Pebble: A Component-Based Operating System for Embedded Applications

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

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

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

Abstract

The Pebble operating system is intended to support complex embedded applications. This is accomplished through two key features: (1) safe extensibility, so that the system can be constructed from untrusted components and reconfigured while running, and (2) low interrupt latency, which ensures that the system can react quickly to external events.

In this paper we discuss Pebble's architecture and the underlying technology used by Pebble, and include microbenchmark performance results on three MIPS target systems. The performance measurements demonstrate that Pebble is a good platform for complex embedded applications.

1 Introduction

This paper describes the Pebble operating system architecture, which is designed to be used as a platform for high-end, embedded, communicating devices constructed from reusable software components.

A *component-based approach* to building applications is becoming increasingly popular, as exemplified by the popularity of COM, Java Beans, and CORBA. The advantage of software components with clean, abstract interfaces is that they allow code to be combined and reused in ways that were not imagined by their developers, allowing the cost of development to be amortized over more uses.

When constructing a system from software components, a choice has to be made as to what method will be used to isolate the components from one another. There are three basic approaches: provide no protection between components, as exemplified by COM-based systems; provide software protection, as used by Java-based systems; and provide hardware protection, as exemplified by CORBA- and RPC-based approaches. Each method has its drawbacks. With no protection, a misbehaved component can compromise the integrity of the system, as it has access to all data and resources of the system. Software protection typically requires that the system be written in a special, safe programming language, which may not be acceptable to all developers. Hardware protection schemes have traditionally exhibited poor performance, due to the cost of switching protection domains when performing an inter-protection-domain call. However, recent work (e.g., on L4 [Liedke97] and Exokernel [Kaashoek97]) has shown that commonly held assumptions about the intrinsic performance cost of hardware protection schemes need to be reexamined.

The Pebble architecture began with the idea that operating system services should be implemented by a collection of fine-grained, replaceable user-level components. The techniques we have applied in order to get good performance from operating system components are also used by component-based applications running on Pebble, and applications share in the performance benefit provided by these techniques. The performance improvement are striking: for example, on Pebble, a one-way inter-protection domain call takes roughly 120 machine cycles, which is within an order of magnitude of the cost of performing a function call; an equivalent call when running OpenBSD (on the same hardware) takes roughly 1000-2000 machine cycles [Gabber99].

With Pebble, an application can dynamically configure the services provided by the system, and safely load new, untrusted components, written in an unsafe programming language, into the system while the system is running. Moreover, old servers may be retired gracefully when new versions of the service are introduced without disrupting the operation of the system. This capability is essential for high-availability systems that must operate continuously.

Future communication devices, such as PDA-cell phone hybrids, set-top-boxes, and routers will require just this type of dynamic configurability and the ability to safely run untrusted code. We feel that the approach that we are taking with Pebble will be valuable for building such embedded systems.

The remainder of the paper is organized as follows. Section 2 describes the Pebble philosophy. Section 3 contrasts software and hardware protection for component-based applications. Section 4 discusses the technologies used to implement Pebble, including protection domains, portals, scheduling, synchronization, device drivers and interrupt handling. Section 5 explains how portals are used to optimize file operations. Section 6 presents performance measurements of Pebble on three different MIPS-based target machines. The paper concludes with a survey of related work, current status and a summary.

2 Pebble Philosophy

The Pebble architectural philosophy consists of the following four key ideas. When combined, they enable our goals of providing low interrupt latency and low-cost inter-component communication.

- *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, and other than synchronization code, is the only part of the system that must be run with interrupts disabled. By reducing the length of time that interrupts are disabled, we reduce the maximum interrupt latency seen.

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 its essential service.

- *Each component is implemented by a separate protection domain. The cost of transferring control from one protection domain to another should be small enough that there is no performance-related reason to co-locate components.*

Microkernel systems, such as Mach, have in the past tended towards coarse-grained user level servers, in part because the cost of transferring between protection domains was high. By keeping this cost low, we enable the factoring of the operating system, and application, into smaller components with small performance penalty.

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

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

A noteworthy example is the scheduler: Pebble does not handle scheduling decisions at all. The user-replacable scheduler is responsible for all scheduling and synchronization operations.

- *Transferring control 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, inter-protection domain calls, 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. The portal mechanism provides two important features: abstract communication facilities, which allow components to be isolated from their configuration, and per-connection code specialization, which enables the application of many otherwise unavailable optimizations. Portals are discussed in depth in Section 4.

3 Software vs. Hardware Protection

Suggesting the use of a hardware protection scheme, in the age of Java and software protection, is a controversial (perhaps heretical) proposal. However, each approach has its applications.

Component-based systems that use software protection schemes are typically written in a type-safe byte-coded language, such as Java [Gosling96] and the Limbo language of the Inferno system [Dorward97]. Components run in a single hardware protection domain, but the run-time environment implements (in effect) a software protection domain.

These systems are typically designed with the following three goals:

1. Provide an architecture-independent distribution format for code.
2. Ensure that resources (such as memory) are returned to the system when no longer needed.
3. Ensure that the component does not view or modify data to which it has not been granted access.

In the case of Java, these goals are satisfied by (1) the machine-independent Java byte-code, (2) the garbage collector provided by Java run-time environments, and (3) the run-time Java byte-code verifier.

Java byte-code offers a hardware-architecture-neutral distribution format for software components. However, such an architecture-neutral format could also be used for untrusted code. Most compiler front ends generate a machine-independent intermediate form, which is then compiled by a machine-specific back end. Such an intermediate form could be used as a distribution format for components written in any language, trusted or untrusted.

Software protection has problems too. Putting all software-protected components in the same address space makes it hard to pin down a buggy component that is not caught by the type system or the garbage collector.

Hardware protection schemes run each component in a separate hardware protection domain. As an example, a traditional operating system (such as Unix) could be thought of as a hardware-protected component-based system, where the components are programs, and protection is provided by the operating system working in concert with the hardware memory management unit.[1]

Typically, hardware schemes do not attempt to satisfy (1), since components are distributed in the machine language of the target hardware. (2) is satisfied by careful bookkeeping: the system keeps track of the resources assigned to each component (process), and when the component (process) terminates, the resources are returned to the system. (3) is implemented using hardware memory protection: each component is run in a separate address space. If a component attempts to view or modify data outside its address space, a hardware trap is taken and the component (process) is terminated.

By running multiple Java virtual machines on top of hardware protection one can separate the components in a way that makes it easier to identify buggy components.

Software schemes are the only option when hardware protection is unavailable, such as low-end processors without memory management units. (The designers of Inferno/Limbo chose a software protection scheme for precisely that reason.)

Hardware schemes are called for when component code is written in an unsafe language, such as C or C++. Although Java provides many facilities unavailable in C and C++, there are often good reasons for running code written in an unsafe language. For example, a component may include legacy code that would be difficult or costly to reimplement in a safe language, or may include hand-tuned assembler code that uses hardware specific features. One example of the latter is an implementation of a computation-intensive algorithm (such as an MPEG decoder) using special hardware instructions designed to support such tasks.

The garbage collection facility offered by systems such as Java can be seen as both a blessing and a curse. On one hand, garbage collection frees programmers (and software testers) from ensuring that all allocated resources are eventually freed. Storage leaks are notoriously hard to find, and automated garbage collection greatly simplifies the task of writing robust code. However, when building applications that have fixed latency requirements, garbage collectors can be more trouble than they are worth. Because the free memory pool may become empty at any time, any memory allocation could trigger a collection, which makes estimating the cost (in time) of a memory allocation a stochastic, rather than deterministic, process. In a real-time embedded system, the uncertainty introduced by the presence of a garbage collector may not be worth the benefit it offers. This is the reason why some systems, such as Inferno, invested much effort to ensure that garbage collection does not delay critical functions.

One final historical argument for choosing software protection over hardware protection is that the cost of transferring between components under a hardware protection scheme is several orders of magnitude higher than it is under a software protection scheme. In this paper we show that the techniques used by Pebble bring the cost of cross-component communication under a hardware scheme to within an order of magnitude of the

1. To continue the analogy, such components can be composed using Unix pipes.

cost of a function call, which we feel makes this point moot.

4 Pebble Technology

In this section we discuss the technology used to implement Pebble.

4.1 Protection Domains and Portals

Each component runs in its own *protection domain (PD)* which consists of a set of memory pages, represented by a page table, a *portal table*, which holds the addresses of the *portal code* for the portals to which the component has access. A protection domain may share both memory pages and portals with other protection domains, as discussed below.

A parent protection domain may share its portal table with its child. In this case, any changes to the portal table will be reflected in both parent and child. Alternately, a parent protection domain may create a child domain with a copy of the parent's portal table at the time when the child was created. Note that both copying and sharing portal tables are efficient, since portal tables contains pointers to the actual portal code. No copying of portal code is needed in either case.

A thread belonging to protection domain A can invoke a service of protection domain B only if A has successfully opened a portal to B.[1] Protection domain B, which exports the service, controls which protection domains may open portals to B, and hence which components can invoke B's service. Protection domain B may delegate the execution of its access-control policy to a third party, such as a directory server or a name-space server.

To transfer control to B, A's thread executes a trap instruction, which transfers control to the nucleus. The nucleus determines which portal A wishes to invoke, looks up the address of the associated portal code, and transfers control to the portal code. The portal code is responsible for saving registers, copying arguments, changing stacks, and mapping pages shared between the domains. The portal code then transfers control to component B. Figure 1 shows an example of portal transfer.

When a thread passes through a portal, no scheduling decision is made; the thread continues to run, with the same priority, in the invoked protection domain.

1. "Protection domain A" here is shorthand for "the protection domain in which component A is running."

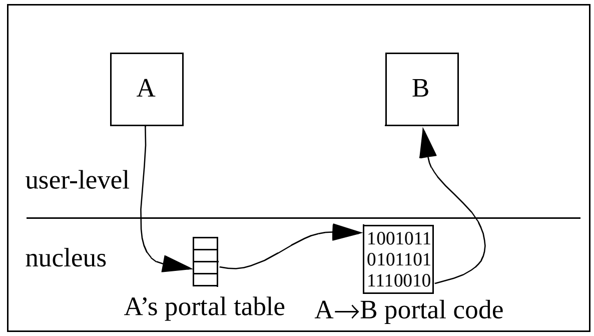

Figure 1. Portal Transfer. Protection domain A invokes protection domain B via a portal transfer. Protection domain A transfers indirectly through its portal table to the portal code specific to this communication path. The portal code transfers control to protection domain B.

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 portal invocation, 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.

Portal code may never block and may not contain loops. This is essential to ensure that the portal can be traversed in a small, finite amount of time. If the portal has to block (e.g. the invoked domain's stacks queue is empty), then the portal code transfers control to the scheduler, inside which the calling thread is waiting for the resource.

An important point that has not yet been mentioned is that specialized portal code is generated, on-the-fly, when a portal is opened. This allows portal code to take advantage of the semantics and trust relationships of the portal. For example, if the caller trusts the callee, the caller may allow the callee to use the caller's stack, rather than allocate a new one. If this level of trust does not exist, the caller can require that the callee allocate a new stack. Although sharing stacks decreases the level of isolation between the caller and callee, it can improve performance.

4.2 Server Components

As part of the Pebble philosophy, system services are provided by *server components*, which run in user mode inside separate protection domains. Unlike applications, server components 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 can not modify its portal table directly).

There are many advantages for 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. Clearly, 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.

4.3 Scheduling and Synchronization

Pebble's scheduler implements all actions that may change the calling thread's state (e.g. $run \rightarrow blocked$ or $blocked \rightarrow ready$). Threads cannot block anywhere except inside the scheduler. In particular, Pebble's synchronization primitives are managed entirely by the user-level scheduler. When a thread running in a protection domain creates a semaphore, two portals that invoke the scheduler (for *P* and *V* operations) are added to the protection domain's portal table. The thread invokes *P* in order to acquire the semaphore. If the *P* succeeds, the scheduler grants the calling protection domain the semaphore and returns. If the semaphore is held by another protection domain, the *P* fails, the scheduler marks the thread as blocked, and then 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.

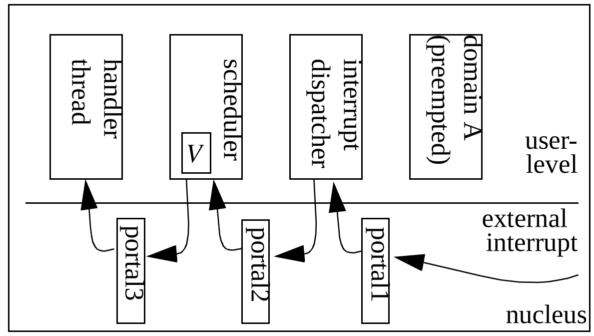

Figure 2. Interrupt Handling. An interrupt causes a portal call to the interrupt dispatcher, which calls the scheduler to performs a *V* operation on the device's semaphore. The scheduler wakes up the handler thread that waits on this semaphore. See text for explanation of different portal activities.

The scheduler runs with interrupts disabled, in order to simplify its implementation. Work on the use of lock-free data structures has shown that, with appropriate hardware support, it is possible to implement the data structures used by Pebble's scheduling and synchronization component without locking [Herlihy91]. Such an implementation would allow the scheduler to run with interrupts enabled which would reduce interrupt latency even further. (We have not yet implemented the scheduler using such data structures, although we plan to investigate such an implementation in the future.)

4.4 Device Drivers and Interrupt Handling

Figure 2 depicts the interrupt handling in Pebble. Each hardware device in the system is associated with a semaphore that used to communicate between the interrupt dispatcher component and the device driver component for the specific device. Typically, the device driver will have left a thread blocked on that semaphore.

The portal table of each protection domain contains entries for the machine's hardware interrupts. When an interrupt occurs, *portal1* saves the context of the currently running thread, including the contents of the entire register set. *Portal1* then switches the stack pointer to the interrupt stack and calls the interrupt dispatcher, which determines which device generated the interrupt. The interrupt dispatcher calls the scheduler to performs a *V* operation on the device's semaphore via *portal2*. This portal saves only a few registers and allows the scheduler to share the same stack as the interrupt dispatcher. The *V* operation unblocks the handler thread. If the handler thread has a higher priority than the thread which was running at the time when the interrupt was received, the scheduler calls *portal3* with the identity of the handler thread. *Portal3* restores the context of the handler thread, including registers and stack, and the interrupt is handled immediately. Otherwise, the handler thread is added to the ready queue and the scheduler selects to resume the thread which was running previously by calling *portal3* with the identity of this thread. Note that *portal3* performs the actual context switch. The scheduler just supplies the identity of the next thread to run.

Note that Pebble does not rely on hardware interrupt priorities in order to schedule interrupt handler threads. The interrupt dispatcher is called promptly for all interrupts, and the Pebble scheduler decides whether to run the associated handler thread. Pebble unifies interrupt priority with thread priority, and handles both in the scheduler

4.5 Writing Portal Code

Portal definitions are written using a simple interface definition language. The portal manager uses this definition to dynamically generate specialized code when a portal is created. The interface definition specifies which registers to save, whether to share a stack with the receiving domain, and how to process each argument.

Simple arguments (e.g., integers) are not processed at all; more complex argument types may require more work. For example, an argument may specify the address of a memory window that should be mapped into the receiver's address space, or a capability (see section 4.7) that must be transformed before being transferred. On one hand, in order to be efficient, such transformation code may need to have access to private data structures of a trusted server (e.g., the virtual memory system or the capability system); on the other hand, the trusted servers should be allowed to keep their internal data representations private.

The solution we plan to implement is to allow trusted services to register argument transformation code templates with the portal manager. When the portal manager instantiates a portal that uses such an argument, the code template is used when generating 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.

4.6 Short-Circuit Portals

In some cases the amount of work done 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 obtain the time of day. 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.

4.7 Capabilities

Pebble's design includes support for *capabilities*, abstract tokens that represent access rights. The capability manager, a trusted user-level server, keeps track of the capabilities available to each protection domain. Additionally, it registers a capability argument type and associated transformation code with the portal manager. When a capability is passed through a portal, the portal code adds the capability to the receiving protection domain's capability list, and transforms the sending protection domain's external representation of the capability to that of the receiving domain. We plan to implement the standard capability operations (e.g., revocation, use-once, non-transferability, reduction in strength, etc.).

5 Portal Example

Portals can be used to model not only code, but also data; a set of portals can be used to represent an open file descriptor. In Pebble, an *open* call creates three portals in the invoking protection domain, one each for *read*, *write*, and *seek* on the corresponding file. A *read* call transfers directly to the appropriate routine, so no run-time demultiplexing is needed to determine the type of the underlying object; the appropriate code (for a disk file, socket, etc.) will be invoked. Additionally, a pointer to its control block can be embedded in the portal code and passed directly to the service routine, so no run-time validation of the file pointer needs to be done. Because the portal code can not be modified by the client, the server can trust that the control block pointer passed to it is valid, and can access the particular file immediately. There is no need for a separate file descriptor table; the data normally associated with the tables is found in the dynamically generated portal code.

6 Performance

We measured the operation of Pebble for several micro-benchmarks on three different test hardware platforms, named LOW, MID and HIGH, which represent low-, medium- and high-end embedded system configurations, as described in Table 1. All three platforms included MIPS processors from QED (RM5230 and RM7000) and IDT (R5000). All motherboards were manufactured by Algorithmics, a developer of systems for embedded applications.

The LOW platform is representative of low-cost hand-held devices, which have a single-level cache hierarchy and smaller memory. The MID platform is representative of more powerful appliances, such as a set-top box, which contain a more powerful processor, two-level cache hierarchy and larger memory. The HIGH platform is representative of high-end systems, which contain top-of-the-line processors with larger caches (Note that it is inevitable that over time the HIGH platform will come to be considered MID and the MID platform LOW.).

The L1 cache in all targets can be accessed in a single machine cycle, and does not cause any pipeline delay. Access to higher levels of the memory hierarchy causes a delay, which is depicted in Table 1.

6.1 Basic Operations

Table 2 depicts the time to perform several primitive operations on the three test platforms. We present the results in machine cycles and not in absolute time, since we believe that these measurements will remain the same (in machine cycles) for faster processors with similar memory hierarchy. The reported times are the average of 10,000 repetitions.

All of the operations reported in Table 2 are simple communication or synchronization building blocks. Their performance suggests that higher-level operations will be efficient as well.

The reported operations are:

- **s-c portal:** the time of the `get_asid()` short-circuit portal, which returns the identity of the calling domain. This is the equivalent of the UNIX null system call. A short-circuit portal performs the action and returns to the caller immediately without a context switch.

	LOW	**MID**	**HIGH**
board	P4032	P5064	P5064
processor	RM5230	R5000	RM7000
processor speed (MHz)	133 Mhz	166 MHz	200 MHz
pipeline	single	single	dual
L1 cache	2-way 16 KB I + 16 KB D	2-way 32 KB I + 32 KB D	4-way 16 KB I + 16 KB D
L2 cache	—	off-chip direct-map 1 MB	on-chip 4-way 256 KB
L2 access (cycles)	—	10	3
L3 cache	—	—	off-chip direct map 2 MB
L3 access (cycles)	—	—	17
main memory	16 MB	64 MB	64 MB
memory access (cycles)	40	26	41

Table 1. Test hardware. We ran our tests on three platforms that represent three points in the embedded system hardware spectrum. The platforms share a common CPU architecture, and vary in cache size and architecture and processor speed.

- **ipc**:[1] the time for one leg of an IPC between two domains, which is implemented by a portal traversal. The portal passes four integer parameters in the machine registers. No parameter manipulation is performed by the portal code. The portal allocates a new stack in the target domain and frees the stack on return. The reported time in Table 2 is one leg time (half of the total time). Additional measurements

1. The acronym *ipc* traditionally refers to an *inter-process communication*. By serendipity, we can also use *ipc* to represent an *inter-protection domain call*; in Pebble, the two are equivalent.

(reported in [Gabber99]) show that the per-leg time is constant for a chain of IPCs through a sequence of domains.

- **yield:** measures the thread yield operation, in which the current thread calls the scheduler and requests to run the next thread with a higher or same priority. We present four measurements, for one, two, four and eight threads (**yield1**, **yield2**, **yield4**, and **yield8**, respectively). There is one active thread in each domain. The reported time in the table is a single context switch time (total time / total number of yields by all threads).
- **sem:** measures the time to pass a token around a ring of n threads, each running in a separate domain. Each thread shares one semaphore with its left neighbor and one semaphore with its right neighbour. The thread waits on its left neighbor. Once this semaphore is released, the thread releases its right semaphore and repeats the process. We present four measurements, for one, two, four and eight threads (**sem1**, **sem2**, **sem4**, and **sem8**, respectively). There is one active thread in each domain. The reported time is the time to pass the token once, which is the time for a single pair of semaphore acquire/release.

The results reported in Table 2 indicate that the performance degrades with the number of active threads, which is expected due to more frequent cache misses. Performance degrades most with LOW platform, since it has no L2 cache, and least with HIGH platform, which has a large L3 cache that is large enough to hold the entire working set of the test.

Note that the HIGH platform is not significantly faster than the LOW and MID platforms for **s-c portal**, **ipc** and **sem1** tests, although it has a dual-issue pipeline and much larger caches. This is because these tests do not cause much L1 cache misses, and the portal code is dominated by sequences of load and store instructions that cannot be executed in parallel.

6.2 Interrupt Latency

When an interrupt is generated, there are two factors that control the delay before the interrupt is handled. First, there can be a delay before the system is notified that an interrupt has been received, if the processor is running with interrupts disabled. Second, there may be a delay between the time the interrupt is delivered and the time the device is serviced. The sum of these two delay components is the interrupt latency.

operation	LOW	MID	HIGH
s-c portal	44	45	41
ipc	118	117	119
yield1	667	423	348
yield2	661	425	346
yield4	1593	549	346
yield8	1628	549	452
sem1	543	549	524
sem2	775	781	720
sem4	1872	942	720
sem8	1878	1198	857

Table 2. Basic operations time (cycles). These measurements give an estimate of the performance of Pebble on basic communication operations.

The first delay component, L, is bounded (below) by the length of path through the system where interrupts are disabled.[1] In particular, L is determined by the portal code and by the scheduler, which are the only frequently-used portions of the system that run with interrupts disabled.

The value of L is the time reported in Table 2 for **s-c portal**, **ipc** and **yield1**-**yield8**. It is half the time reported for **sem1**-**sem8**, since these times are for a pair of semaphore operations.

The second delay component, C, is bounded (below) by the minimum amount of time required to deliver the interrupt to the interrupt handler. Thus the interrupt latency will range from $[C, C+L]$, provided that interrupt handling does not generate many additional cache misses, such as in the MID and HIGH platforms.

Table 3 shows the interrupt response time of Pebble when the system is performing different background tasks. We present the median and 99th percentile response time of 10,000 repetitions.

The interrupt latency is measured by computing the difference between the time that an interrupt was generated

1. Length, in this context, refers to the amount of time it takes to process the instruction sequence. Intuitively, it can be expected to be roughly proportional to the length, in instructions, of the code path.

and the time that a user thread that waited for this interrupt actually woke up. To accomplish this, we have the measurement thread sleep for some (randomly chosen) duration, and compare the time it is woken up with the time it expected to be woken up. The difference between the two is the interrupt latency.

This test is very precise; each of the test platforms include a high-resolution timer that is incremented every second processor cycle and the ability to schedule timer interrupts with the same granularity. The code fragment Figure 3 shows the inner loop of the measurement thread.

```
for (i = 0; i < N; i++) {
  delay = rand(MIN, MAX);
  start = hrtime();
  hrsleep(delay);
  latency = hrtime() - start - delay;
}
```

Figure 3. Measurement Thread. The code of the inner loop of the interrupt latency measurement thread.

The `rand()` function generates a uniformly distributed random number in the range [MIN, MAX]. The `hrtime()` function returns the current high-resolution time. The `hrsleep()` routine waits until the specified time.

In order to estimate interrupt latencies under various loads, we run the measurement thread concurrently with a background task that repeatedly performs a specific operation. Different operations exercise different interrupt-disabled paths of the system, and hence will have different interrupt latency characteristics. The background threads we tested were:

- **idle:** a background thread that spins in a tight loop on a particular variable in user mode. The idle task can be preempted at any time. The value reported for this case is an estimate of *C*, the lower bound of the interrupt latency.
- **s-c portal**: a background task that calls the `get_asid()` routine repetitively. `Get_asid()` is implemented by a short-circuit portal. The background thread is identical to the **s-c portal** program from Table 2. Interrupts are disabled while executing this portal.
- **ipc**: a background thread that repeatedly performs an IPC to the protection domain in which the measurement thread is running. The portal code associated with this portal is minimal, just transferring control, and call itself returns immediately. The background thread is identical to the **ipc** program from Table 2. Interrupts are disabled during each leg of the IPC (call and return), and are enabled when executing in the caller and called domains.
- **yield**: a background thread that repeatedly calls the scheduler to yield control. This thread is identical to the **yield1** program from Table 2. As there is one active thread in the system, the scheduler just returns to the calling thread. Interrupts are disabled during the portal call to the scheduler, inside the scheduler, and during the portal return to the thread.
- **sem**: a pair of background threads, which pass a token back and forth. Each thread runs in separate protection domain. This test is identical to the **sem2** program from Table 2. Interrupts are disabled during the semaphore operations.

background		LOW	MID	HIGH
idle	median	1200	1294	1224
	99th %	1298	1296	1228
	max.	2164	1322	1322
s-c portal	median	1170	1176	1240
	99th %	1240	1200	1262
	max.	2026	1202	1266
ipc	median	1152	1224	1274
	99th %	1240	1290	1340
	max.	2088	1300	1342
yield	median	1210	1346	1404
	99th %	1474	1580	1584
	max.	2282	1584	1588
sem	median	2302	1492	1400
	99th %	3024	1712	1596
	max.	3996	1722	1604
est. interrupt latency lower bound (*C*)		9.0µs	7.8µs	6.1µs

Table 3. Interrupt latency (cycles). The time, in cycles, between the expected delivery of an interrupt and when it is received.

Table 3 indicates that the interrupt latency is bounded by the sum of a platform-specific constant plus a time which is proportional to the longest interrupt-disabled path in the background task. The platform-specific constant is the minimal interrupt response time *(C)*, which is the median value of the **idle** test. The measurements in Table 2 are the upper bound of the duration of the corresponding interrupt-disabled paths.

Note that the median and the 99th percentile of the **idle** test on all platforms are very close. This indicates that the interrupts are served immediately.

The maximal interrupt latency on the MID and HIGH platforms is very close to the 99th percentile on these platforms, which indicates that the system performance is highly predictable. However, the maximal interrupt latency on the LOW platform is up to 60% higher than than the 99% percentile latency. This is the result of the small cache size of LOW, which result in excessive cache misses due to infrequent background events, such as the timer interrupt.

In the case of the LOW and MID platforms, the **s-c portal** and **ipc** tests had a slightly lower median interrupt latency than the **idle** test, but they differ by less than 5%. This may be caused by a better cache hit rate.

We see that the interrupt latencies for the MID and HIGH systems are quite close, and much lower than those for LOW. Although the cache architectures of the two differ, both MID and HIGH appear to have more effective caches than LOW. The L1 cache of MID is twice the size of the L1 cache of LOW, and although the L1 cache of HIGH is the same size as that of LOW, it is 4-way set associative. In addition, the L2 cache of HIGH can be accessed in only three cycles. Thus an L1 miss on HIGH is not as painful as it would be on MID or LOW.

6.3 Summary

In this section we have shown microbenchmark results for Pebble that indicate that the cost of hardware protection of both components and system services is very low (as summarized in Table 2) and that interrupt latency on Pebble is quite low, with a lower bound of 1200-1300 cycles (6.1µs to 9.0µs) depending on the target architecture.

7 Related Work

Traditional microkernels, such as Mach, Chorus, and Windows NT leave much more of the operating system's code running in privileged mode than does Pebble. In fact, the trend has been to run more of the system in privileged mode as time goes by—the current release of NT includes the window system in the privileged mode kernel.

By moving services into the privileged mode kernel to reduce communication overhead, operating system designers are, in essence, giving up on the microkernel approach. Recent work has focused on finding and fixing the performance bottlenecks of microkernel approach, which has required rethinking its basic architecture.

Liedtke, in his work on L4, has espoused the philosophy of a minimal privileged mode kernel that includes only support for IPC and key VM primitives [Liedtke97]. Pebble goes one step further than L4, removing VM as well (except for TLB fault handling, which is done in software on MIPS).

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

Pebble was inspired by the SPACE project [Probert91], which was in turn inspired by the Clouds project [Dasgupta92]. Many of the concepts and much of the terminology of the project come from these systems; e.g., SPACE provided us with the idea of cross-domain communication as a generalization of interrupt handling.

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.

8 Status

The Pebble nucleus and a small set of services (scheduler, portal manager, interrupt dispatcher, and minimal

VM) and devices (console, clock, and ethernet driver) currently runs on the MIPS processor. Work on implementing networking support (TCP/IP), file system support, and a port to the x86 is underway.

9 Summary

The Pebble architecture provides for an efficient operating system that is easy to modify and debug. Pebble provides hardware protection for running components written in any language. Components communicate via portals and run in user mode with interrupts enabled. Through the use of portals, which are specialized to the specific interrupt or communication they are to handle, Pebble is able to "compile out" run-time decisions and lead to better performance than typical operating system implementations. In addition, a portal can be configured to save and restore only a subset of the machine state, depending on the calling conventions of, and the level of trust between, the client and server. Additionally, the low interrupt latency provided by the Pebble architecture makes it well-suited for embedded applications.

10 References

[Bershad89] B. Bershad et al., "Lightweight Remote Procedure Call," *Proc. 12th SOSP*, pp. 102–113 (December 1989).

[Dasgupta92] P. Dasgupta, R. LeBlanc, M. Ahamad, U. Ramachandran, "The Clouds Distributed Operating System," *IEEE Computer 24*, 11 (November 1992).

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

[Gabber99] E. Gabber, J. Bruno, J. Brustoloni, A. Silberschatz, C. Small, "The Pebble component-based operating system," *Proc. 1999 USENIX Technical Conference*, Monterey, CA (June 1999).

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

[Herlihy91] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems 13*, 1 (January 1991)).

[Kaashoek97] M. F. Kaashoek et al., "Application Performance and Flexibility on Exokernel Systems," *Proc. 16th SOSP*, pp. 52–65 (October 1987).

[Liedtke97] J. Liedtke et al., "Achieved IPC Performance," *Proc. 6th HotOS*, pp. 28–31 (May 1998).

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

[Mendelsohn97] N. Mendelsohn, "Operating Systems for Component Software Environments," *Proc. 6th HotOS*, pp. 49–54 (May 1978).

[Probert91] D. Probert, J. Bruno, M. Karaorman, "SPACE: A New Approach to Operating System Abstractions," *Proc. IWOOOS*, pp. 133–137 (October 1991). Also available from ftp.cs.ucsb.edu/pub/papers/space/iwooos91.ps.gz

[Pu95] C. Pu et al, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System," *Proc. 15th SOSP*, pp. 314–324 (December 1995).