The Portable Common Runtime Approach to Interoperability

Mark Weiser, Alan Demers and Carl Hauser

Xerox PARC
3333 Coyote Hill Road
Palo Alto, California 94304*

Abstract: Operating system abstractions do not always reach high enough for direct use by a language or applications designer. The gap is filled by language-specific runtime environments, which become more complex for richer languages (CommonLisp needs more than C++, which needs more than C). But language-specific environments inhibit integrated multi-lingual programming, and also make porting hard (for instance, because of operating system dependencies). To help solve these problems, we have built the Portable Common Runtime (PCR), a language-independent and operating-system-independent base for modern languages. PCR offers four interrelated facilities: storage management (including universal garbage collection), symbol binding (including static and dynamic linking and loading), threads (lightweight processes), and low-level I/O (including network sockets). PCR is "common" because these facilities simultaneously support programs in several languages. PCR supports C, Cedar, Scheme, and CommonLisp intercalling and runs pre-existing C and CommonLisp (Kyoto) binaries. PCR is "portable" because it uses only a small set of operating system features. The PCR source code is available for use by other researchers and developers.

**email: weiser.pa@Xerox.com*

This appeared in the Conference Record of the 13th Annual ACM Symposium on Operating Systems Principles, December 1989.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Copyright 1989 ACM.

1. Introduction

1.1 The Problem - interlanguage interoperability

Although there are many facets to interoperability, one remains largely unassailed: closely coupled interoperation between programs written in different languages. By closely coupled we mean that an application as real-time or sophisticated as a device driver or a database management system might have different parts written in different languages. The parts could share data structures, memory, and threads of control. Interoperation without giving one language a primary role is to be preferred: the choice of a language should be determined by the semantic model needed, not by the degree of support from the operating environment.

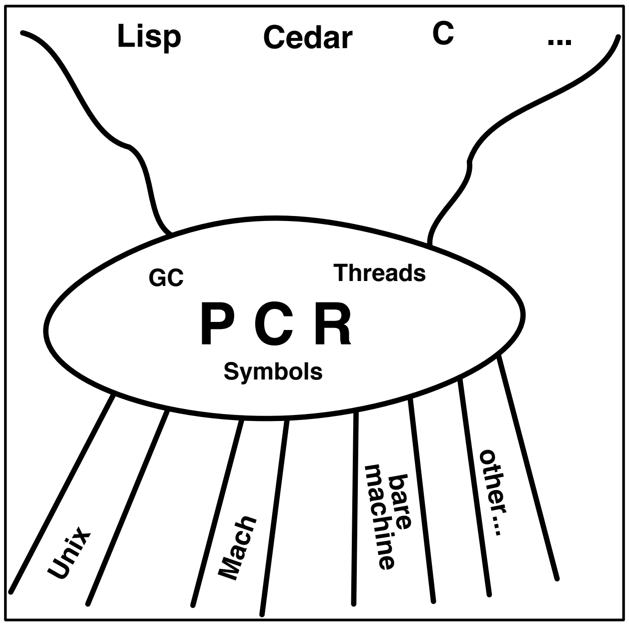
The principal prerequisites to closely coupled interoperation are the ability to share an address space, the ability to bind symbolic names between programs written in different languages, the ability to share I/O, and the ability to share data representations between programs written in different languages. In addition to these problems of interlanguage interoperation, intralanguage interoperation in the form of lightweight concurrent threads is an important concept of modern programming languages and this feature must be made interoperable as well, so that all programs can properly respect each others' critical sections.

1.2 The Portable Common Runtime solution

The Portable Common Runtime addresses the shared address space, symbol binding, and lightweight threads requirements for interoperability. The problem of shared data representations is beyond the scope of PCR, although certainly important for achieving much of the benefit of interlanguage interoperability.

Ordinarily, the language implementor produces a language-specific runtime layer directly on top of the operating system. In this scenario, features such as garbage collection must either be added at the operating system level or the language-specific runtime level. If the latter approach is chosen the feature becomes very difficult or impossible to use interoperably. However, an approach relying on a new operating system is a barrier to interoperability with existing applications and the existing operating systems they rely upon.

The PCR approach is to produce a common runtime layer between the operating system and the language runtimes. The PCR abstractions complement typical operating system abstractions such as virtual memory, communications, and file system. PCR allows exploration of the kind of features future generations of operating systems will have in their kernels to support interoperability.



PCR differs from language-specific runtimes both in the sophistication of some of its features, and the paucity of others. Compared with the C standard library, for instance, it offers the new features of threads, garbage collection and dynamic loading, but does not offer string functions or sophisticated printing or input scanning. Our choice is to focus deliberately on those features that language implementations must share for closely coupled interoperability, while avoiding other features in a runtime library that are not so important to interoperation. We assume that features we do not implement can continue to be done on a language-dependent basis without seriously reducing interoperation. For example, the string functions of the C library are adequate for manipulating C strings in PCR, while another library could be used for another language's character strings. This is not true for garbage collection, say, or the process model for which sharing data or critical sections requires a common abstraction.

PCR fails to solve the whole problem of language and application interoperation in at least two ways. As mentioned above, PCR does not solve the interlanguage data representation problem. Other attacks on interoperation, such as remote procedure call and Presentation Manager [Apik and Diehl 1988], do impose a standard method of data exchange. Second, PCR says nothing about a user interface. Again, other approaches, like *Open Look* and *Motif*, address this [Hayes and Baran 1989].

For additional alternatives to our approach, see section 5 of this paper on *Related Work*.

2. PCR Design Principles

The PCR design was constrained by the following principles:

1. implement above existing operating systems.
2. support existing simple applications.
3. permit the use of existing compilers, libraries, and binaries.
4. let sophisticated applications be written.

Implementing above the operating system means, in the first place, avoiding changes to operating system kernels, and the second place, not duplicating operating system functions. Therefore, for instance, a PCR implementation on Mach [Accetta et al. 1986] maps PCR threads into Mach threads. However, implementing PCR well requires from its base operating system certain functions not available from every operating system: it requires the ability to share memory and open files between operating system processes; it requires the ability to protect pages of memory, and to catch and restart from protection failures; it requires a file system. These features are available more and more, and so we traded off loss of portability to older operating systems for much greater functionality.

Supporting existing simple applications means that, as we add potentially interfering new features, older programming styles can mostly remain intact. For instance, although we garbage collect storage allocated by C code, we do not require that C programmers replace their 'malloc' and 'free' calls. PCR simply ignores the 'free's, and invisibly collects 'malloc'ed space. Binary files that can be dynamically loaded in PCR can also be statically linked using the vanilla Unix 'ld' command. PCR is not perfect in this respect, as the details in following sections make clear, but it achieves a useful compromise between backward compatibility and new functionality.

By permitting the use of existing compilers, libraries, and binaries, we help to enforce on ourselves our rule of supporting existing simple applications. We co-exist with a machine's native stack and calling conventions, so compiler back-ends do not have to change, and we accept standard relocatable object file format, so precompiled code continues to work. For example, the complete SunView window system library operates unrecompiled, dynamically loaded and garbage collected in PCR. One thing that does not work is dynamically loading binaries from which relocation information has been removed.

By saying we want to permit sophisticated applications, we mean applications most naturally expressed using PCR-specific features. For instance, an application managing many concurrent activities will use the threads facilities. The language that has stretched our interfaces farthest has been CommonLisp, because it already has notions of dynamic loading and of garbage collection. We had to make sure we offered facilities on top of which

CommonLisp language implementors could work. For example, an implementation using tagged pointers must be able to co-exist with our collector. In general, our solutions were of two types: make the interfaces more general, and provide for upcalls [Clark 1985] when there was no other way.

3. Design and Implementation of PCR

3.1 Threads

3.1.1 Background

The PCR threads interface offers the usual semantics of monitors, monitor locks, condition variables, fork/join, aborting, etc. [Hoare 1974, Brinch-Hansen 1975]. As indicated above, we have worked to make the interface general enough to be used cooperatively by many different languages. PCR threads meet the runtime requirements of languages such as Cedar/Mesa [Swinehart et al. 1986], Modula-3 [Cardelli et al. 1988], CommonLisp [Steele 1984], and ARGUS [Liskov et al. 1987]; and can easily simulate other threads packages such as Cooper's C-Threads for Mach, Sun's lwp [Sun 1988a], Bershad's [Bershad et al. 1988], etc. The following overview highlights noteworthy features of our implementation.

Threads implementations fall into two categories: inside or outside the OS kernel. Implementations inside the kernel, such as Mach [Accetta et al. 1986] or V [Cheriton and Zwaenepoel 1983], have explicit knowledge of multiple threads per address space, and the OS scheduler treats such threads separately. Implementations outside the kernel generally use coroutines in a single heavyweight process. Coroutine implementations can be faster at thread switching, because they avoid any overhead associated with entering and leaving the kernel (similar to the speedup achieved by *{Synthesis}* [Pu et al. 1988], although via a different method). However, their reliance on only one heavyweight process introduces two serious problems: first, if that process ever blocks, *all* threads are blocked and second, there is no opportunity to use a multiprocessor to achieve true concurrent thread execution. Techniques for avoiding blocking--use of the Unix non-blocking I/O primitives, for example--can alleviate this problem, but they cannot entirely eliminate it, since some kinds of blocking (e.g. page faults) cannot be predicted or even detected outside the kernel.

3.1.2 Implementation

Our approach to implementing PCR threads in an operating system like Unix, which has no notion of a lightweight process, is to have a small number of heavyweight processes act as a pool of "virtual processors" ("VPs") to execute the many threads. All VPs share a common address space. Each VP is treated by the PCR scheduler exactly like a cpu in a shared-memory multiprocessor system. This implementation avoids the problems of coroutine threads implementations while not suffering the cost of a kernel entry for every thread switch.

In the normal case of a thread blocking predictably (e.g. by waiting on a monitor lock or condition) or being preempted at a timeslice, scheduling a new thread under this scheme is essentially a coroutine jump within a single VP. Non-blocking I/O and other techniques are used to make most instances of thread blocking predictable, and thus avoid most instances of VP blocking. Occasionally, however, a thread blocks unpredictably, say for a page fault or file system I/O. In that case the VP running the thread blocks; but the remaining VPs are still available for heavyweight process scheduling by the OS, and continue to run other threads. On a uniprocessor, assuming the number of available VPs exceeds the number of unpredictably blocked threads, the net effect is just to trade a heavyweight process switch (between VPs) for a lightweight switch (between threads in a single VP); some threads continue to make progress at all times. On a multiprocessor, ready threads execute concurrently (depending only on a reasonable base kernel implementation) with no change to the PCR implementation.

The PCR implementation relies on a relatively small number of underlying kernel features, chief of which is the ability to share memory among heavyweight processes. Since this feature exists in OS/2, the Unix SVID, Mach, SunOS, and many other modern operating systems, we anticipate no serious portability problem. Other OS features required by PCR are the ability for heavyweight processes to interrupt one another and to catch interrupts, and the ability to define a medium-grained interval timer (our scheduler wakes up ten times a second for time-slicing). Our implementation runs better if it can also write-protect pages (used for stack red-zoning and parallel garbage collection), catch and restart from protection violations, and remap pages to different addresses.

3.1.3 Debugging

Debugging of threads is currently a bit difficult, and we are working to improve it. At present, there are a few interactive commands by which one can stop all VPs, run on a single VP, freeze or thaw individual threads, or *examine* an individual thread. Examining works like this: before examining, a single-process debugger (say Unix dbx) is pointed at a distinguished VP, and a breakpoint is set at a well-known location. When the examine command is given for a thread, the thread is scheduled on that VP, and forced to execute through the breakpoint location. The specified VP hits the breakpoint with the desired thread's stack appearing as the VP process stack.

Independently, we are developing an extensible, multi-language debugger for PCR [Sturgis 1989]. This debugger, called Cirio, handles operations such as starting and stopping threads and setting breakpoints in a uniform, language-independent way. Each supported language registers with Cirio a collection of objects that embody language-specific features such as symbol table interpretation, data type representation and stack frame layout. A preliminary, cross-machine version of Cirio is working for Cedar and C programs.

3.2 I/O

The I/O interface currently provided by PCR is a nearly-exact emulation of the Unix I/O system calls. This is certainly the least portable aspect of the PCR design, and we plan eventually to replace it. However, developing the ultimate general-purpose, powerful and fully portable I/O interface will involve substantial research and effort; the current design was simple to produce (we copied it) and has enabled us to write PCR-based applications and validate some implementation techniques.

One limitation of Unix (and some other systems as well) is particularly troublesome when combined with the implementation of threads described above: the maximum number of open files that a single heavyweight process can hold is much less than the total number of open files supported by the system. In "normal" use of Unix, with each heavyweight process running a single application, the open file limit is large enough to be uninteresting. But we want to implement network servers and other large systems using PCR; it is important that the per-heavyweight-process resource limitations of Unix not translate into system-wide resource limitations for PCR.

To deal with this problem, our implementation on top of Unix uses additional heavyweight processes as "I/O processors"("IOPs"), essentially to serve as caretakers for file descriptors. It works as follows:

A file is opened by allocating a file descriptor slot in one of the IOPs and sending a message to that IOP asking it to open the file. While the file remains open, its descriptor remains in the IOP; the descriptor slots of the VPs are treated as an LRU cache of copied descriptors. To perform I/O on a descriptor, a thread first ensures that a copy of that descriptor exists in the VP's descriptor cache. If necessary, the least recently used descriptor in the cache is replaced by a copy of the desired descriptor, which is transferred from the corresponding IOP using Unix-domain IPC or using stream operations on a specially written pseudo-device driver. Currently, all VPs maintain identical file descriptor caches, though this constraint could be relaxed at the cost of some complexity in the implementation. The thread then attempts a non-blocking I/O operation on the descriptor. If the operation fails because it would block, the thread sends a message to the IOP requesting notification when the descriptor becomes ready. It then waits on a condition variable, allowing the VP to schedule a different thread without blocking. Eventually the descriptor becomes ready and the IOP notifies the waiting thread, which wakes up and retries the I/O operation. This scheme works well under the obvious condition that the working set of descriptors fits in the VP's descriptor cache.

User code sees none of this, of course. PCR imposes a layer of indirection in the file descriptors, and mimics all the Unix I/O system call layer (read, write, open, ...). It does the same for the SunOS socket-oriented calls and the System V stream-oriented calls.

This I/O design enables us to support applications requiring more open files than allowed in a single Unix heavyweight process, at the cost of occasionally having to fault copies of descriptors into the VP descriptor caches.

3.3 Storage Management

3.3.1 Background

Storage management for many modern languages requires garbage collection. If programs are to make the most of a shared address space, it must be possible for them to share allocated data structures. This implies that storage allocation and garbage collection must be part of the common runtime rather than the individual language runtimes. An additional benefit of including storage management with garbage collection in the common runtime is that programmers in languages like C, which do not require garbage collection in their runtime, benefit from its inclusion.

In order to work for languages that cannot guarantee pointer locations, the Portable Common Runtime uses a conservative collection scheme as implemented by Boehm [Boehm and Weiser 1988]. Two different storage allocation systems have been implemented for PCR. The first is a direct adaptation of Boehm's Russell collector, with additions for typed objects and finalization. The second is a new implementation that is real-time, parallel, generational but noncopying, and handles pointers to the interior of objects. Because of its unique features, this second implementation is described in more detail in a separate paper [Demers et al. 1989]. Here we focus on the highlights common to both collectors, and in particular on the mechanisms common to both for finalizing objects in a conservative world, and for pointer-finding upcalls.

Garbage collectors can be either reference counting or mark-and-sweep. Reference counting collectors impose overhead on each pointer manipulation; mark-and-sweep collectors defer the overhead to garbage collection time. Conservative collectors [Bartlett 1988] are a new type of mark-and-sweep collector. The advantage of conservative collectors is that they require no support from language implementations. Even without exact knowledge of which words in memory are pointers a conservative collector will never identify a reachable object as garbage; however, it may err in the other direction.

As Bartlett and Boehm have shown, conservative or partially conservative collectors work for many languages. For PCR they have been extended in two ways: finalization and pointer finding upcalls.

3.3.2 Finalization

Finalization is the method by which an application can request that it get a chance to look at an object just before it is freed. The application can abort the free at that point, or let it continue. In PCR, finalization works as follows: finalization may be requested for any object by passing it to the PCR routine XR RegisterForFinalization. XR RegisterForFinalization returns a *handle* to the object. The handle may be turned into a true pointer to the object at any time, but is not a pointer for purposes of collection (i.e. an object can be finalized while handles for it still exist).

During each collection, after the mark phase but before sweeping, the PCR collector executes the algorithm below:

```
for each unmarked finalizable object o
    for each pointer p in o
        mark p^, and mark all p^'s descendants
for each finalizable object, o, still unmarked
    queue o for finalization by its owning application
```

This algorithm has the difficulty of never finalizing circular lists. The circularity of such lists must be broken by using a handle produced by XR RegisterForFinalization for one of the links, rather than an actual pointer. The handle isn't treated as a pointer by the collector, so finalization still occurs.

An alternative implementation of finalization, used in the Cedar reference counted storage system[Rovner 1985], is a dangerous technique called *package ref counts*. It involves artificially lowering the reference count stored in each finalizable object. and having no way to tell a known from an unknown reference. We believe that our method, using explicit handles that can be turned into pointers, is safer and less error prone.

Finalization is tricky, however it is done, but it is not frequently programmed directly. For instance, in the two million lines of Cedar code in use at PARC, only twelve calls register objects for finalization. Requiring careful programming in the use of this feature is therefore reasonable. However, doing without finalization is not possible: the twelve kinds of finalizable objects in Cedar include stream and network I/O objects, so nearly all applications indirectly use finalization.

3.3.3 Improving performance of the conservative collector

The PCR collector is conservative and so works even for languages that permit any word in memory to contain a pointer (such as C). However, for some languages (such as Cedar and Lisp) it is possible to tell exactly which bit patterns in memory are pointers. For these languages, collector speed can be improved because only words containing pointers need to be examined. Collection precision may also be improved because false pointers will not unnecessarily hold storage under the conservative assumption, although for at least one test this effect was small [Boehm and Weiser 1988]. Another reason for non-conservative pointer-finding is that some language implmentations use non-standard pointer representations to improve non-pointer performance (e.g. tagged pointers in some Lisp implementations). For all these reasons, the PCR design incorporates the notion of a pointer-finding upcall. The pointer-finding upcall works as follows:

Each object is typed by the kind of pointer-finding upcall needed to deduce its pointers. There can be as many different upcalls as needed to find pointers in objects: one for tagged pointers, one finding pointers according to datatype-dependent pointer maps, another for entirely conservative pointer-finding, etc. New upcalls are introduced by registering them with the collector and receiving in return an upcall type code. The upcall type code is an optional additional parameter to object creation and is permanently associated with the created object. During the mark phase of collection, the collector uses the upcall associated with each object to find the pointers it contains. In the absence of an upcall, fully conservative pointer-finding is used.

Our largest PCR applications do not use the upcall at all--they run completely conservative, even though in the case of Cedar we theoretically have enough type information to be more precise. Our experience with the upcall is in a special version of PCR that uses the upcall for compatibility with a CommonLisp implementation that uses tagged pointers. The cost of the upcall is about a microsecond per object on a 16 Mhz SPARC (sun-4/260). This is roughly twice the cost of conservatively examining a word in the object to see if it is a pointer (which requires at least a range check to see if it could be a value in the heap). Thus the upcall performance is better than the conservative performance if it can reject non-pointers twice as fast as the conservative check, with at least a constant improvement of two pointer checks. For instance, suppose a rate of pointers in objects of 25%. Then an upcall that positively identified pointers in objects, and spent an average of three conservative checktimes per pointer per object doing so, would on average have better performance for all objects of size greater than 5 words (cost of 2 word-times overhead for the upcall, and cost of 3.75 word-times for the 1.25 expected pointers per 5 word-object).

3.4 Symbol Binding and Incremental Loading

Finally, we come to the part of the PCR that does incremental linking and loading, maintaining the symbol tables used for this purpose and for debugging. This part of the PCR consists of two components. The first internalizes object code from external files and the second does symbolic name binding on the internalized code.

Internalizing object code is, of course, dependent on the particular file formats used for object code. The incremental loader reads the code and data portions of the object file into dynamically allocated storage. It creates an internal symbol table corresponding to the symbol table in the file. It also relocates the loaded code and data and records the location of the code and data in a file for later use by the debugger. This file is used to generate a synthetic a.out file containing symbols for all the statically and dynamically loaded code. Debuggers such as dbx and gdb can then be used on dynamically loaded code by using the a.out file as their symbol source.

The second component manages the internalized symbol tables. It is responsible for attempting to resolve undefined symbols in each incrementally loaded file against defining occurrences in previously loaded files and in libraries. Of course, finding a defining occurrence in a library causes the appropriate library component to itself be incrementally loaded.

Each incrementally loaded module is checked for two special names: XR install and XR run. If present, these are called in that order. XR install performs any language-dependent symbol binding (Cedar and Lisp use it, for instance). XR run is the entry point to actually start executing the loaded code.

The incremental loading code is fully compatible with existing Unix programs and libraries. Anything that can be dynamically loaded can also be statically bound into an instance of PCR. This enables us to debug PCR-based applications using dynamic loading, and then, using those same modules, easily construct a single executable program indistinguishable from any other executable binary on the machine. We use this, for instance, to make some of our common tools, like the Cedar compiler and Postscript and Interpress decomposers, look like ordinary Unix programs.

4. Performance

Our early experience with PCR as the foundation of our Cedar programming environment suggested that its performance was not an issue for that use. To quantify this feeling we undertook some measurements of PCR performance.

This section is broken into 4 parts corresponding to overall performance as a user at the keyboard might see it, followed by sections detailing the performance of the PCR components. Times were measured on a Sun-4/260 running SunOS 4.0.1 with the SunView window system up but idle, and no other activity besides the benchmark.

4.1 Overall System Performance

In this section, all times are averages over several runs of combined system and user time as measured by the SunOS getrusage call, reported either directly (for PCR), or from the the cshell 'time' command.

Running a tight loop counting to 30 million takes 31.3 seconds in raw Unix, 31.7 seconds in PCR. This 1% overhead is accounted for by PCR's internal 20-times-per-second clock interrupts, and its 10-times-per-second preemptive rescheduler. This measurement shows the penalty due to PCR for compute-bound jobs.

As another measurement of PCR overhead, we ran the same tight counting loop many times at once, comparing multiprogramming using Unix processes with using PCR threads. For this measurement the number of times through the loop was in each case divided by the number of simultaneous executions, so an ideal completion time would have been the same as for the single tight counting loop, above. The details are in Table 1. This implementation of Unix seems to have a process switch overhead that goes up by 3% from 16 to 32 processes, and again from 32 to 64 processes. These measurements show the benefit of lightweight threads over Unix processes for supporting many compute bound activities simultaneously.

	number of processes				
	1	**2**	**16**	**32**	**64**
UNIX, secs	31.3	32.8	34.7	35.5	36.8
PCR, secs	31.7	33.2	33.9	34.0	34.0

Table 1. Cpu-bound multiprocess times.

Finally, as a realistic example, native unix troff takes 27.7 seconds to process the Unix C-shell (CSH.1) manual entry. With troff dynamically loaded and linked into PCR, the same computation takes 28.0 seconds, exclusive of link and load time. This is the minimum overhead of about 1%.

4.2 Threads

4.2.1 Thread Switch Time

Currently, the cost of a thread switch is about 77 usec (measured by *ITIMER PROF* option of the *getitimer* call). This includes a 35 usec overhead of a trap to the SunOS kernel to save the register-window. It also includes overhead for the debugging features discussed in Section 3.1.3: the ability to freeze and thaw individual threads, to examine blocked or faulted threads, to switch dynamically between single and multiple virtual processors, etc. Some of these features will be greatly simplified or eliminated when the Cirio debugger is completed, and thread switch times should improve as a result, but even the current value would be acceptable. (There is an additional, hidden cost to thread switching, not reflected in our measurements: when a thread is dispatched, its register file is initially empty, so the thread incurs the overhead of "faulting-in" registers as necessary.)

4.2.2 Monitor Entry and Exit Times

An important performance parameter is the cost of acquiring and releasing a monitor lock in the (usual) case that there is no contention for the lock. In PCR the cost of calling a null ENTRY procedure (i.e., one that does nothing but acquire and then release a monitor lock) is 4.2 usec. Currently both locking and unlocking are done by procedure call, so this cost could be improved slightly by in-line expansion. Additional improvement could be achieved, at some expense in debuggability, by eliminating a field containing the identity of the thread holding a lock.

4.3 I/O

A program that opens a file, and then does 100,000 iterations of lseeking to the beginning and reading 1024 bytes takes 23.3 seconds in raw unix, and 31.3 seconds in PCR. This is an overhead of about 40 usec for each of the lseek and read calls. Some of this time is in entering and leaving monitors (two per call), and the rest is the overhead of checking for the special cases for such things as the read blocking or the descriptor not being in the cache. A better tuning for common cases would make a large improvement. On the other hand, 40 usecs overhead for an I/O that will take at least a few milliseconds seems acceptable.

We do not yet have applications large enough to let us make realistic measurements of the PCR file descriptor cache hit ratio. Using a synthetic benchmark, we have measured the cost of a descriptor cache miss to be about 4.6 msec. This figure includes system and user times, both on the virtual processor on which the fault occurred and on the I/O processor owning the desired descriptor.

4.4 Storage management

Our most widely used PCR implementation today (March 1989) uses a collector based on Boehm's [Boehm and Weiser 1988], modified to keep a type word before the first word of the object. Its collection speed is about a half second per megabyte of active object space (assuming no paging). Our newer collector ought to run at a similar speed, but on only 100kbytes at a time and touching many fewer pages [Demers et al 1989].

4.5 Symbol binding and dynamic loading

Dynamically loading an object file into PCR is somewhat faster than processing the file using the Unix ld command. For example, the troff program discussed above is loaded into PCR in 1.3 seconds compared with the 1.7 seconds ld requires to process it. The reason seems to be that ld has many general cases to handle, and must also build an output file, while PCR loads and relocates in place. Of course, the cost of ld'ing is amortized over all the executions of the resulting output file, while the cost of dynamic loading can only be amortized over executions taking place within a single instance of PCR.

5. Related Work

Our work builds on previous research in light-weight processes, garbage collection, library management, etc., and references to these are in the main body of the text. In this section we collect the discussion about alternative approaches to language interoperability.

One current approach to language interoperability, exemplified in Mercury [Liskov et al. 1988] and HRPC [Bershad et al. 1987], uses client-server models of interoperation where remote procedure call connects, and insulates, applications in different languages. The problem here is the lack of tight coupling. Remote procedure call, even when local but across address spaces, is usually much more expensive than calls within the same address space. When the language partitioning and the client/server partitioning match, RPC does well. When they do not match, they force the application writer to introduce artificial distinctions.

Another approach to language interoperability uses a common base language to which other languages must conform. The foreign function call interfaces in Common Lisp [Sun 1988b, Franz 1988], are examples of this approach. The problem here is that the privileged language enjoys easier debugging, better access to services, and more attention from developers. The choice of language in which to write an application becomes distorted by issues beyond appropriate language semantics, and the languages interoperate asymmetrically.

A third approach to language interoperability is to standardize on a common intermediate form. This is a variation on the privileged language approach, permitting different languages to interoperate as long as they use a common compiler back-end. In spite of several attempts in this direction [e.g. Tanenbaum et al. 1983], the restrictions on language designers and implementors have proven too severe for wide adoption. We hope our more modest approach (agreement on important parts of the runtime environment), by analogy with the success of common operating systems, will prove better in practice.

A fourth "approach" is to say that language interoperability is bunk. Either there is one true language, or what really matters is not language but environment. Proponents of these views either focus efforts on inventing new languages to solve all their problems [U.S. DOD 1983] or in developing the single language environment as in Smalltalk [Goldberg and Robson 1983], Interlisp [Xerox 1985], Cedar [Swinehart et al. 1986], or the new DARPA environments proposal [Gabriel 1989]. We think different problems are attacked better in different languages, and that software engineers and computer scientists should not be restricted to a single semantic arrow in their quivers.

6. Experience and Conclusions

The Portable Common Runtime is in daily use by about ten researchers at PARC and by scores of users elsewhere in Xerox. PCR is the lowest-level foundation of future work in PARC's Computer Science and Electronic Documents Labs. We are running about 500,000 lines of Cedar code on top of PCR as of August 1989 [Atkinson et al. 1989]. PCR, itself, is about 20,000 lines of C, and about 200 lines of assembler.

Several of our uses push hard on the PCR facilities. For instance, we have an X window client that creates several threads per window. We also have an Interpress printer driver that reuses lots of free storage, and so stretches the collection facilities. To bring up a full Cedar world on our Sun workstations, more than 60 large modules (totaling over 5 megabytes) must be dynamically loaded.

We routinely use PCR to intercall between C and Cedar, and intercalling with Kyoto Common Lisp receives a small amount of use. A small part of the Kyoto runtime was changed to use the PCR collector and dynamic loader; otherwise it is unchanged. We have tested the dynamic loading and automatic garbage collection of large pre-existing SunView applications, merely relinked to be relocatable instead of executable, and they run fine.

Most of our use of PCR is under SunOS 4.0 on SPARC-based processors. We also have a small amount of 68020 and Mach use. The use that shows PCR's portability best is on CSL's own SPARC-based computer, which has no operating system at all but shares memory with another processor running the Cedar operating system. This PCR has nothing Unix-like nearby to rely upon. Bringing up this PCR from our original SunOS SPARC-based version took less than a month.

For the future, we hope to see PCR in wider use, both inside and outside PARC and Xerox. The source code and documentation for PCR is available from the Computer Science Laboratory at PARC for a copying charge. We hope to interest other portable language efforts, such as C++, Objective C, and Modula-3, in using PCR as their base. We hope to see at least the facilities offered by PCR threads, language-independent garbage collection, and user-controlled incremental linking and loading available in all future operating systems. Finally, we expect to continue push on the enabling technologies for language interoperation. Of particularly critical importance is further work on data representation.

7. Acknowledgements

We would like to acknowledge the contributions to this work by our PARC colleagues. Many people in the Computer Science and Electronic Documents Laboratories have done the porting of Cedar to PCR: Russ Atkinson, Christian Jacobi, Bill Jackson, Michael Plass, Ken Pier, Eric Bier, Jim Foote, Willie-Sue Orr and Eduardo Pelegri. Christian LeCocq and Christoph Cuenod contributed the Cedar support for the standalone PCR implementation. Bryan Lyles converted Unix performance tools to work with PCR. Scott Shenker, Danny Bobrow and Barry Hayes helped us improve our understanding of garbage collection. Gregor Kiczales and Danny Bobrow helped with issues in running LISP on PCR.

8. References

[Accetta et al. 1986]
Accetta, J. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F. Tevanian, A., and Young, M. W., ''Mach: A New Kernel Foundation for UNIX Development'', *Proceedings of Summer Usenix*, July, 1986.

[Apik and Diehl 1988]
Apik, S., and Diehl, S., "Presentation Manager and LAN Manager", *BYTE*, Vol. 13(10), October 1988, pp. 157-159.

[Atkinson et al. 1989]
Atkinson, R., Demers, A., Hauser, C., Jacobi, C., Kessler, P. and Weiser, M., ''Experiences Creating a Portable Cedar'', *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[Bartlett 1988]
Bartlett, J. F., "Compacting Garbage Collection with Ambiguous Roots", DEC Western Research Lab Research Report 88/2, February 1988.

[Bershad et al. 1987]
Bershad, B. N., Ching, D. T., Lazowska, E. D., Sanislo, J. and Schwartz, M., ''A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems'', *IEEE Transactions on Software Engineering* **SE-13**, *8*, August, 1987, pp. 880-894.

[Bershad et al. 1988]
Bershad, B. N., Lazowska, E. D., Levy, H. M., Wagner, D. B., "An Open Environment for Building Parallel Programming Systems", *Proceedings ACM/SIGPLAN PPEALS 1988 Parallel Programming: Experience with Applications, Languages and Systems, SIGPLAN Notices,* Vol. 23(9), September 1988, pp. 1-9.

[Boehm and Weiser 1988]
Boehm, H-J., and Weiser, M., "Garbage Collection in an Uncooperative Environment", *Software-Practice and Experience,* Vol.

18(9), September 1988, pp. 807-820.

[Brinch-Hansen 1975]
Brinch-Hansen, P., ``The Programming Language Concurrent Pascal'', *IEEE Transactions on Software Engineering* **SE-1**, *2*, June, 1975, pp. 199-207.

[Cardelli et al. 1988]
Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G., ``Modula-3 Report'', DEC Systems Research Center, August, 1988.

[Cheriton and Zwaenepoel 1983]
Cheriton, D. and Zwaenepoel, W. ``The Distributed V Kernel and its Performance for Diskless Workstations'', *Proceedings of the Ninth ACM Symposium on Operating Systems Principles,* Bretton Woods, NH, October, 1983, pp. 128-140.

[Clark 1985]
Clark, D. ``The Structuring of Systems Using Upcalls'', *Proceedings of the Tenth ACM Symposium on Operating Systems Principles,* Orcas Island, WA, December, 1985, pp. 171-180.

[Demers et al. 1989]
Demers, A., Weiser, M., Hayes, B., Boehm, H., Bobrow, D., and Shenker, S. "Combining Generational and Conservative Garbage Collection", submitted to the ACM Conference on Principles of Programming Languages, January 1990.

[Franz 1988]
Franz Inc., "Foreign Functions", *Allegro Common Lisp User Guide, Release 2.2*, Section 10, January 1988.

[Gabriel 1989]
Gabriel, D., Ed. "Draft Report on Requirements for a Common Prototyping System", *SIGPLAN Notices*, V. 24, No. 3, March 1989, pp. 93-166.

[Goldberg and Robson 1983]
Goldberg, A. and Robson, D., *Smalltalk-80: the language and its implementation*, Addison-Wesley, 1983.

[Hayes and Baran 1989]
Hayes, F. and Baran N., ``A Guide to GUIs'', *Byte* **14**, *7,* July, 1989, pp. 250-257.

[Hoare 1974]
Hoare, C. A. R., ``Monitors: An Operating System Structuring Concept'', *CACM* **17**, *10*, October, 1974, pp. 549-557.

[Liskov et al. 1987]
Liskov, B., Curtis, D., Johnson, P. and Scheifler, R. ``The Implementation of Argus'', *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles,* Austin, TX, November, 1987, pp. 111-122.

[Liskov et al. 1988]
Liskov, B., Bloom, T., Gifford, D., Scheifler, R., and Weihl, W., "Communication in the Mercury System", *Proc. of the 21st Annual Hawaii Intl. Conf. on System Sciences*, Kailua-Kona, HI, January 1988, pp. 178-187.

[Pu et al. 1988]
Pu, C. and Massalin, H. and Ioannidis, J., "The {Synthesis} Kernel", Computing Systems, Vol 1(1), Winter 1988, pp. 11-32.

[Rovner 1985]
Rovner, P. "On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically Checked, Concurrent Language", Xerox Palo Alto Research Center Technical Report CSL-84-7, July 1985.

[Steele 1984]
Steele, G., *Common LISP: The Language*, Digital Press, 1984.

[Sturgis 1989]
Sturgis, H., ``Cirio: A Multiple Language Symbolic Debugger for the Portable Common Runtime'', Xerox PARC, in preparation.

[Sun 1988a]
Sun Microsystems, ``Lightweight Process Library'', *SunOS Reference Manual, Sun Release 4.0,* 1988, section 3L.

[Sun 1988b]
Sun Microsystems, "Working Beyond the Lisp Environment", *Sun Common Lisp 3.0 Advanced User's Guide*, chapter 5, part no. 800-3049-10, August 1988.

[Swinehart et al. 1986]
Swinehart, D., Zellweger, P., Beach, R., Hagmann, R., ``A Structural View of the Cedar Programming Environment'', *TOPLAS* **8**, *4*, October, 1986.

[Tanenbaum et al. 1983]
Tanenbaum, A.S., van Staveren, H., Keizer, E. G., Stevenson, J. W., "A Practical Tool Kit for Making Portable Compilers", *Communications of the ACM*, Vol. 26(9), September 1983, pp. 654-660.

[U.S. DOD 1983]
U. S. Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815 A, January, 1983.

[Xerox 1985]
Xerox Corporation, *Interlisp-D Reference Manual*, October, 1985.