Critique of

“

‘The Pebble Component-Based Operating System’

by

Eran Gabber, Christopher Small, John Bruno, Jos Brustoloni and Avi Silberschatz

”

by anonymous

12/11/1999

This paper describes the architecture and implementation of a new component-based operating system named Pebble. It is presented as an evolutionary work which combines good features drawn from a number of other systems. It is intended as both a general and an embedded operating system, and this critique will look at the success it has in achieving these goals.

The paper is well-written and well-researched. The presentation of figures and results is good. The Pebble system itself is well-designed and consistent, and it is based on appropriate philosophical foundations, which capture important concepts behind the design.

Pebble

The Pebble operating system consists of a kernel-mode nucleus and a collection of user-level servers, each with their own protection domain. The nucleus handles only control transfer — everything else is left to user-levels servers including memory management, scheduling and even interrupt handling. Threads are not tied to a particular domain (as they are in L4), but rather move between domains by passing through portals. A portal is most similar to a call gate in the Intel x86 architecture — it is a parametrised protected domain control transfer point — an embellished entry point. It shares many features with similarly (or identically) named concepts, such as portals in Mach or doors in Spring.

A portal connects one protection domain to exactly one other. Portals are the means of IPC. In the spirit of µ-kernels such as L4, interrupts and exceptions are transformed into IPCs to device drivers or exception handlers. This includes the timer interrupt to the scheduler and even TLB miss exceptions. IPC is thus designed to be very fast.

Portals are stored in a protected portal table in each protection domain — a thread cannot modify its own portal table, this must be done by a controlling server. A portal is invoked through a system call. A portal has parameters (usually in registers) which are marshalled by trusted portal code — this is written in a safe interface-definition language and compiled by the portal server when the portal is created. This special code references only the portal it applies to, and is executed when that portal is invoked to transfer control and data (possibly including memory windows, the equivalent of page mappings) to the destination protection domain.

The following sections detail a number of observations I made in reading the paper.

Praise

Since the paper is for the most part good, this section is reserved for ideas singled out as clever or noteworthy. The paper and system deserve overall deserve general praise.

The related work section is an excellent summary of the relevant operating systems from which Pebble draws its concepts. Referencing Dijkstra's THEOS is a nice touch. However the CISC mention at the end seems out of place, although it is certainly relevant to the implementation an understanding of Pebble portals. However a number of opportunities for references in later sections to other works seem to have been missed, which is not completely excused by the quality of this section.

Pebble is based on a good software engineering philosophy and leads to a good operating system design. In particular, the 'small is good', 'replaceable component-based OS' and 'no user/kernel co-location' principles are excellent starting points.

The flexibility of the memory windows is also commendable — this paper makes windows sound interesting — especially the tear-away read/write properties for fast I/O data propagation, whether between servers or to eventual clients.

The protection mechanism of portal tables, under a strict process hierarchy, seems to hint at the containment possibilities of the system, although this term is not mentioned explicitly. In particular the power and transparency of server interposition is impressive — for the system administrator at any rate — and allows all manner of extensions and expansions without changes to of awareness of user-code. The possibility of easy compatibility with a number of different operating systems (just as µ-kernels can provide) is also promising. (However the example of such interposition given in the Appendix seems out-of-place — it violates the paper's anti-multiplexing mantra :)

Another consideration not explicitly named is that of real-time use. Great care has been taken to ensure reasonable worst-case-bounds in code executing when interrupts are disabled, specifically isolation from the effects of user code and the absence of loops in the nucleus, but there is no measurement of these bounds.

The interface definition language is another nice concept. I initially thought the string-representation would hurt performance, but the measurements in section 7.5 show otherwise. The idea of a safe language for dynamic kernel code is not new (e.g. grafting), but using a small and concise language only for interface glue is a worthy notion. The way that nucleus variables can be inserted is also impressive.

The consolidation of interrupt and thread priorities shows a good implementation method.

Overall the performance benchmarks performed were well measured and good methodology was used. In particular the use of hbench:OS over lmbench is commendable. The actual performance derived is also good, for the small measurements made. See below for more comment on the performance analysis.

Problems

This section contains issues I consider problems with either the operating system.

Foremost of these is terrible use of the cache! Code is generated for each portal, and the portal table is duplicated — which contains the interrupt-receiving portals including the TLB refill handler — for every protection domain.

This means that if the TLB refill handler is cached from one protection domain and the thread moves to another, that copy of the handler will not be used in the new domain, and the cache may take another completely unnecessary hit.

This cache misuse is made worse by the absence of shared libraries. Such dynamic code generation was a good idea when RAM speed keeps pace with processor speed, but over the last decade their performance gap has increased. Even the once-revered 'compiled sprite' used in graphics engines is no more.

I very much doubt that the small saving in demultiplexing costs is offset by the greatly increased use of the cache.

The operating system uses a safe language compiled by a trusted server at runtime to guarantee protection in portal use. This is to some extent admitted in section 4.3 However the 'Philosophy and Architecture' section at the start claims that its security is derived primarily from hardware-based memory protection, unlike other systems which depend on safe code. Pebble too depends on safe code (although significantly less of it)

The capability to perform dynamic updates is claimed but not explained — if it is possible to even replace the virtual memory system or the scheduler without restarting the system, then this is a great advantage should be made clear and explained.

The system as presented has no demand-paged virtual memory. The kind of memory management which it does have is not clear. This is admitted in section 5.2, and in section 5.5 concerning the footprint of the OS there is no mention of any memory management component.

Portals are used everywhere, but they are reasonably expensive entities, as they each entail a significant amount of code, and therefore space. Semaphores in particular would be quite expensive in terms of memory. Also the use of four portals for each open file is perhaps unwise, as it limits the extensibility of that system (what if a flush command is required — a separate interface must be created or existing programs recompiled). However as an example of the parameter checking it is useful.

Among the provided results, there is the assertion that short-circuit portal execution costs are equivalent to that of a null system call. They are certainly a useful measure, but I dispute that it is equivalent to a null system call — they do not perform the actions that the vast majority of system calls require — one may as well implement the null system call in a library and measure the cost of that. Also the result tables with the column 'window+fault' are misleading, the title should read 'window+access', because clearly a fault does not occur here (or it takes only 2-3 cycles!) when there is only one protection domain.

Finally there are problems with the benchmarks performed. They were all done on small amounts of user code — where the caching problems caused by code duplication are less apparent. Indeed, there are a number of comments in the paper which tie worsening times to the caches reaching capacity. I suspect that if the (admitted) microbenchmarks were performed with a greater number of processes, or more user

code, or if macrobenchmarks were performed, then less impressive times would result. A test with 128 instead of 8 simultaneous processes would likely bring cache inefficiencies into sharper focus.

Questions

This section contains questions which my reading of the paper raised, and which were not eventually answered.

Are the system structures actually inherent to Pebble if the nucleus knows nothing about them — that is, is it possible to reuse the nucleus for other operating system or µ-kernel designs?

The interface language is not explained well enough. For example, the difference between ‘c’ = Constant and ‘i’ = Word is not explained. Are these to allow IPC to systems of different endianness, or is a constant replaced by some nucleus variable. If so, which variable, and what are available?

In section 4 on page 6 an example of a distributed shared memory component is made, sitting between the standard virtual memory and a network file system. But the paper states that the DSM component should provide a VM interface, whereas surely it should provide a file system interface if the VM system is to think it a file system.

The description of stack queues in section 5.4 raises the question of why semaphores are used for (whole-module) concurrency control in drivers when it can be done automatically by limiting the number of stack queues.

Section 5.4 also raises the question of why every thread needs its own interrupt stack if the interrupt dispatcher runs with interrupts turned off, and it is the only component to use this stack.

Memory windows sound interesting, but by the end of the paper there is no deeper explanation of them. In section 6 page 9 this brings on the question of how memory window arguments can be passed on automatically through an interposing protection domain. How are they specified — where do they end up in the invoked component’s address space — or is there a separate global shared space? It is understood that the protection domain would also acquire the window, but how do the address translations work unchanged?

On a similar topic, the ease with which different protection domains can share stacks is also not explained.

Conclusion

Pebble is certainly a promising operating system. However this critique has hopefully shown that it is not appropriate for all uses. It is probably most suited for embedded applications (which is not surprising considering the source of the paper), but falls short of the requirements of a general-purpose operating system.

The work is all slanted towards embedded uses — the small number of processes in the tests, the concentration on real-time-like concerns, the (current) absence of demand-paged virtual memory all indicate this.

So this paper succeeds in describing an interesting and well-designed operating system for use in small systems. However its cache failings would almost certainly cause significant problems in larger (desktop, server) systems.