriptor (*e.g.*, 0, 1 and 2), which are redirected by the parent process. Second, portal interposition is more comprehensive than Unix I/O redirection, since we can control *all* interactions between the controlled domain and its environment. Third, interposition can be recursive: a controlling domain interposes the portals of a child domain, which does the same to its child, *ad infinitum*. The last comment deals with the semantics of certain system services, like `fork()` and `sbrk()`, which change the internal state of the calling domain; these are somewhat tricky to implement in the face of transparent interposition. We have had to make special accommodations to allow the controlling domain to issue them on behalf of the controlled domain.

6.1 Implementing a Transparent Sandbox by Portal Interposition

The Appendix contains a code excerpt from a program that implements a transparent sandbox around its child domain. The program counts the number of times each portal was called by the child domain, and completes all child portal traversals by calling the appropriate server. It is a fully functional program; we omitted only error handling code, due to space constraints. When run on our test hardware (see Section 7, below) the overhead of this process is 1511 machine cycles for one iteration (two `sem_wait()` and two `sem_post()`), which is roughly twice the execution time of the original code without interposition.

The program starts by calling `portal_notify()`, which registers the routine `notify()` with the portal manager. Any modification to the calling domain's portal table will call `notify()` immediately even before the portal that caused it has returned. `Portal_notify()` is necessary to handle any portal call that the parent executed on behalf of the child which created a new portal in the parent's portal table. This new portal should be replicated also in the child's portal table to ensure correct operation. The above situation occurs in the example when the parent executes `sem_create()` on behalf of the child.

The `notify()` routine receives the template of the newly created portal and its position in the portal table. It creates a portal in the child's portal table at the same position. The portal's template is modified to pass the portal number as the first argument.

The program proceeds to create a child domain by `domain_fork()`. The child starts with a copy of the parent's portal table. However, all of the entries in the child's portal table now point at the `intercept()` routine in the parent domain. The first argument to the `intercept()` routine is the index of the called portal in the portal table. This routine increments the counters and then performs the required action by invoking the portal with the same index in the parent domain. `invoke_portal()` let applications invoke a specific portal in the caller's portal table. The `intercept()` routine assumes that portals have no more than five parameters.

The child domain executes the `measure()` routine, which measures the execution time of a semaphore ping-pong between two threads in the same domain. The `hrtime()` function returns the current value of the high-resolution timer, which is incremented every two machine cycles. `Measure()` creates two semaphores by calling `sem_create()`. The scheduler creates two new portals for each semaphore in the parent domain, which calls `notify()` to create the corresponding stubs in the child domain's portal table.

7 Performance Measurements

In this section we measure the performance of Pebble and, where possible, compare it with OpenBSD running on the same hardware. The test hardware is an Algorithmics P-5064 board, which includes a 166 MHz MIPS R5000 processor with 32 KB instruction + 32 KB data level one cache (two way set associative), one megabyte integrated level two cache and 64MB of memory. We ran version 2.4 of OpenBSD.

Times were measured using the high-resolution on-chip timer, which is incremented every two clock cycles. All results are presented in terms of elapsed machine cycles, not elapsed time, as our tests generally fit into the level one or level two cache. As long as cache memory speed scales with processor speed, cycle-based results will remain meaningful. To convert cycle counts to elapsed time, multiply by the cycle time (6 ns).

As the code size of Pebble is very small, and the cache associativity of the level one cache is low (two-way), the performance of Pebble is very dependent on how code and data is placed in the cache. Out of a sense of fairness, in our experiments we specifically do not make any attempt to control cache layout. We believe that with careful tuning of the cache layout, we could reduce the number of cache misses and conflicts. Given the performance results we have seen to date, we have felt little need to go to this effort.

The context switch, pipe latency, and semaphore latency tests were adapted from the hBench:OS test suite [Brown98]. All tests on Pebble were run 10,000 times. The context switch and pipe latency times presented for OpenBSD were the 80% trimmed mean (excluding the smallest 10% and largest 10% of the measurements) of twenty results of 10,000 iterations, as per the hBench:OS measurement methodology. In all cases the standard deviation for Pebble measurements was less than 1%.

7.1 IPC

A naive implementation of inter-process communication (IPC) will emulate the behavior of a remote procedure call (RPC), marshalling all arguments into a buffer, copying the buffer from the invoking protection domain to the invoked protection domain, unmarshalling them, and then calling the server function. Several common optimizations can be performed that greatly improve the performance of IPC.

First, the amount of data transmitted in an IPC follows a bimodal distribution [Bershad89]; either a small number of bytes are sent (in which case they can be passed in registers) or a large number of bytes are sent (in which case it may make more sense to transfer the data using virtual memory mapping operations).

In this test we measure the cost of performing an IPC when all data fits into registers, when a one-page memory window is passed to the invoked domain (but the invoked domain does not access the page), and when the one-page memory window is written by the invoked domain. Because virtual memory and the TLB are managed in software on the MIPS, the memory management unit is not involved if when passing a memory window if the window is never used, although there is some additional portal overhead. When the window is used in the invoked domain, a TLB fault takes place, and the memory management unit comes into play. Moreover, the portal code may have to remove the resulting TLB entry on return.

Simply measuring the per-leg cost of an IPC between two domains does not tell the entire story. In a system that has been factored into components, we may find that a client request to service *A* causes *A* to make a request of *A′*, *A′* to make a request of *A″*, and so on, until the initial request is finally satisfied. For example, a client page fault generates a request to its VM service, then makes a request of the file system, which then makes a request the disk driver to bring the page into memory. Although simple IPC between two protection domains must be cheap, it is also critical that when a cascade of IPCs takes place performance does not drop precipitously.

In this test we measure the time to perform an IPC to the same domain and return ($A \rightarrow A \rightarrow A$), the time required to perform an IPC to a second domain and return ($A \rightarrow B \rightarrow A$), an IPC involving three domains ($A \rightarrow B \rightarrow C \rightarrow B \rightarrow A$) and so on, up to a total of eight domains. We used the portal specification `“npciii”` (no window) and `“npcwii”` (with memory window), which means that a new stack was allocated on call and reclaimed on the return. Also, all processor registers that should be preserved across calls according to the C calling convention were saved on call and restored on return. See Section 4.4 for a description of portal specification. The results are presented as the per-leg (one-way) time, in cycles.

As a point of comparison, we included the time required to perform a “null” short-circuit portal traversal (user level → nucleus → user level). This is the Pebble equivalent to a “null” system call, and can be thought of as the minimum time required to enter and leave the nucleus. Results of these tests are found in Table 1. In all cases. parameters are passed only in registers and not on the stack.

n domains	no window	window	window + fault
short-circuit	45	—	—
1	114	133	135
2	114	134	185
4	118	139	190
8	133	153	209

Table 1. IPC in Pebble, new stack and partial save, All times in CPU cycles, the mean of 10,000 runs.

We see that the times per leg with no window and with an unused window remains roughly constant as the number of domains traversed increases, at about 114 and 135 cycles; the overhead of passing a window through a portal is thus 21 machine cycles. The time per leg increases above 4 domains due to cache contention. When the memory window is used, the cost increases by about 50 cycles, which is the time required to handle a TLB fault and then remove the TLB entry on return from the IPC. The one outlier is in the single domain case, where there is no TLB fault at all; this is because the page is already mapped in the domain (as there is only one domain).

An optimization can be performed if the invoking domain trusts the invoked domain (as would be the case with an application invoking a system service). The two can share a stack, saving the costs of allocating a stack from a pool in the invoked protection domain and copying data to the new stack. Also, no additional processor registered are saved on the call, since the invoking domain trusts the invoked domain to save and restore those registers. We used the portal specifications `“smciii”` and `“smcwii”`. Even in the tested case, where no data is passed on the stack, this optimization has a significant performance benefit, as seen in Table 2.

n domains	no window	window	window + fault
1	95	115	118
2	95	116	168
4	95	116	168
8	98	120	182

Table 2. IPC in Pebble, shared stack and minimal save. In CPU cycles, the mean of 10,000 runs

The savings of this optimization are measured here to be about 20 cycles, which reduces the per-leg time by 17%. In addition, by sharing stacks between invoking and invoked protection domains, the number of stacks, and hence amount of memory, needed by the system is decreased, which is an absolute good.

Pebble IPC time is slightly higher than Aegis, an exokernel, on MIPS processors [Engler95]. Aegis performs a minimal one-way protected control transfer in about 36 cycles on MIPS R2000 and R3000 processors, and performs a null system call without a stack in about 40 cycles. Pebble’s IPC takes longer since it maintains an invocation stack, which enables easy scheduling of the thread.

7.2 Context Switch

As described above, portal traversal does not involve a scheduling decision. In this section we show the cost of a context switch in Pebble.

We measure Pebble context switch cost in two ways, first using Pebble's explicit yield primitive, and then by passing a one-byte token around a ring of pipes. The latter test was derived from hBench:OS, and was used to compare the performance of Pebble with OpenBSD. In both cases a number of protection domains, with a single thread each, are arranged in a ring, and scheduled in turn. Measurements are found in Table 3.

n domains	Pebble yield	Pebble pipe	OpenBSD pipe
2	425	411	1195
4	549	963	2093
8	814	1162	2179

Table 3. Context switch times, Pebble vs. OpenBSD. In CPU cycles, the mean of at least 10,000 runs.

We see that the cost of an explicit yield increases with the number of protection domains, up to a certain point, and then levels off. As the work done by the scheduler in this case is independent of the number of processes (it simply selects the next thread from the ready queue), the increase in time is due to cache effects: as we grow out of the level one cache, we rely more on the level two cache, to the point where we are running almost entirely out of the level two cache (at six protection domains). We would expect to see a similar jump at the point where we begin to overflow the one-megabyte level two cache.

The OpenBSD pipe test shows similar behavior, leveling off at four protection domains and roughly 2200 machine cycles.

7.3 Pipe Latency

This test measures the time required to pass a single byte through pipes connecting a ring of processes. Each value represents the time to transfer one byte between two adjacent processes, and includes the context switch time. By measuring the time required to transmit a single byte, we capture the overhead associated with using pipes; the more data that is sent, the more the data copy time will mask pipe costs. Results are found in Table 4.

n domains	Pebble pipe	OpenBSD pipe
2	1310	3088
4	1914	3979
8	2061	4055

Table 4. Pipe latency, Pebble vs. OpenBSD. In CPU cycles, the mean of at least 10,000 runs.

We see that, as with the context switch times, the OpenBSD pipe time increases up to five domains, and then levels off. The difference between the numbers in Table 4 and Table 3 gives us the time required to transfer data through a pipe on each system. On OpenBSD the pipe overhead is roughly 2000 cycles; on Pebble it is approximately half that.

7.4 Semaphore Acquire/Release

This test is very similar to the test in Section 7.3, but instead of using pipes we use semaphores. A number of processes are arranged in a ring, and are synchronized by means of n semaphores. Each process performs a *V* operation on its right semaphore and then a *P* operation on its left semaphore. Each value in the table represents the time to release a semaphore in process *p* and acquire it in process *(p + 1) mod n* around a ring of n processes, including the context switch time. Results are found in Table 5.

n domains	Pebble semaphore	OpenBSD semaphore
2	781	2275
4	942	3415
8	1198	5091

Table 5. Semaphore acquire/release, Pebble vs. OpenBSD. In CPU cycles, the mean of 10,000 runs.

When there are two processes the difference between Pebble and OpenBSD is roughly 1500 cycles, 1000 cycles of which can be attributed to the difference in context switch times. As the number of domains (and thus semaphores) increases, the difference widens; because Pebble's semaphores are a highly optimized key system primitive, and OpenBSD's semaphores are not, we believe that this is due to a restriction in the implementation of OpenBSD semaphores, and is not a reflection of the difference in system structure.

7.5 Portal Generation

Table 6 shows the portal generation time for two typical portals. This is the time measured by an application program, including all overheads incurred by the portal manager. The first portal (with specification `"smcii"`) is typically used to call a trusted server with only integer arguments. The second portal (with specification `"npcwi"`) is typically used to call an untrusted server with a memory window argument. See Section 4.4 for additional explanations of portal specifications.

portal spec.	portal len (instr.)	time (cycles)	cycles per instr.
smcii	64	7282	114
npcwi	112	8593	77

Table 6. Portal generation time.

Table 6 indicates that portal generation time is relatively fast. An examination of the portal manager reveals that portal generation time includes a large fixed overhead for interpretation of the specification string and for cache flushing. We can reduce this time by employing various techniques used for run-time code generation, *e.g.*, the techniques used by `VCODE` [Engler96].

8 Status and Future Work

The Pebble nucleus and a small set of servers (scheduler, portal manager, interrupt dispatcher, and minimal VM) and devices (console and clock) currently run on MIPS-based single-board computers from Algorithmics. We support both the P-4032 (with QED RM5230 processor) and P-5064 (with IDT R5000 or QED RM7000 processors). We are currently porting Ethernet and SCSI device drivers to Pebble.

Next we plan to port Pebble to the Intel x86 to verify that Pebble mechanisms and performance advantages are indeed architecture independent. We also plan to implement a demand-paged virtual memory system. Building a high-performance VM system for Pebble is a challenge, since the servers cannot (and should not) share data structures freely. We also plan to port a TCP/IP stack to Pebble and compare its performance with similar user-level protocol stacks.

In addition to the Intel x86 port, we plan to port to a symmetric multiprocessor and to an embedded processor such as the StrongARM. We also plan to investigate the various processor architecture support for component-based systems such as Pebble.

9 Summary

Pebble provides a new engineering trade-off for the construction of efficient component-based systems, using hardware memory management to enforce protection domain boundaries, and reducing the cross domain transfer time by synthesizing custom portal code. Pebble enhances flexibility by maintaining a private portal table for each domain. This table can be used to provide different implementations of system services, servers and portal interposition for each domain. In addition, portal interposition allows running untrusted code in a robust sandbox with an acceptable overhead while using unsafe languages such as C.

Having a small nucleus with minimal functionality enhances system modularity, while it enables non-stop systems to modify their behavior by integrating new servers on-the-fly.

In this paper we showed that Pebble is much faster than OpenBSD for a limited set of system-related micro-benchmarks. Pebble efficiency does not stem from clever low-level highly-optimized code; rather it is a natural consequence of custom portal synthesis, judicious processor state manipulations at portal traversals, encapsulating state in portal code, and direct transfer of control from clients to their servers without scheduler intervention.

Pebble can be used to build systems that are more flexible, as safe as, and have higher performance than conventionally constructed systems.

Acknowledgments

The authors would like to thank the anonymous referees for their insightful comments.

References

[Accetta86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, “Mach: A New Kernel Foundation for UNIX Development,” *Proc. Summer 1986 USENIX Conf.*, pp. 93–112 (1986).

[Bershad89] B. Bershad, T. Anderson, E. Lazowska, H. Levy, “Lightweight Remote Procedure Call,” *Proc. 12th SOSP*, pp. 102–113 (1989).

[Bershad95] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, S. Eggers, “Extensibility, Safety, and Performance in the SPIN Operating System,” *Proc. 15th SOSP*, pp. 267–284 (1995).

[Brown98] A. Brown, M. Seltzer, “Operating System Benchmarking in the Wake of lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture,” *Proc. 1997 SIGMETRICS*, pp. 214–224 (1997).

[Bruno99] J. Bruno, J. Brustoloni, E. Gabber, A. Silberschatz, C. Small, “Pebble: A Component-Based Operating System for Embedded Applications,” *Proc. USENIX Workshop on Embedded Systems*, Cambridge, MA (1999).

[Custer92] H. Custer, *Inside Windows NT*, Microsoft Press, Redmond, WA (1992).

[Dijkstra68] E. W. Dijkstra, “The Structure of “THE” Multiprogramming System,” CACM, Volume 11, Number 5, pp. 341-346 (1968).

[Dorward97] S. Dorward, R. Pike, D. Presotto, D. Ritchie, H. Trickey, P. Winterbottom, “Inferno,” *Proc. IEEE Compcon 97*, pp. 241–244 (1997).

[Engler95] D. Engler, M. Frans Kaashoek, J. O’Toole Jr., “Exokernel: An Operating System Architecture for Application-Level Resource Management”, *Proc. 15th SOSP*, pp. 251-266 (1995).

[Engler96] D. Engler, “`VCODE`: A Retargetable, Extensible, Very Fast Dynamic Code Generation System”, *Proc. Conference on Programming Language Design and Implementation (PLDI’96)*, pp. 160-170 (1996).

[Ford96] B. Ford, M. Hibler, J,. Lepreau, P. Tullmann, G. Back, S. Clawson, “Microkernels Meet Recursive Virtual Machines,” *Proc. 2nd OSDI*, pp. 137–151 (1996).

[Golub90] D. Golub, R. Dean, A. Forin, R. Rashid, “UNIX as an Application Program,” *Proc. 1990 Summer USENIX*, pp. 87–96 (1990).

[Gosling96] J. Gosling, B. Joy, G. Steele, *The Java™ Language Specification*, Addison-Wesley, Reading, MA (1996).

[Helander98] J. Helander and A. Forin, “MMLite: A Highly Componentized System Architecture”, *Proc. 8th ACM SIGOPS European Workshop,* Sintra, Portugal (1998).

[Intel94] Intel Corp., *Pentium Family User’s Manual Volume 3: Architecture and Programming Manual* (1994).

[Kaashoek97] M. F. Kaashoek, D. Engler, G. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, “Application Performance and Flexibility on Exokernel Systems,” *Proc. 16th SOSP*, pp. 52–65 (1997).

[Liedtke95] J. Liedtke, “On Micro-Kernel Construction,” *Proc. 15th SOSP*, pp. 237–250 (1995).

[Liedtke97] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, T. Jager, “Achieved IPC Performance,” *Proc. 6th HotOS*, pp. 28–3 (1997).

[Massalin92] H. Massalin, *Synthesis: An Efficient Implementation of Fundamental Operating System Services*, Ph.D. thesis, Columbia University Department of Computer Science, New York, NY (1992).

[Mitchell94] J. G. Mitchel *et al*, “An Overview of the Spring System”, *Proc. Compcon Spring 1994*, pp. 122-131 (1994).

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, “Plan 9 from Bell Labs,” *Proc. Summer 1990 UKUUG Conf.*, pp. 1–9 (1990).

[Probert91] D. Probert, J. Bruno, M. Karaorman, “SPACE: A New Approach to Operating System Abstractions,” *Proc. Intl. Workshop on Object Orientation in Operating Systems (IWOOS)*, pp. 133–137 (1991), Also available on-line at ftp.cs.ucsb.edu/pub/papers/space/iwooos91.ps.gz

[Pu95] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, K. Zhang, “Optimistic Incremental Specialization: Streamlining a Commercial Operating System,” *Proc. 15th SOSP*, pp. 314–324, (1995).

[Rozier88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, W. Neuhauser. “Chorus Distributed Operating System.” *Computing Systems 1*(4), pp. 305–370 (1988).

[Seltzer96] M. Seltzer, Y. Endo, C. Small, K. Smith, “Dealing With Disaster: Surviving Misbehaved Kernel Extensions,” *Proc. 2nd OSDI*, pp. 213–227 (1996).

[Veitch96] A. C. Veitch and N. C. Hutchinson, “Kea - A Dynamically Extensible and Configurable Operating System Kernel”, Proc. 3rd Conference on Configurable and Distributed Systems (ICCDS’96), Annapolis, Mariland (1996).

Appendix: Implementing a Transparent Sandbox by Portal Interposition

```
/* see Section 6.1 for explanations */
#include <pebble.h>

#define  N  10000

int child_asid;
int count[NPORTALS];

/* child domain runs this routine */
void measure(void)
{
   int code, i;
   unsigned long long start, elapsed;
   int sem_id1, sem_id2;

   /* create semaphores */
   sem_id1 = sem_create(0);
   sem_id2 = sem_create(0);

   /* create child thread in the same
      domain */
   if ((code = fork()) == 0) {
      /* child thread wakes parent */
      sem_post(sem_id2);

      for (i = 0;; i++) {
         sem_wait(sem_id1);
         sem_post(sem_id2);
      }

      /* never reached */
      exit(1);
   }

   /* parent thread waits until child
      is active for accurate timing */
   sem_wait(sem_id2);

   /* time semaphore ping-pong with
      child */
   start = hrtime();

   for (i = 0; i < N; i++) {
      sem_post(sem_id1);
      sem_wait(sem_id2);
   }
   elapsed = 2*(hrtime() - start);

   printf("each iteration: %d \
      cycles\n", (int)(elapsed/N));
}

void dump_counters(void)
{
   int i;

   for (i = 1; i < NPORTALS; i++)
      if (count[i] != 0)
         printf("portal %d called %d\
            times\n", i, count[i]);
}

/* parent domain intercepts child por-
tal call */
int intercept(int id, int p1, int p2,
         int p3, int p4, int p5)
{
   count[id]++;
   if (id == SYS_EXIT)
      dump_counters();

   return invoke_portal(id, p1, p2,
                        p3, p4, p5);
}

/* parent domain gets notification */
int notify(int asid, int id,
      char *template)
{
   char s[NAMELEN];

   sprintf(s, "sm=%s", template+3);
   portal_create_child(child_asid,
      id, s, 0, intercept);
   return 0;
}

void main(void)
{
   portal_notify(notify);

   child_asid =
      domain_fork(intercept);
   if (child_asid == 0) {
      /* child domain */
      measure();
      exit(0);
   }

   /* parent waits here until child
      exits */
   sem_wait(sem_create(0));

   exit(0);
}
```