Performance implications of multiple pointer sizes

Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, Amitabh Srivastava
Digital Equipment Corporation Western Research Laboratory

Abstract

Many users need 64-bit architectures: 32-bit systems cannot support the largest applications, and 64-bit systems perform better for some applications. However, performance on some other applications can suffer from the use of large pointers; large pointers can also constrain feasible problem size. Such applications are best served by a 64-bit machine that supports the use of both 32-bit and 64-bit pointer variables.

This paper analyzes several programs and programming techniques to understand the performance implications of different pointer sizes. Many (but not all) programs show small but definite performance consequences, primarily due to cache and paging effects.

1. Introduction

> *There is only one mistake that can be made in computer design that is difficult to recover from--not having enough address bits for memory addressing and memory management [4]* (p. 2*).*
>
> *Smaller is faster [11]* (p. 18*).*

Whenever someone has declared that a computer has more than enough addressing bits, time (and not much of it) has proven otherwise. Just when systems with 32-bit addresses have become commonplace, many users have found this inadequate; their problems are too large to conveniently implement using 32-bit pointers. While one can sometimes extend a 32-bit architecture with tricks such as segmentation, ultimately the only efficient and clean solution is to use a large, flat address space. Experience suggests that only power-of-two pointer sizes make sense, and indeed most major vendors are either shipping or planning 64-bit systems.

An increase in address size is a discontinuity: the fraction of ‘‘interesting’’ bits in a pointer shrinks a lot. Programs that were pushing the limits of the old address space consume almost none of the new address space.

While some programs will soon grow to need the new address space, many (if not most) will not. A large fraction of the storage used for pointers often goes to waste. A program that needs 33 bits of address space will not run at all on a 32-bit system, but a program that runs happily on a 32-bit system will ‘‘waste’’ more than half of the bits in every 64-bit pointer.

Does this hurt performance? In this paper, we examine a number of ways in which pointer size affects performance for programs that do not need addresses larger than 32 bits. We will show that the effects depend on many variables, including problem size, intensity of pointer use, memory system design, and luck. In many (perhaps most) cases, performance is independent of pointer size, but sometimes, using too-large pointers can result in extra cache misses, TLB faults, and paging.

64-bit systems not only support larger addresses; they also support larger integer data types (most 32-bit systems already support 64-bit floating-point types). Large integers are clearly useful; for example, a 32-bit unsigned counter that increments every microsecond will overflow in about one hour. We will not, in this paper, examine the performance consequences of large integers, because programmers have understood this issue for many years and because all 32-bit and 64-bit systems allow programmers to control the size of integer variables.

Many programs really do need addresses larger than 32 bits, so 64-bit systems are essential and inevitable. We argue, on the evidence of the experi-

ments reported on this paper, that the best system design gives programmers a choice between 32-bit and 64-bit pointers, on a program-by-program or even variable-by-variable basis.

2. Related work

Microsoft's Windows™ system supports several different kinds of pointer, because the underlying Intel 80286 memory architecture does not directly support a 32-bit flat address space [15]. (Windows also runs on the Intel 80386, which has a flat 32-bit addressing model, but applications meant to be portable must be compiled to use the more restrictive 80286 model.) 80286 addresses use a 16-bit segment number and a 16-bit offset, and Windows cannot guarantee that data segments are contiguous. Thus, when a program uses a 32-bit pointer, the compiler generates code to extract the segment number and offset, loads the segment register, and then uses the offset to reference the memory object. This makes 32-bit pointers far more expensive than 16-bit pointers, but for reasons unrelated to the issues discussed in this paper. (Programs compiled for the Win32s interface use the 80386 addressing model, and so run more efficiently but cannot be used on 80286 systems.)

Our comparison of applications using 32-bit and 64-bit pointers presupposes that one has the option of using small pointers on a 64-bit system. Some applications do use larger data sets than can be addressed using 32-bit pointers. Moreover, some research projects into operating system design are using 64-bit addresses to accomplish other goals; this means that even the smallest application must use 64-bit addresses. Such systems rely on large addresses; 32-bit pointers simply would not suffice.

Several research groups are examining the use of 64-bit architectures to build single-address-space protected operating systems [6, 7, 27]. In such systems, all protected objects (processes, in particular) share a single address space. The system prevents unauthorized access not by modifying the virtual memory map on each context switch, but by hiding each object at a randomly chosen location in a large, sparsely populated address space. An object's address acts as a capability, since the full address space is too large for a malicious or buggy program to search for an object whose location has not been obtained by proper means.

Carter et al. have proposed building a distributed shared memory (DSM) system using a large address space to give a process direct access to memory objects spread across many nodes [5]. Although most current systems cannot support enough real memory to exhaust a 32-bit address space, the aggregate memory of a large number of such systems could easily exceed the range of a 32-bit address, especially if segmented for convenience in allocation and management.

3. Technical context

In this section, we discuss computer system technology as it relates to pointer sizes. We use Digital's Alpha AXP™ architecture [23, 24], and implementations thereof, as a specific example. In addition to architecture, we look at issues in hardware implementation and program compilation. (Operating system features affect the cost of translation buffer and page faults, but those issues are beyond the scope of this paper.)

3.1. Architecture

The Alpha AXP architecture is a load-store design with flat (unsegmented) 64-bit byte addresses. The architecture does single-instruction loads and stores of properly aligned 64-bit and 32-bit values; other values are accessed with multi-instruction sequences.

All instructions are 32 bits wide; load/store instructions specify memory addresses using a general-purpose (64-bit) register and a 16-bit signed displacement.

The architecture does not specify a single memory-system page size, but rather allows an implementation's basic page size to be 8 KB, 16 KB, 32 KB, or 64 KB. Page table entries may use ''granularity hints'' to inform the Translation Buffer (TB, sometimes known as a Translation Lookaside Buffer or TLB) that a block of pages can be mapped with a single TB entry; however, in the experiments described in this paper, this feature is not used.

3.2. Hardware implementation

We did the experiments reported on in this paper on several workstations of relatively similar design. Common elements include:

- DECchip 21064-AA CPU [8]
- 8 KB on-chip separate instruction and data caches
  - Physically addressed, direct-mapped, write-through
  - 32-byte blocks
- 4-entry 32-byte/entry on-chip write buffer
- 8 KB page size
- 32-entry on-chip fully-associative data TLB
- 8-entry on-chip fully-associative instruction TLB

- 2 MB board-level cache
  - Direct-mapped, write-back
  - 32-byte blocks

The specific systems differ in CPU clock rate, main memory size, and disk access time. For the tests reported in this paper, we used three different systems:

- **System A**, a DECstation™ 3000/600 (175 MHz clock, SPECint92 = 114.1), with 128 MB of main memory.
- **System I**, a DECstation 3000/800 (200 MHz clock, SPECint92 = 130.2), with 1024 MB of main memory.
- **System N**, a DECstation 3000/500 (150 MHz clock, SPECint92 = 84.4), with 64 MB of main memory. (This system has a 512 KB board-level cache.)

3.3. Compilation system

All the programs described in this paper are coded in C, run on the DEC OSF/1® operating system, and were compiled using the standard DEC OSF/1 V3.0 compilers.

The compiler supports several integer data types: ''short'' (16 bits), ''int'' (32 bits), and ''long'' (64 bits). By default, pointers are 64 bits wide. The compiler supports a `#pragma` statement that allows the programmer to select 32-bit pointers. The programmer can choose a pointer size for an entire module, or for specific declarations within that module. The programmer also specifies at compile time whether the pragma should be interpreted or ignored, so the modified program source can optionally be compiled to use only 64-bit pointers.

For example, this program:

```
char *lp;
#pragma pointer_size (short)
char *sp;
main() {
  printf(
    "sizeof(sp) = %d, sizeof(lp) = %d\n",
    sizeof(sp), sizeof(lp));
}
```

produces

```
sizeof(sp) = 4, sizeof(lp) = 8
```

Normally, the linker arranges programs to start above address 2^{32}; this means that any attempt to dereference a 32-bit value causes an addressing trap, which aids in the detection of portability problems. A program that uses 32-bit pointers clearly cannot be located above 2^{32}, so at compile time the programmer must tell the linker to use its ''truncated address space option,'' which forces all program addresses to lie below that limit. Of course, this cannot be used with programs requiring more than 2^{32} bytes of virtual address space.

Note that the compiler used in these tests generates essentially the same type and number of instructions for both 32-bit and 64-bit pointers. The *only* difference is that loads and stores of pointer variables read or write 32 bits of memory instead of 64 bits. One could do somewhat better than this for some uses of 32-bit pointers, so our measurements may not reflect the entire potential effect of their use.

Because the programs we tested execute essentially the same instruction stream regardless of pointer size, *all* of the performance effects shown in this paper reflect aspects of the memory system (caches, RAM, TLB, and disk).

4. Results for a contrived program

What aspects of system design interact with pointer size to affect performance? We wrote a simple, contrived program (figure 4-1) to illustrate several of these aspects. We designed the program to emphasize the worst-case effects of using large pointers; it most certainly does not represent real programs, and almost any other program will show smaller effects.

The program simply constructs a circular queue and then follows the pointer chain around the queue. It takes three arguments: N, the number of queue elements; M, the number of iterations (pointer dereferences); and P, the number of references between random ''seeks'' to a different part of the queue. If $P = M$, then no random seeks are done (after the first one); if $P = 1$, then every reference is to a randomly-chosen element.

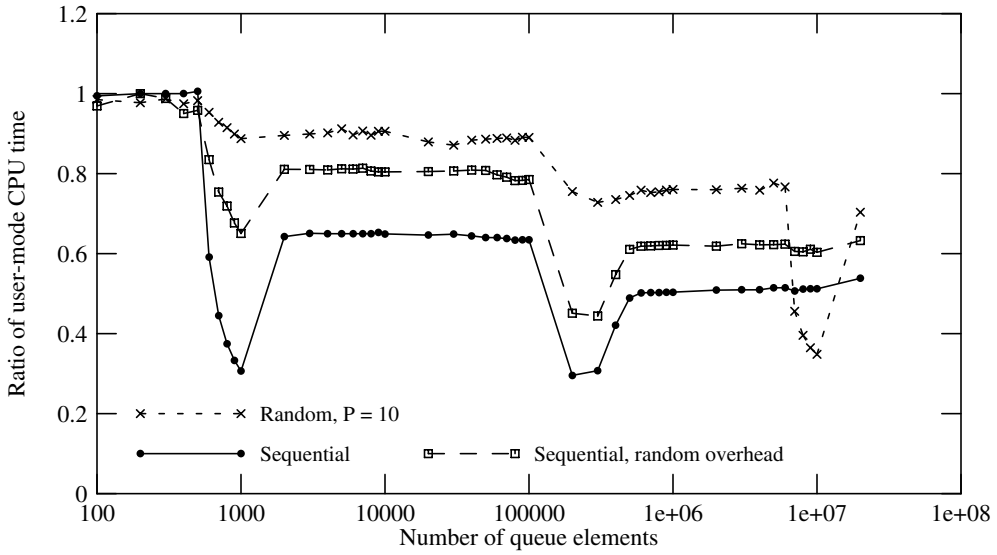
The program can be compiled to use either 32-bit pointers or 64-bit pointers. We ran each version with a pseudo-logarithmic series of values for N, large values of M (at least 10,000,000), and two values of P ($P = M$, effectively sequential access, and $P = 10$, almost random access). We measured the user-mode and kernel-mode CPU time, and the elapsed time, for each run.

We plotted the ratios of the times taken by the 32-bit version to the times taken by the 64-bit version; this is more useful than a plot of the actual times, which vary widely. Figure 4-2 shows the ratios of user-mode CPU times; figure 4-3 shows the ratios of kernel CPU times; figure 4-4 shows the elapsed-time ratios.

From figure 4-2 one can see that for small working sets, the pointer size has little or no effect on performance. Once the entire data set no longer fits in the 8 KB on-chip cache, however, the cost of a memory reference increases significantly. Because

```
 1  #include <stdio.h>
 2
 3  #pragma pointer_size (short)
 4
 5  struct QueueElement {
 6    struct QueueElement *forward;
 7    struct QueueElement *backward;
 8  };
 9
10  main(argc, argv)
11  int argc;
12  char **argv;
13  {
14    long N = atoi(argv[1]);
15    long M = atoi(argv[2]);
16    long P = 1000;
17    int rval;
18    int i, j;
19    struct QueueElement *qp;
20    struct QueueElement *qpstore;
21
22    if (argc > 3)
23      P = atoi(argv[3]);
24
25    qpstore = (struct QueueElement *)
26                malloc(sizeof(*qp)*N);
27    if (qpstore == NULL) {
28      perror("malloc");
29      exit(1);
30    }
31
32    qp = qpstore;
33
34    qp->forward = qp;
35    qp->backward = qp;
36
37    /* make queue elements */
38    for (i = 0; i < N; i++) {
39      /* Insert next element
40         after head of queue */
41      qp[i].forward = qp[0].forward;
42      qp[i].backward = &qp[0];
43      qp[i].backward->forward = &qp[i];
44      qp[i].forward->backward = &qp[i];
45    }
46
47    for (i = 0; i < M/P; i++) {
48      rval = random() % N;
49      qp = &(qpstore[rval]);
50      for (j = 0; j < P; j++) {
51        /* follow pointer */
52        qp = qp->forward;
53      }
54    }
55  }
```

Figure 4-1: Contrived test program



Ratios are *32-bit pointer time/64-bit pointer time*

Figure 4-2: Ratios of user-mode CPU times for contrived program

2000 32-bit pointers (1000 queue elements) fit in 8 KB, but only 1000 64-bit pointers (500 queue elements) fit, the user-time ratio drops precipitously at N = 500. It recovers somewhat at N = 2000 (where even with 32-bit pointers the on-chip cache is too small), but because a cache block holds four queue elements with 32-bit pointers, and only two queue elements with 64-bit pointers, in the sequential case the 64-bit version cache-misses about twice as often.

For the random-access case (P = 10), the cost of frequently generating random numbers partially evens out the ratio of user-mode CPU times. To show this effect, we include the curve in figure 4-2 marked ‘‘Sequential, random overhead’’. This is for a version of the program that calculates a new random value every 10th iteration, as in the P = 10 case, but then ignores this value and runs through the queue sequentially.

The next obvious dip in the user-mode CPU time ratio comes when the working set no longer fits into the 2 MB off-chip cache. This cache holds about 256K queue elements with 32-bit pointers, but only 128K elements with 64-bit pointers.

Figure 4-3 shows the ratios of kernel-mode CPU times, which are rather ''noisy'' because they include extraneous system activity. These ratios generally track the user-mode ratios until the 64-bit version starts paging, because both programs incur relatively constant kernel-mode overheads of about 1%-2%[1]. Once paging begins, the (kernel-mode) CPU cost of handling page faults becomes comparable to the user-mode CPU time. When the problem size is large enough that both programs are paging, the kernel-mode ratio returns to about 50%, since the 32-bit version page-faults half as often.

The elapsed time ratios, plotted in figure 4-4 on a log-log scale, show that cache-miss effects dominate until the problem size exceeds the memory size, at which point paging latency becomes the bottleneck. The ratio drops when the 64-bit version starts to page, then recovers (to approximately 0.55) once the 32-bit version starts paging.

4.1. TLB effects

Each TLB miss causes invocation of a handler that runs with interrupts disabled, and thus does not directly appear in the CPU time statistics sampled by the interrupt clock. We would expect to see TLB miss costs reflected as an increase in the apparent user-mode CPU time. This should appear as the working set exceeds the span of the data TLB (32 entries mapping a total of 256 KB), increasing the TLB fault rate (simulations confirm this). The ratio curves should show a change at 16K queue elements, but there is no such dip in the sequential-case curve in either of figures 4-2 or 4-3. The random-case curve in figure 4-2 does show a small dip. Apparently, the cost of extra TLB misses is insignificant even for this rather stressful program.

4.2. Dependence on data layout

The results we measured for this program depend entirely on how its data structures are laid out. In fact, our first attempt at a test program, which allocated a separate chunk of memory for each queue element (that is, made *N* calls to *malloc()* instead of one call), showed essentially no difference between the 32-bit and 64-bit versions. On this system, *malloc(x)* always consumes at least 32 bytes of memory for any $x <= 24$, so both the 32-bit and 64-bit versions of the first-attempt program consumed the same amount of space, and both versions put exactly one queue element in each cache block.

4.3. Dependence on code scheduling

The contrived programs described above do nothing but follow pointers, and occasionally generate random numbers. Even the simplest real program usually does some computation on the objects it finds while following pointers. We modified the program in figure 4-1 by adding a `float data` field to the `QueueElement` structure; this increased the size of the structure by 50%, for both 32-bit and 64-bit pointers. We modified the inner loop (lines 50-53 in figure 4-1) in two slightly different ways. Version A is:

```
for (j = 0; j < P; j++) {
  qp->data = (i + j + 3.3)/(j + 1.0);
  qp = qp->forward;
}
```

Version B is:

```
for (j = 0; j < P; j++) {
  struct QueueElement *qpn;
  qpn = qp->forward;
  qp->data = (i + j + 3.3)/(j + 1.0);
  qp = qpn;
}
```

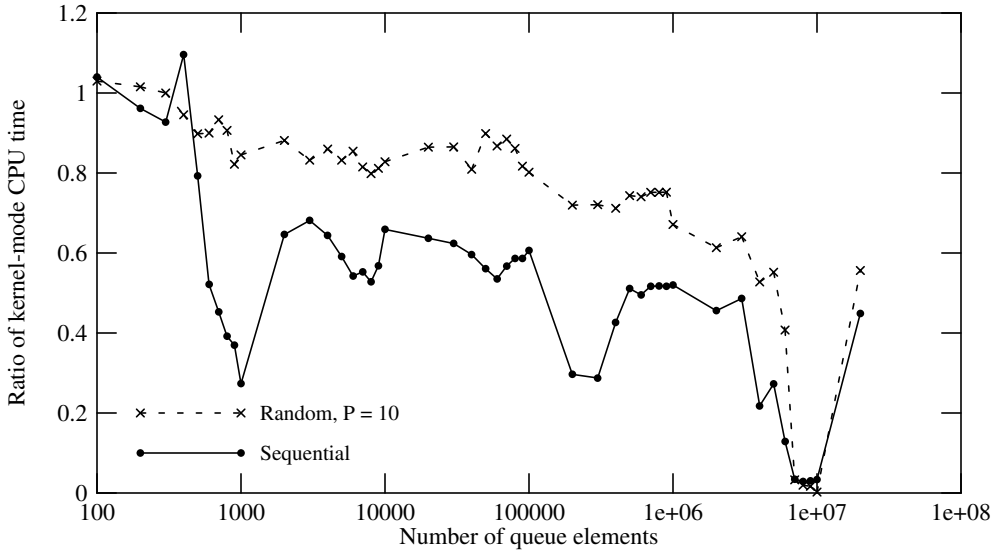
We then ran a set of trials of each program (with $P = 10$), and plotted the total CPU time (user and kernel combined) in figure 4-5.

The performance of either version of the compute-intensive shows much less dependence on pointer size than does the original, pointer-intensive program (note that the vertical scale in figure 4-5 does not start at zero). This should not be a surprise, since the compute-bound program spends a smaller fraction of its time doing pointer operations.

This only explains part of the difference, however, since version B of the compute-bound program shows a generally closer ratio than does version A (and version B is slightly faster, in absolute terms). Version B manages to bury some of the latency of loading `qp->forward`, because it can do the arithmetic computation before it needs to use the loaded value. The CPU thus avoids some of the pipeline stall that would otherwise occur. Even for version A the compiler manages to bury much of the load latency; the use of the auxiliary variable `qpn` simply increases this effect slightly.

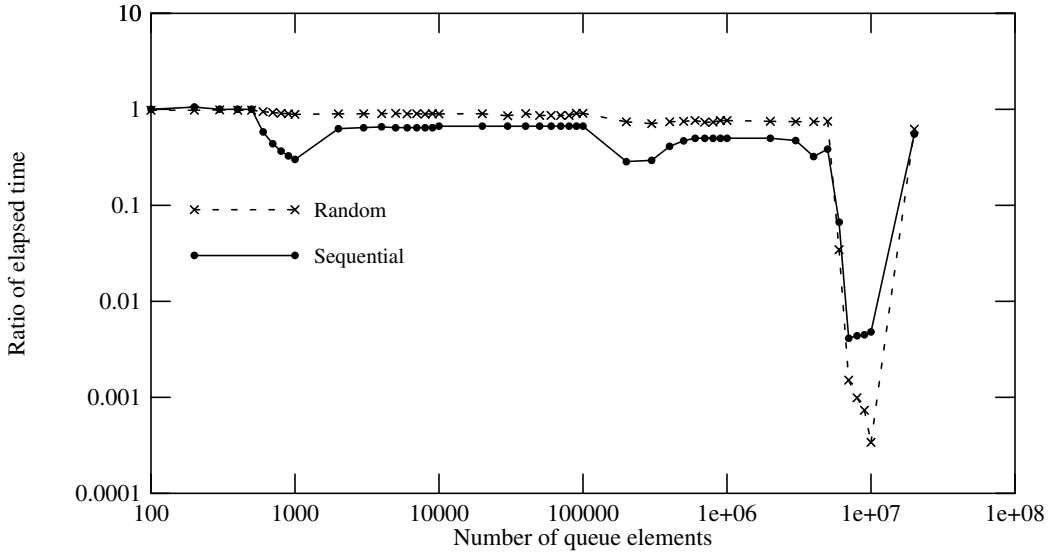
So, a real program may not see some or all of the additional load latency imposed by using larger

[1]We believe this overhead is due to device interrupts in progress when the interrupt clock samples the CPU state. This does not necessarily mean that the system spends 1%-2% of its time fielding interrupts [14].



Ratios are *32-bit pointer time/64-bit pointer time*

Figure 4-3: Ratios of kernel-mode CPU times for contrived program



Ratios are *32-bit pointer time/64-bit pointer time*

Figure 4-4: Ratios of elapsed times for contrived program

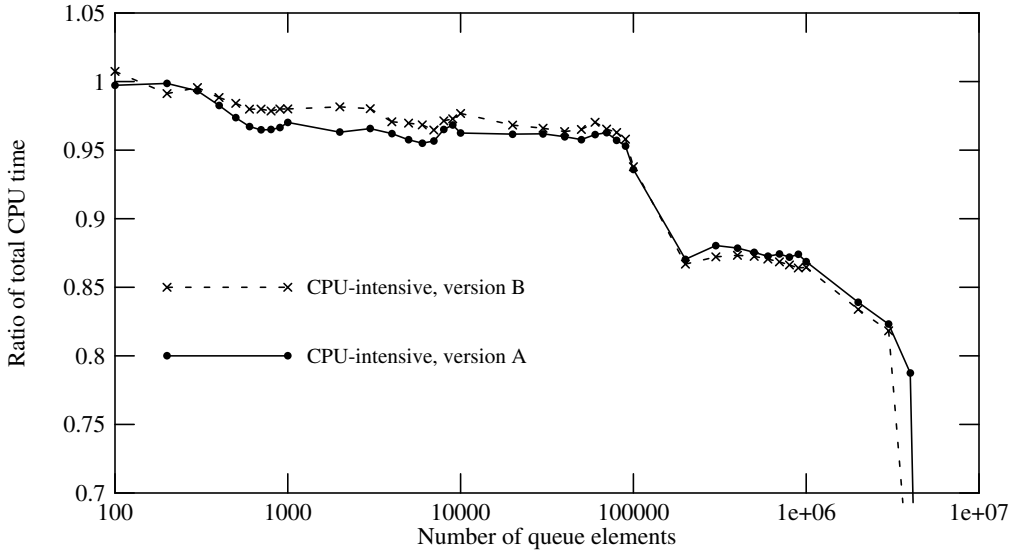
pointers. This depends on the application, the quality of the compiler, and the architecture and implementation of the CPU. The trend in compiler and CPU technology is toward greater tolerance of load latency [9], so one should expect the pointer-size effects to diminish over time. (Store latencies usually do not affect program performance, because modern CPUs and caches can perform stores asynchronously, using mechanisms such as write-buffers and write-back caches.)

4.4. Summary: contrived programs

To summarize what we learned from our contrived programs, with larger pointers:

- Cache misses may increase, as the same number of data items can require more cache lines.
- TLB faults may increase, but probably not enough to worry about.
- Page faults may increase, as the working set increases.

Nevertheless, it is actually rather hard to contrive a program that displays any significant performance dependence on pointer size. Accidents of memory layout, and compiler or CPU techniques to hide load latencies, can reduce or eliminate the cache-related



Ratios are *32-bit pointer time/64-bit pointer time*

Figure 4-5: Ratios of total CPU times for compute-intensive program

performance effects. The remaining effect of larger pointers, the earlier onset of paging, afflicts only rather large programs, but may be harder to ameliorate.

5. Results for real programs

Because the results for our contrived program may not represent what happens in real programs, in this section we present measurements for a variety of more-or-less real programs. All tests were run on System A, except where noted.

5.1. Selected SPEC integer benchmarks

The programs in the SPECint92 benchmark suite were intended to be reasonably realistic examples of programs that actual users run. For most of the C-language programs in this suite, we compiled and measured versions using both 32-bit pointers and 64-bit pointers. We did not measure *gcc*, because this program cannot easily be ported to use true 64-bit pointers. Also, we made no attempt to recreate the compiler flags and measurement conditions used for the official SPEC reports; one should not take our measurements as actual SPEC benchmark values.

The SPECint92 C-language programs are [20]:

- *compress*: A file compression program using Lempel-Ziv coding. The same binary is also used to *uncompress* files.
- *eqntott*: Translates a boolean equation into a truth table.
- *espresso*: Generates and optimizes Programmable Logic Arrays.
- *li*: Lisp interpreter running a recursive backtracking algorithm to solve the ‘‘nine queens’’ problem.
- *sc*: A spread sheet program, calculating several different problems.

Table 5-1 reports our results, expressed as the ratio of times for the 32-bit version to times for the 64-bit version. These timings include a little overhead for execution of measurement scripts, so the actual ratios might be slightly larger than the reported ratios (the overhead is less than 1% of the user-mode CPU time, and somewhat larger for kernel-mode CPU time and elapsed time). We report the mean times for at least 10 trials of each benchmark; in some cases, we had to run many more trials, to get run times long enough for accurate measurement.

We include in figure 5-2 a few additional measurements of *compress* and *uncompress* (version 4.0) operating on a larger file than the 1 MB input used in the SPECint92 benchmark. Note that the use of 32-bit pointers slightly hurts performance on *eqntott*, *uncompress* and the SPEC-related trials of *compress*, but improves the performance of *compress* applied to a larger file. We ascribe this to cache access patterns, but have not yet confirmed that.

5.2. Late code modification

Compilers typically perform optimizations when compiling individual modules of a large program. One can do certain additional optimizations only on the entire program as a whole. A technique called ‘‘late code modification’’ performs these further optimizations at program link time.

Application	Number of trials	User-mode CPU time ratio	Kernel-mode CPU time ratio	Elapsed time ratio
compress	150	1.008	1.01	1.01
uncompress	150	1.005	1.00	1.01
eqntott	10	1.001	0.98	1.00
espresso	10	0.95	0.98	0.96
li	10	0.96	0.96	0.96
sc	10	0.98	0.98	0.98

Ratios are mean of *32-bit pointer time/64-bit pointer time*

Table 5-1: Performance ratios for selected SPECInt92 programs

Application	Input bytes	Output bytes	Number of trials	User-mode CPU time ratio	Elapsed time ratio
compress	3397159	1488027	10	0.993	0.98
uncompress	1488027	3397159	10	1.008	1.05

Ratios are mean of *32-bit pointer time/64-bit pointer time*
All output directed to `/dev/null`

Table 5-2: Performance ratios for compression of larger files

OM [25, 26] is an optimizing linker that does late code modification. OM translates the object code of the entire program into symbolic form, recovering the original structure of loops, conditionals, case-statements, and procedures. It then analyzes this symbolic form and transforms it by instrumenting or optimizing it, and generates executable object code from the result. OM makes heavy use of pointers, because its internal representation of a program captures the many relationships between program objects such as procedures, variables, basic blocks, etc.

We ran OM on a number of input programs:

- *scixl*, a Scheme interpreter with X-library stubs (see section 5.4).
- *fea*, a finite element analysis tool.
- *vcr*, a VLSI circuit router.

The *fea* and *vcr* programs are pseudonyms for real programs with substantial market share; for contractual reasons, we cannot give their actual names.

Table 5-3 shows our measurements. We ran trials both on System I, which has a lot of memory, and System A, on which OM's processing of *fea* and *vcr* exceed the available memory. This causes System A to page. (It also increases the elapsed times from minutes to hours, and so we could not run many trials on System A.) The kernel-CPU and elapsed time ratios in this table may be somewhat inaccurate, since we could not easily eliminate other activity during these trials.

Table 5-4 shows how much space OM requires to process each target program. In general, the elapsed time ratios in table 5-3 mirror the size ratios in table 5-4, except in one case. Why, for *vcr* on System A, do 32-bit pointers outperform 64-bit pointers by so much, in terms of elapsed time? System A has just under 128 MB of real memory; OM's processing of *vcr* requires just a bit more than that using 32-bit pointers, but quite a lot more using 64-bit pointers. The 32-bit pointer version does far less paging, and so completes much sooner. For both pointer sizes, System A does not page when processing *scixl*, and pages heavily when processing *fea*, so for these programs the elapsed-time ratios correspond to the size ratios (although for *fea*, System A is too slow to be feasible with either pointer size).

These results confirm the lessons of section 4, that smaller pointers usually perform slightly better (and more generally, that performance ratios correspond to size ratios). However, when larger pointers push the working set beyond the size of a cache or real memory, small pointers may show a dramatic advantage.

5.3. Corner-stitching in the Magic CAD system

Many VLSI designers employ the *Magic* CAD system [18] to lay out and process their chips. Most VLSI designs can be expressed as a set of rectangles; Magic represents these rectangles and their positions using an algorithm called ‘‘corner-stitching’’ [17].

Program	Test system	Number of trials	User-mode CPU time ratio	Kernel-mode CPU time ratio	Elapsed time ratio
fea	System I	10	0.94	0.90	0.93
	System A	3	0.92	0.84	0.84
scixl	System I	10	0.95	1.00	0.91
	System A	3	0.96	0.85	0.87
vcr	System I	10	0.93	1.00	0.89
	System A	3	0.91	0.45	0.38

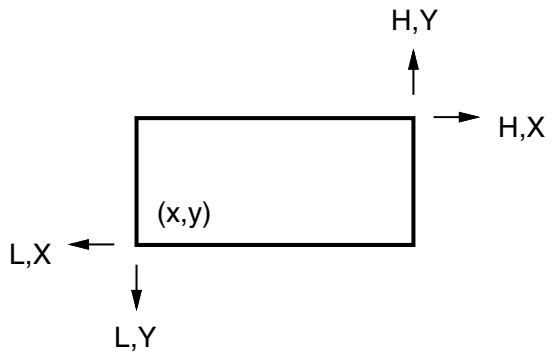
Ratios are mean of *32-bit pointer time/64-bit pointer time*

Table 5-3: Performance ratios for OM-optimization of selected programs

Program	Total object file size	Space required using 32b pointers	Space required using 64b pointers	Space ratio	Page-in ratio
fea	48080 KB	242360 KB	289264 KB	0.84	0.83
scixl	7792 KB	33448 KB	40488 KB	0.83	None
vcr	27184 KB	128328 KB	155800 KB	0.83	0.34

Table 5-4: OM's space requirements for selected target programs

A Tile, the basic object in the corner-stitching algorithm, has this structure:



The data structure for a Tile stores the (x, y) coordinates of its lower left corner, and four ''corner stitches'': (L,X), (L,Y), (H,X), and (H,Y). The ''stitches'' point to neighboring Tiles. For example, the (H,Y) stitch points to the rightmost top neighbor of the Tile. The data structure must also store some information about the nature of the Tile (for example, its layer in the VLSI design).

The minimal representation for a Tile requires four pointers and three integers. Since all coordinates are integral, the integers easily fit into 32 bits. Using 32-bit pointers, a minimal Tile takes 28 bytes; using 64-bit pointers, a minimal Tile takes 44 bytes, although a C compiler would normally pad this to 48 bytes, to naturally align the 64-bit fields within the structure. Thus, the use of 64-bit pointers increases the minimal Tile size by about 70%.

Magic actually adds two pointers, rather than one integer, to the coordinates and stitches, for a total of two integers and six pointers. Therefore, in Magic a Tile requires 32 bytes using 32-bit pointers, and 56 bytes using 64-bit pointers, for a net increase of 75%.

It takes many Tiles to represent a modern VLSI design. For example, BIPS-0, a 32-bit MIPS processor without floating point or virtual memory support [12], required about 3.6 million Tiles. This would need about 112 MB for Tile storage using 32-bit pointers, and 197 MB using 64-bit pointers.

We measured Magic performance on three much smaller designs, a communications interface for a multiprocessor [22], a memory chip [1], and a mesh router chip [10].

We ''flattened'' the cell hierarchy of the communications interface before running it through Magic; the flattened version requires 208801 Tiles. These Tiles should occupy 6525 KB using 32-bit pointers, and 11419 KB using 64-bit pointers. In fact, to represent and process this design using 32-bit pointers, Magic allocates 9408 KB in addition to its initial memory requirements; using 64-bit pointers, it allocates 13392 KB. Thus, the design-specific memory use increases by only 42%, because Magic allocates quite a bit of non-Tile storage. (Most of the rest of this storage is not actually design-specific, but technology-specific: for example, design rules for the specific CMOS process with which the chip will be fabricated.) The total memory use increases by only 26%, because Magic's other memory use is essentially independent of pointer size.

We used Magic to do a ‘‘design-rule check’’ (DRC) on these three chips. (Design rules specify things such as minimums for rectangle width, spacing, and overlaps.) A DRC spends much of its time following corner-stitch pointers, and doing simple geometric calculations; it should exhibit moderately good locality of reference. In each trial, we had Magic do 50 DRCs of the chip. Table 5-5 shows the mean results over 10 trials, expressed as the ratio of times for the 32-bit version to times for the 64-bit version. The table also shows relative space requirements for all three chips.

For the communications interface chip, the use of 64-bit pointers appears to reduce performance (both user-mode CPU time, and elapsed time) by about 5%. Kernel-mode CPU time shows an even larger change, but has little effect on the elapsed time because it represents only 1.5% of the total CPU time. For the memory chip, pointer size has a smaller effect on user-mode CPU time, probably because far fewer Tiles are used. Kernel-mode CPU time represents a larger fraction (about 5%) of the total CPU time on this problem, so it contributes slightly more to the change in elapsed time.

We believe that the additional kernel-mode time comes mostly from the additional page faults. Almost none of these page faults involve the backing store, because we had plenty of main memory on the test machine. Most are ‘‘zero-fill’’ faults used to add new pages to the address space, and the 64-bit version does about 22% more of these for the communications chip, and about 6% more for the memory chip.

The use of 64-bit pointers appears to impose a relatively small CPU-time cost on Magic. However, for the BIPS-0 design, Magic with 64-bit pointers needs almost 100 MB of additional memory. Put another way, on a workstation with 128 MB of main memory, it would be feasible to design-rule check BIPS-0 only using a 32-bit pointer version of Magic; with 64-bit pointers, the system would page excessively. The most important effect of larger pointers is not the slight increase in CPU time, but the much stricter constraint on feasible problem size.

5.4. Garbage collection in Scheme

Some modern programming languages support ‘‘garbage collection’’ instead of explicit deallocation of dynamic storage. Garbage collection makes programs much simpler to write (one no longer has to worry about forgetting to free a data item, or freeing it too many times, or at the wrong time). It also may improve the performance of code that manipulates complex structures, since it obviates the need to maintain reference counts or to call explicit deallocation routines. Here we examine the performance of garbage collection in an implementation of the Scheme programming language [2, 19].

Garbage collection complicates the performance picture. (For a full discussion of the cache-related effects of garbage collection, see Reinhold [21].) This implementation uses a ‘‘mostly-copying’’ algorithm [3], which requires that a pool of free space be kept available. The total size of the garbage-collected address space, including live storage and the overhead, is called the ‘‘heap.’’ When the heap gets too small, the cost of garbage collection becomes excessive. Even with enough headroom to work in, garbage collection has a run-time cost, which depends somewhat on the size of objects used.

Use of larger pointers can affect the performance of garbage collection:

1. It increases the storage-allocation rate (the rate at which address space is consumed). This, in turn, increases the frequency of garbage collection.
2. It increases the cost of each garbage collection phase, since this cost is proportional to the amount of live storage.
3. It reduces the number of structures that fit into a given amount of real memory, and thus may cause the garbage collector to run out of headroom.

In the worst case, the first and second effects would each increase run-time linearly with the increase in structure size. That is, if a change in pointer size adds 50% to the size of a garbage-collected structure, one would expect the rate of address space consumption to increase by 50%. Assuming that the number of live structures does not change, the amount of data copied during the collection phase would also increase by 50%.

In practice, run-time costs increase less than linearly with pointer size. Programs do things besides consume address space. During garbage collection, the cost of copying an object includes fixed overheads not dependent on object size. Larger objects may show better locality of reference during copying.

The third effect, the loss of headroom available to the garbage collector, cannot be analyzed so easily. For any given application working on a specific problem, some minimum amount of memory is sufficient to support the garbage collector without excessive overhead. When the available memory is less than that amount, garbage-collection costs increase dramatically, and may become effectively infinite.

Design	Total space ratio	Number of tiles	Tile space ratio	User-mode CPU time ratio	Kernel-mode CPU time ratio	Elapsed time ratio	Page-fault ratio
Communications interface	0.79	208801	0.70	0.95	0.93	0.95	0.83
Memory	0.83	55904	0.73	0.96	0.96	0.96	0.87
Mesh router	0.85	25046	0.85	0.96	0.96	0.96	0.90

Ratios are mean of *32-bit pointer value/64-bit pointer value*

Table 5-5: Performance ratios for design-rule checking in Magic

(Normally, the garbage collector expands its heap as necessary, but the user can limit the expansion to keep it from overflowing real memory; this would cause the entire program to page excessively.)

We measured the performance of our Scheme system compiled to use either 64-bit pointers or 32-bit pointers for its primary data type (we kept all other pointers at 64 bits). To maintain the required 64-bit alignment for some data objects, the 32-bit version must in some cases pad the pointer fields in its data objects to a 64-bit boundary.

For a sample application, we ran the Scheme->C compiler [2], a Scheme implementation that achieves high portability by using C as its intermediate language, and used it to compile the largest source file in an application called *ezd*. Figure 5-1 shows the ratios of 32-bit times to 64-bit times for elapsed time, garbage collection CPU time, and application (non-GC) CPU time. These tests were run on System N.

One can see from figure 5-1 that the application CPU time depends somewhat on pointer size (use of 64-bit pointers costs about 5%), probably because Scheme programs manipulate pointers heavily. The application CPU time varies little with heap size, because the heap size does not really affect the nature of pointer references made during application execution.

The garbage-collection CPU time depends strongly on heap size, for heap sizes below a threshold: about 7 MB for 32-bit pointers, and about 12 MB for 64-bit pointers. In other words, the use of larger pointers reduces the garbage-collection headroom. Even for heap sizes large enough to reduce the cost of garbage collection below one per cent of the total CPU time, the use of larger pointers seems to add at least 23% to the cost of garbage collection. We believe this results from the increased size of ‘‘live’’ data copied during a garbage collection.

The total elapsed time depends mostly on garbage collection time, for heap sizes below the headroom threshold. For sufficiently large heaps, garbage collection minimally affects elapsed time, and because the application does some I/O, the elapsed time ratio approaches unity.

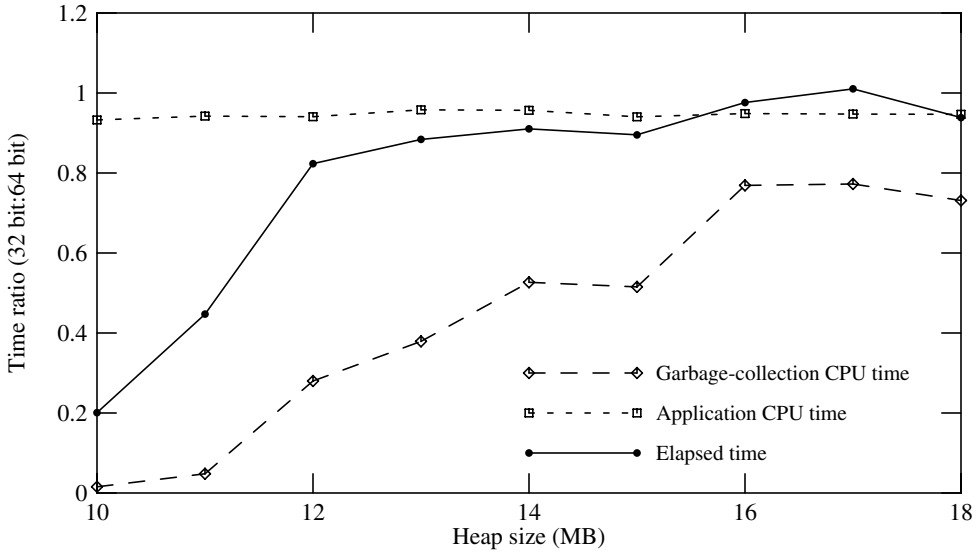
In summary, for this application we found that the use of larger pointers increases the amount of heap (main memory) required for reasonable performance, and slightly increases application CPU time. Large pointers do not significantly affect the running time of the application *if* sufficient RAM is available for the garbage collector; otherwise, they seriously increase elapsed time.

5.5. Sorting

Unnecessarily large pointers could also hurt the performance of pointer-based sorting on large data sets. A sort program spends a large part of its time exchanging the order of records. It can do this by exchanging the records themselves, by exchanging the keys along with pointers to the full records, or by exchanging just the pointers. Pointer sort may be the most efficient technique if the keys are large, especially if the full array of pointers can fit into the CPU board-level data cache. For keys of moderate size, it might be more efficient to use a key sort, which keeps the keys and the pointers together (and so increases locality) [16]. Some phases of a key-based QuickSort can run entirely in a small, on-chip cache [13].

If one is sorting less than a few billion bytes, a sort program has no need for 64-bit pointers. Use of larger pointers than necessary will probably reduce the number of keys (or pointers) that can fit into the CPU’s caches, and so reduces the size of the problem that can be sorted without excessive cache-miss overheads.

In an environment that provides only 64-bit pointers, one could implement a pointer-based sort using 32-bit record indices. This requires the execution of several additional instructions each time an index is converted to a pointer, but could be less costly than incurring the extra cache misses imposed by use of 64-bit pointers. It would also require one to maintain separate source-code versions of the sorting program for 32-bit and 64-bit machines.



Ratios are *32-bit pointer time/64-bit pointer time*

Figure 5-1: Time ratios for Scheme application

We do not have access to a state-of-the-art sorting program, but we did measure the performance of the UNIX® *sort* command applied to several large files. This sort program manipulates pointers rather than the actual key values. Table 5-6 shows the results; the performance differences of about 5% are consistent with results from other programs.

6. Future work

The results we presented in section 5 suggest a minor but consistent advantage to the use of smaller pointers, in those programs that heavily use pointers but do not need to address a huge data set. However, we would not want to conclude from these results that small pointers are inherently faster. We suggest that future studies should include:

- A wider set of large benchmark applications; the SPEC benchmarks are relatively small, and most of the other applications we measured are not easily suited for use as benchmarks.
- Additional architectures and CPU implementations; we only measured a single CPU implementation of a single architecture, and some of our results may depend on those particulars.
- Multiprocessor and perhaps distributed applications; pointer size could have significantly different implications in such environments.

Programmers have understood for decades that their choice of integer and real variable size can affect performance, and most modern programming languages allow such choices. We do not know of any careful studies quantifying these effects on modern computers, however, but we believe that they could be at least as large, if not larger than, the effects of pointer size.

Given that pointer (and scalar numeric variable) size can effect performance, must the programmer make the choice? Compilers (and optimizers) have freed programmers from many other performance-related decisions. One could imagine a compilation environment (perhaps including link-time optimization) in which the choice of pointer size was deferred until the compiler could determine the necessary range of each pointer variable. For some pointer variables, the compiler might not be able to conservatively infer the necessary size without help from the programmer (in the form of an assertion pragma); in others (for example, a program with constant-sized buffers), the inference might be fairly easy (especially in a type-safe programming language).

7. Summary and conclusions

We have shown that pointer size can affect application performance. The effects depend on pointer-use frequency, address reference patterns, memory system design, memory allocation policy, and other aspects of both program and system, but our results for real programs are consistent with what we learned from a contrived program: larger pointers put greater stress on the memory system, and can greatly affect cache-hit ratios and paging frequency.

Since some applications really do need large pointers, and the performance of other applications

File size	Number of trials	User-mode CPU time ratio	Elapsed time ratio
9589500 bytes	10	0.975	0.96
95895000 bytes	1	0.948	0.95

Ratios are mean of *32-bit pointer time/64-bit pointer time*
All output directed to `/dev/null`

Table 5-6: Performance ratios for *sort* program

does not depend on pointer size, we draw the lesson that programmers should use the ''right'' pointer size for the job. On a 64-bit system, the compiler should give the programmer a choice of pointer size, just as programmers have always had a choice of numeric variable size.

Acknowledgements

We thank Jeremy Dion, Alan Eustace, Jon Hall, Norm Jouppi, and Stefanos Sidiropoulos for their help in preparing this paper. Jim Gray and Chris Nyberg contributed to the section on sorting. Russell Kao, Louis Monier, and David Wall assisted on an earlier draft.

References

[1] Bharadwaj S. Amrutur and Mark A. Horowitz. Techniques To Reduce Power In Fast Wide Memories. In *Proceedings of the 1994 Symposium On Low Power Electronics*, pages 92-93. San Diego, October, 1994.

[2] Joel F. Bartlett. *SCHEME->C: a Portable Scheme-to-C Compiler*. WRL Research Report 89/1, Digital Equipment Corp. Western Research Lab., January, 1989.

[3] Joel F. Bartlett. *Mostly-Copying Garbage Collection Picks Up Generations and C++*. Technical Note TN-12, Digital Equipment Corp. Western Research Lab., October, 1989.

[4] C. G. Bell and W. D. Strecker. Computer structures: What have we learned from the PDP-11? In *Proc. Third Annual Symposium on Computer Architecture*, pages 1-14. Pittsburgh, PA, January, 1976.

[5] John B. Carter, Alan L. Cox, David B. Johnson, and Willy Zwaenepoel. Distributed Operating Systems Based on a Protected Global Virtual Address Space. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 75-79. IEEE Computer Society, Key Biscayne, FL, April, 1992.

[6] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319-327. San Jose, CA, October, 1994.

[7] Jeff Chase, Mike Feeley, and Hank Levy. Some Issues for Single Address Space Systems. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 150-154. IEEE Computer Society, Napa, CA, October, 1993.

[8] Daniel W. Dobberpuhl, Richard T. Witek, et al. A 200-MHz 64-bit Dual-Issue CMOS Microprocessor. *IEEE Journal of Solid-State Circuits* 27(11):1555-1567, November, 1992.

[9] Keith I. Farkas and Norman P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *Proc. 21st International Symposium on Computer Architecture*, pages 211-222. April, 1994.

[10] C. Flaig. *VLSI Mesh Routing Systems*. TR 35241/87, California Institute of Technology, May, 1987.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[12] Norman P. Jouppi, et. al. A 300Mhz 115W 32b Bipolar ECL Microprocessor. *IEEE Journal of Solid-State Circuits* 28(11):1152-1166, November, 1993.

[13] Harold Lorin. *Sorting and Sort Systems.* Addison-Wesley, Reading, MA, 1975.

[14] Steven McCanne and Chris Torek. A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling. In *Proc. Winter 1993 USENIX Conference*, pages 387-394. San Diego, CA, January, 1993.

[15] Microsoft Corporation. *Guide to Programming for the Microsoft Windows Operating System* Version 3.1 edition, Redmond, WA, 1992.

[16] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. AlphaSort: A RISC Machine Sort. In *Proc. SIGMOD '94*, pages 233-242. Minneapolis, MN, May, 1994.

[17] John Ousterhout. Corner Stitching: A Data Structuring Technique for VLSI Layout Tools. *IEEE Trans. on Computer-Aided Design* CAD-3(1):87-100, January, 1984.

[18] John K. Ousterhout, Gordon T. Hamachi, Robert N. Mayo, Walter S. Scott, and George S. Taylor. The Magic VLSI Layout System. *IEEE Design & Test of Computers* 2(1):19-30, February, 1985.

[19] Jonathan Rees and William Clinger (Editors). Revised^3 Report on the Algorithmic Language Scheme. *SIGPLAN Notices* 21(12):37-79, December, 1986.

[20] Answers to Frequently Asked Questions about SPEC Benchmarks. URL news:2vptki$kvs@inews.intel.com. June, 1994.

[21] Mark B. Reinhold. Cache Performance of Garbage-Collected Programs. In *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 206-217. Orlando, FL, June, 1994.

[22] Stefanos Sidiropoulos, Chih-Kong Ken Yang, and Mark Horowitz. A CMOS 500 Mbps/pin synchronous point to point link interface. In *1994 Symposium on VLSI Circuits Digest of Technical Papers*, pages 43-44. Honolulu, HA, June, 1994.

[23] Richard L. Sites (editor). *Alpha Architecture Reference Manual.* Digital Press, Burlington, MA, 1992.

[24] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM* 36(2):33-44, February, 1993.

[25] Amitabh Srivastava and David W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages* 1(1):1-18, March, 1993.

[26] Amitabh Srivastava and David W. Wall. Link-Time Optimization of Address Calculation on a 64-bit Architecture. In *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 49-60. Orlando, FL, June, 1994.

[27] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space. In *Proc. Summer 1993 USENIX Conference*, pages 175-186. Cincinnati, OH, June, 1993.

Jeffrey Mogul received an S.B. from the Massachusetts Institute of Technology in 1979, and his M.S. and Ph.D. degrees from Stanford University in 1980 and 1986. Since 1986, he has been a researcher at the Digital Equipment Corporation Western Research Laboratory, working on network and operating systems issues for high-performance computer systems. He is a member of ACM, Sigma Xi, ISOC, and CPSR, the author or co-author of several Internet Standards, an associate editor of *Internetworking: Research and Experience*, and was Program Committee Chair for the Winter 1994 USENIX Technical Conference.

Joel Bartlett received his B.S. and M.S. degrees from Stanford University in 1972. In 1986 he joined Digital Equipment Corporation Western Research Laboratory, working in garbage collection, Scheme, graphics, and PDA's. He is the author of Scheme->C and ezd.

Robert Mayo received his B.S. degree from Washington University in St. Louis, in 1981, and the M.S. and Ph.D. degrees from the University of California at Berkeley in 1983 and 1987, all in computer science. During 1988 he was an Assistant Professor at the University of Wisconsin, but quickly discovered there was no good chinese food in Madison. In 1989 he moved back to the bay area to join Digital Equipment Corporation's Western Research Laboratory. Dr. Mayo's interests include late code modification, computer-aided design tools for VLSI, kung pao chicken, and mongolian beef.

Amitabh Srivastava received a B.Tech. in Electrical Engineering from Indian Institute of Technology, Kanpur, and his M.S. in Computer Science from Pennsylvania State University. Since 1991, he has been a researcher at the Digital Equipment Corporation Western Research Laboratory, working in compilers and link-time code modification. He is the architect of the OM link-time technology which he used to build OM and ATOM systems. Prior to that he was researcher at the Texas Instruments Central Research Labs working on Lisp machines, compilers and object-oriented programming extensions to Scheme. He is the author of the SCOOPS system.

Address for correspondence: Digital Equipment Corporation Western Research Laboratory, 250 University Avenue, Palo Alto, California, 94301 ({mogul,bartlett,mayo,amitabh}@wrl.dec.com)