Linux/Alpha
or How to Make Your Applications Fly

David Mosberger
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
davidm@cs.arizona.edu

Abstract

This paper discusses techniques to optimize code for platforms running Linux on Alpha processors. It is based on four years of experience with the Alpha architecture and uses many real-world examples to illustrate the practicality and importance of the techniques. The primary lesson from this experience is that, for many applications, the memory system, and not the processor itself, is the primary bottleneck. For this reason, most techniques are targeted at avoiding the memory system bottleneck. Since the gap between processor and memory system speed is large, these techniques achieve performance improvements of up to 1700%. While the focus is on the Alpha architecture, many of the covered techniques are readily applicable to other RISC processors and even modern CISC CPUs.

1 Introduction

Ever since its announcement in the Fall of 1991, the Alpha [SW92] architecture has been the foundation of the world's fastest systems. In fact, except for one or two brief blips, Alpha systems have been *the* highest-performing systems based on single-CPU SPECmark performance. But with this outstanding performance record comes marketing hype and sometimes unrealistic expectations. It is not all that uncommon to find email messages or USENET articles of the form "I heard the Alpha is so fast, but now I find that my dusty deck is just 10% faster on the Alpha than on the other system." So what's the truth? The honest answer is that it depends on what you're doing. Alpha systems are without a doubt fast machines, but it is unreasonable to expect that taking a dusty deck and running it on an Alpha will result in the best possible performance. This is particularly true for programs that were written with the mind-set of the eighties, when CPU cycles were at a premium and memory bandwidth was abundant. Reality looks quite different nowadays: CPU clock-rates above 150MHz are the rule and even laptops can run at 200MHz or more. The result is that today the memory system and not the CPU is often the first-order bottleneck.

The purpose of this paper is to demonstrate a few simple techniques that help avoid the memory system bottleneck. Except for one case, the focus is on integer-intensive applications. The topic of optimizing floating-point intensive applications is certainly just as important, but unfortunately well beyond the scope of this paper.

The techniques presented can result in tremendous performance improvements. While the techniques will be helpful for all modern systems, they normally extract the biggest benefits on Alpha based machines. There are several reasons for this:

- The Alpha architecture has been designed with longevity in mind. Specifically, the Alpha architecture should be good for the next 15-25 years, which corresponds roughly to a 1000-fold increase in overall performance. For this reason, some design-tradeoffs were made in favor of long-term viability rather than for short-term benefits. For example, the Alpha was right from the start a 64 bit architecture even though at the time of its announcement, 32 bit address spaces were considered comfortably large.

- The current Alpha implementations are designed to achieve high performance by pushing clock frequency to the limit. What this means is that the CPU to memory system performance gap is the largest for Alpha based systems. For example, suppose a memory access takes 100ns. On a 500MHz Alpha CPU, this corresponds to 50 clock cycles. In contrast, on a 250MHz CPU, this is only 25 cycles. So the *relative* performance penalty of a memory access is much higher on a high-clock CPU. This may sound like a bad thing, but since the absolute performance is the same, what this really means is that a fast-clock CPU system running a memory-bound application will be about as fast as a slower-clock system but when running a memory-wise application, it will be much faster.

Before diving into the optimization techniques, this paper gives a brief overview of existing and upcoming Alpha implementations. While it is not usually necessary to optimize for a specific CPU, it is helpful to know what the characteristics of current CPUs and systems are. Section 3 presents a couple of simple performance analysis tools that are available under Linux. When porting legacy code to modern systems, such tools are invaluable since they avoid wasting time trying to optimize rarely executed code. Section 4 then presents the optimization techniques and gives a few examples of the dramatic performance improvements they can achieve. Finally, in Section 5 we present some conclusions.

2 Overview of Alpha Family

So far, the Alpha CPU family tree spans three generations: it all began with the 21064 chip. At the time of its introduction, it was the highest performing CPU and it still makes for ta nice workstation, though it's no longer competitive with the latest generation CPUs. This chip branched off into a version that was called the "Low-Cost Alpha" (LCA), also known as 21066 or 21068. The chip core was identical to the 21064 but it had an integrated memory- and PCI bus-controller. This high integration made it possible to build Alpha-based systems at relatively low cost and for the embedded market. Unfortunately, the design had a major weakness: the memory system was seriously under-powered. This created the paradox situation where a system based on this chip performed on average no better than a 100MHz Pentium, but on select problems outperformed a P6 running at 200MHz! As a result, the reaction to this chip was highly mixed and probably resulted in quite a few disappointed customers for Digital. On the other hand, there is no doubt that the low-cost at which 21066-based systems eventually were sold caused a quantum leap in the number of Linux/Alpha users.

Around June 1994, the 21164 chip was announced. It had dramatically improved performance over the 21064 and was the first, and so far only, Alpha CPU to feature a three level cache hierarchy: the first- and second-level caches were both on chip and only the third level cache was on the motherboard. This chip, in slightly improved versions, is still going strong. At the Fall 1996 Comdex in Las Vegas, such a chip, coupled with a liquid cooling system, was demonstrated running at 767MHz! Another version, called 21164PC, is scheduled to become available around Spring 1997. It omits the relatively expensive second-level on-chip cache but adds multi-media extensions and other performance-enhancing features. As the name indicates, this chip is designed to be price-competitive with PC processors; specifically the forthcoming Intel Klamath (an improved P6). While price-competitive, the 21164PC is supposed to deliver over 50% better performance than the Klamath. For this second-generation low-cost Alpha implementation, it certainly looks like Digital and its co-designer Mitsubishi are not going to repeat the mistakes of the past. The 21164PC promises to be cheap *and* fast.

If you happen to have a deep pocket or want to take a glance at what PC processors might look like in two or three years, the 21264 might be of interest. It is scheduled to become available in high-end machine during the second half of 1997. With this chip, CPU performance is expected to take another giant leap. Current estimates call for a performance level that is three to four times faster than the fastest CPUs available today.

Between each major chip generation, there are typically "half-generation" CPUs which have improvements that derive primarily from a shrink of the chip manufacturing process. For example, the 21064 chip was followed by the 21064A and similarly the 21164 was followed by the 21164A. In the former case, the core of the chip remained virtually identical to the 21064 but the primary caches doubled in size from 8KB to 16KB. In the latter case, instructions for byte and word accesses were added and the maximum clock frequency increased from 333 to 500MHz.

A summary of the performance attributes of the current Alpha chip family is presented in Table 1.

CPU	SPEC int	SPEC fp	mem [MB/s]	clock [MHz]	i-cache sz/assoc	d-cache sz/assoc	s-cache sz/assoc	TLB i/d	issue rate/dyn	ISA extension	trans [10^6]
21066	n/a	n/a	30	166	8KB/1	8KB/1	n/a	8+4/32	2/no	!R	n/a
21066A	n/a	n/a	30	233	8KB/1	8KB/1	n/a	8+4/32	2/no	!R	n/a
21064	2	2	80	175	8KB/1	8KB/1	n/a	8+4/32	2/no	!R	1.7
21064A	4	5	80	275	16KB/1	16KB/1	n/a	8+4/32	2/no	!R	2.8
21164	9	8	150	333	8KB/1	8KB/1	96KB/3	48/64	4/no	none	9.3
21164A	15	12	300	500	8KB/1	8KB/1	96KB/3	48/64	4/no	B	9.3
21164PC*	15	20	n/a	>500	16KB/1	8KB/1	n/a	48/64	4/no	M+B	3.5
21264*	40	60	1600	>600	64KB/2	64KB/2	n/a	128/128	4/yes	C+M+B	15.2

* Performance figures are estimates.

Legend:

SPEC: The approximate SPECint95 (column "int") and SPECfp95 (column "fp") results. The results are approximate since this benchmark is a system-level benchmark. This means that the benchmark results depend not only on the CPU but also on the memory system and other attributes of the machine. The SPEC results are quoted only to give a feel of what performance level can be achieved with the respective CPU. There are no SPEC95 results available for the 21066 or 21066A. The SPECint92/SPECfp92 performance for a 233MHz 21066A is around 100/112, i.e., "not very fast."

mem: The approximate memory bandwidth as reported by the McCalpin STREAM benchmark. Since this, too, is a system-level benchmark, the same caveats as for SPEC95 apply.

clock: The maximum clock rate at which the chip is available (not counting over-clocking).

i-cache: The size of the first-level instruction cache. The number behind the slash is the associativity of the cache. A number of 1 means direct mapped, 2 means 2-way set associate, and so on.

d-cache: The size and associativity of the first-level data cache.

s-cache: The size and associativity of the unified second-level on-chip cache where applicable.

TLB: The number of translation lookaside buffer (TLB) entries. The number in front of the slash is the instruction TLB size, the number behind it the size of the data TLB. 8+4 means that the TLB can hold 8 regularly sized (8KB) page table entries and 4 large entries (mapping 4MB each).

issue rate: The maximum number of instructions that can be issued per cycle. The number is followed by "no" if instructions are always issued in order and by "yes" if out-of-order issue is possible.

ISA extensions: The instruction-set architecture extensions that are supported by the chip. "!R" is an anti-extension: the first generation Alpha chips did not have hardware support for rounding towards infinity or minus-infinity. The missing functionality is emulated by the operating system, but if you do interval-arithmetic for a living, these chips probably should not be your first choice. "B" means byte/word-load/store support, "M" means multimedia support (vector minimum/maximum, pixel error, pack/unpack), and "C" means count support (population count, count trailing zeroes, move between integer and floating-point registers, and floating point square root).

trans: The number of transistors on the chip (in millions).

Table 1: Summary of Alpha Chip Family

3 Linux Performance Analysis Tools

The state of Linux performance analysis tools is in rather dire straits (this is true for freeware in general, not just Linux). Commercial products currently have an edge in this area. For example, Digital Unix comes with an excellent tool (or rather tool-generator) called ATOM [SE94]. ATOM is basically a tool that can rewrite any executable. While rewriting, it can add arbitrary instrumentation code to each function or basic-block. Digital Unix comes with a bunch of tools built with ATOM: 3rd degree (a memory-leaks and -bounds checker like the well-known `purify`) and a number of tools that give very detailed information on the performance behavior of a program (such as cache miss frequency, issue rates, and so on). At present, the freeware community can only dream of such versatile tools.

While bleak, the situation is by no means hopeless. The few tools that are available make for powerful companions when properly used. Even good old GNU `gprof` has a few features that you may not be aware of—more on this later. Let's start with the most basic performance tool: measuring time.

3.1 Accurately Measuring Time

The Unix way of measuring time is by calling `gettimeofday()`. This returns the current real time at a resolution of typically one timer tick (about 1ms on the Alpha). The advantage of this function is that it's completely portable across all Linux platforms. The disadvantage is its relatively poor resolution (1ms corresponds to 500,000 CPU cycles on a 500MHz CPU!) and, more severely, it involves a system call. A system call is relatively slow and has the tendency to mess up your memory system. E.g., the cache gets loaded with kernel code so when your program resumes execution, it sees many cache misses that it wouldn't see without the call to `gettimeofday()`. This is all right for measuring times on the order of seconds or minutes, but for finer-grained measurements, something better is needed.

Fortunately, most modern CPUs provide a register that is incremented either at the clock frequency of the CPU or an integer fraction thereof. The Alpha architecture provides the `rpcc` (read processor cycle count) instruction. It gives access to a 64 bit register that contains a 32 bit counter in the lower half of the register. This counter is incremented once every N clock cycles. All current chips use $N = 1$, so the register gets incremented at the full clock frequency (but there may be future Alpha processors where $N > 1$). The top half of the value returned by `rpcc` is operating system dependent. Linux and Digital Unix return a correction value that makes it easy to implement a cycle counter that runs only when the calling process is executing (i.e., this allows to measure the process's virtual cycle count). With `gcc`, it's very easy to write inlined functions that provide access to the cycle counters:

```
static inline u_int realcc (void) {
  u_long cc;
  /* read the 64 bit process cycle counter into variable cc: */
  asm volatile("rpcc %0" : "=r"(cc) : : "memory");
  return cc;                    /* return the lower 32 bits */
}

static inline unsigned int virtcc (void) {
  u_long cc;
  asm volatile("rpcc %0" : "=r"(cc) : : "memory");
  return (cc + (cc<<32)) >> 32; /* add process offset and count */
}
```

With this code in place, function `realcc()` returns the 32 bit real-time cycle count whereas function `virtcc()` returns the 32 bit virtual cycle count (which is like the real-time count except that it doesn't count when the process isn't running).

Calling these functions involves very small overheads: the slowdown is on the order of 1-2 cycles per call and adds only one or two instructions (which is less than the overhead for a function call). A good way of using these functions is to create an execution time histogram. For example, the function below measures individual execution times of calls to `sqrt(2.0)` and prints the results to standard output (as usual, care must be taken to ensure that the compiler doesn't optimize away the actual computation). Printing the individual execution times makes it easy to create a histogram with a little post-processing.

```
void measure_sqrt (void) {
  u_int start, stop, time[10]; int i; double x = 2.0;
  for (i = 0; i < 10; ++i) {
    start = realcc(); sqrt(x); stop = realcc();
    time[i] = stop - start;
  }
  for (i = 0; i < 10; ++i) printf(" %u", time[i]); printf("\n");
}
```

Note that the results are printed in a separate loop—this is important since `printf` is a rather big and complicated function that may even result in a system call or two. If `printf` were part of the main loop, the results would be much less reliable. A sample run of the above code might produce output like this:

```
 120 101 101 101 101 101 101 101 101 101
```

Since this output was obtained on a 333MHz Alpha, 120 cycles corresponds to 36ns and 101 cycles corresponds to 30ns. The output shows nicely how the first call is quite a bit slower since the memory system (instruction cache in particular) is cold at that point. Since the square-root function is small enough to easily fit in the first-level instruction cache, all but the first calls execute at exactly the same time.

You may wonder why the above code uses `realcc()` instead of `virtcc()`. The reason for this is simple: we want to know the results that were affected by a context switch. By using `realcc()`, a call that suffers a context switch will be much slower than any of the other calls. This makes it easy to identify and discard such unwanted outliers.

The cycle counter provides a very low-overhead method of measuring individual clock cycles. On the down side, it cannot measure very long intervals. On an Alpha chip running at 500MHz, a 32 bit cycle counter overflows after just eight and a half seconds! This is not normally a problem when making fine-grained measurements, but it is important to keep the limit in mind.

3.2 Performance Counters

The Alpha chips, like most other modern CPUs, provide a variety of performance counters. These allow measuring various event counts or rates such as the number of cache misses, instruction issue-rate, branch-mispredicts, or instruction frequency. Unfortunately, the author is not aware of any Linux API that would provide access to these counters. This is particularly unfortunate since both the Pentium and the Pentium Pro chips provide similar counters. Digital Unix gives access to these counters via the `uprofile` and `kprofile` programs and an `ioctl`-based interface documented in the `pfm(7)` man page. Hopefully, something similar (but more general) will eventually become available for Linux. With the proper tools, these counters can provide a wealth of information.

3.3 GNU `gprof`

Most readers are probably familiar with the original `gprof` [GKM82]. It's a handy tool to determine the primary performance bottlenecks at the function level. However, with the help of `gcc`, GNU `gprof` can also look *inside* a function. We illustrate this with a truly trivial function that computes the factorial. Assume we've typed up the factorial function and a simple test program in file `fact.c`. We can then compile that program like this (assuming GNU `libc` version 2.0 or later is installed):

```
gcc -g -O -a fact.c -lc
```

Invoking the resulting `a.out` binary once produces a `gmon.out` file that contains the execution counts for each basic block in the program. We can look at these counts by invoking `gprof` with options "`-l --annotate`". This generates a source code listing that shows how many times a basic block in each line of source code has been executed.

Our factorial example results in the listing shown in Figure 1 and here is how to interpret it: the basic-block starting at the `printf` line in function `main()` was executed once, so it has been annotated with a `1`. For the factorial

```
*** File fact.c:
          20 -> static unsigned long fact (unsigned long n) {
                    if (n > 1) {
          19 ->         return n * fact(n - 1);
                    } else {
           1 ->         return 1;
                    }
          20 -> }

                int main (int argc, char **argv) {
           1 ->     printf("fact(20)=%lu\n", fact(20));
                    return 0;
                }


Top 10 Lines:

     Line      Count

        1         20
        7         20
        3         19
        5          1
       10          1

Execution Summary:

        5   Executable lines in this file
        5   Lines executed
   100.00   Percent of the file executed

       61   Total number of line executions
    12.20   Average executions per line
```

Figure 1: Basic-block execution counts

function, the function prologue and epilogue were executed 20 times each, so the first and last line of function `fact` are annotated with 20. Of these 20 calls, 19 resulted in a recursive call to `fact` and the remaining one simply returned `1`. Correspondingly, the `then` branch of the `if`-statement has been annotated with 19 whereas the `else`-statement has an annotation of 1. It's that simple.

There certainly are no surprises in the behavior of function `fact()`, but in realistic, more complicated functions or in code that was written by somebody else, this knowledge can be very helpful to avoid wasting time optimizing rarely executed code.

4 Optimization Techniques: Making Your Applications Fly

Finally we're at the point where we're ready to look into the techniques that can greatly improve the performance of a given piece of code. Most of them are not novel. Some of them have been around for so long that it would be difficult, if not impossible, to give proper credit. Others are "obvious" (once you know them...). But the key point here is that it is the characteristics of modern, and in particular Alpha-based, systems which make these techniques so important and worthwhile.

While the focus of this section is on the Alpha architecture, we would also like to give an idea of how the same techniques work on other architectures. The tricky part is that we want to do this without igniting a war over which CPU architecture is the "best" or the "fastest." Such terms are, to a good degree, meaningless since they can be applied usefully only relative to a given problem. For this reason, performance results are presented as follows: for the Alpha we present both absolute and relative results. The absolute numbers are useful to give a concrete feel for how fast the code is. The relative results (i.e., speed ups) is what tells us how well a given technique works. Where meaningful, we also list the speed up (but not the absolute performance) achieved on a Pentium Pro based system. Only listing speed up for the Pentium Pro case makes it impossible to tell which system was faster on a given problem while still allowing us to compare the relative benefits. (To avoid any misconception: this arrangement was *not* chosen because the Alpha performed poorly; suffice it to say that the author has been using Alphas for some time now and is generally pleased with the performance level they achieve.)

Since an architecture per se doesn't perform at all, let us be a bit more specific about the systems used for the measurements:

Alpha system: The Alpha system was an AlphaStation 600 5/333 (aka Alcor). It has a 333MHz 21164 processor with 4MB of third-level cache and 64MB of main-memory. While a nice (and very expensive) box, it is by no means the latest and greatest of the available Alpha systems. At the time of this writing, much faster and much cheaper 500MHz systems have been around for a while already.

Pentium Pro system: The x86 system was a Gateway 2000 with a Pentium Pro processor with 32MB of main-memory and 256KB of second-level cache. For clarity, this system is referred to as "P6" during the remainder of this section.

Both the Alpha and the P6 systems were running Red Hat 4.0 with kernel version 2.0.18. The compiler used was `gcc` version 2.7.2. On the Alpha, option `-O2` was used (with this version of `gcc`, using an option setting of `-O3` or higher generally results in *slower* code.). On the P6, options `-O6` and `-m486` were used.

It is also illustrative to compare `gcc` to commercial Alpha compilers, such as Digital's GEM C compiler. The GEM C compiler usually generates somewhat better code but every once in a while it creates code much faster than `gcc`'s (this usually happens on floating point intensive code). For this reason, some of the measurements also include the results obtained with GEM C. This compiler was invoked as `cc -migrate -O4 -tune ev5 -std1 -non_shared`. It's not clear what the version of the compiler version was—it was whatever came with Digital Unix version 3.2.

4.1 Avoiding Integer Division

The Alpha architecture does not provide an integer division instruction. The rationale for this is that (a) such operations are relatively rare and (b) division is fundamentally of an iterative nature so implementing it in hardware is not all that much faster than a good software implementation [Alv91]. Nevertheless, there are important routines that depend on integer division. Hash-tables are a good example: computing a hash-table index typically involves dividing by an integer prime constant.

There are basically two ways to avoid integer division. Either, the integer division is replaced by a floating point division or, if the division is by a constant, it is possible to replace the division by a multiplication with a constant, a shift, and a correction by one (which isn't always necessary). Floating point division may sound like a bad idea, but the Alpha has a very fast floating point unit and since a 32 bit integer easily fits into a `double` without loss of precision, it works surprisingly well. Replacing a division by a constant with a multiplication by the inverse is certainly faster, though it's also a bit tricky since care must be taken that the result is always accurate (off-by-one errors are particularly common). Fortunately, for compile time constants, `gcc` takes care of this all by itself.

To illustrate the effect this can have, we measured how long it takes to lookup all symbols in the standard C library (`libc.so`) using the ELF hash-table lookup algorithm (which involves one integer division by a prime constant). With integer division, roughly 1.2 million lookups per second can be performed. Using a `double` multiplication by the inverse of the divisor instead brings this number up to 1.95 million lookups per second (62% improvement). Using integer multiplication instead gives the best performance of 2.05 million lookups per second (70% improvement). Since the performance difference between the double and the integer multiply-by-inverse version isn't all that big, it's usually better to use the floating pointer version. This works perfectly well as long as the operands fit in 52 bits and avoids having to worry about off-by-one errors.

4.2 Keeping the Translation Lookaside Buffer in Mind

Two parts in modern CPUs that are easy to forget about are the data and instruction translation lookaside buffers (TLBs). The TLBs hold the most recently accessed page table entries. TLBs are necessary since it would be far too slow to access the page tables on every memory access. Since each TLB entry maps an entire page (8KB with the current Alpha chips), the TLB is usually not the limiting factor to performance. The catch is that when a program does suffer from excessive TLB misses, performance will go down the drain fast. In such a case, the slow down may easily be big enough that it is worthwhile to switch to a completely different (normally slower) algorithm that has a better TLB behavior. For example, on the Alpha system reading one word from each page in an array of 63 pages (504KB) takes about 15ns per access. But doing the same in an array of 64 pages takes 65ns per access—a slowdown of more than a factor of four! Since the second-level cache in that system is 4MB large, this jump in access time is purely due to the data TLB.[1]

As said above, the TLB is not usually a first-order bottleneck but a small experiment did show that a hash-table that is 50% full and exceeds the data TLB size is no faster than a more compact search tree that needs two memory accesses per lookup (the hash-table needs just one memory access but since it exceeds the TLB size, that one access is slow). In general, the TLB may be the primary bottleneck when large data sets are accessed more or less randomly and sparsely.

4.3 Avoiding Memory Accesses

On modern systems, memory accesses are bad and computation is good. In the ideal case we would like to completely replace memory accesses by computation. This obviously is not always possible; but where it is possible, the payoff can be big.

For example, let us consider the problem of reversing all the bits in each byte in a long. A byte is reversed as follows: bit 0 is swapped with bit 7, bit 1 with bit 6 and so on. Since we want to reverse all the bytes in a long, this algorithm is applied once for each byte in the long.

Why would anyone want to do this? As you may know, both the Alpha and the x86 architecture are little-endian, but when IBM designed the VGA graphics adapter, it chose to use a big-endian bit order for the pixels in the graphics memory. That is, bit 7 in a byte corresponds to the left-most pixel and bit 0 to the rightmost. This is backwards since the coordinate value for the left-most pixel is smaller. So, byte-reversal is indeed a relatively important operation for VGA X servers.

The traditional way of implementing byte-reversal might look something like the code below. To conserve space, we show the code to reverse a 4-byte integer only. The 8-byte long case is a straight-forward extension of this. The code assumes that array `byte_reversed` has been initialized such that `byte_reversed[i]` is the reversed value of `i`.

```
u_int byterev_naive (u_int L) {
  /* this is for sizeof(u_int)==4 */
  return (((u_int)byte_reversed[(L >>  0) & 0xff] <<  0) |
          ((u_int)byte_reversed[(L >>  8) & 0xff] <<  8) |
          ((u_int)byte_reversed[(L >> 16) & 0xff] << 16) |
          ((u_int)byte_reversed[(L >> 24) & 0xff] << 24));
}
```

With this naive algorithm, each byte-reversal involves a table-lookup. Since the Alpha is a 64 bit architecture, this means that each long reversal involves eight memory accesses. How could we avoid these expensive memory accesses? Well, a simple and arguably more intuitive way to implement byte reversal is to use shifting and masking to swap the bits. The code that does this would look something like this:

```
u_long byterev_long (u_long L) {
  u_long mask = 0x0101010101010101;
  return (((L & (mask << 0)) << 7) | ((L >> 7) & (mask << 0)) |
          ((L & (mask << 1)) << 5) | ((L >> 5) & (mask << 1)) |
          ((L & (mask << 2)) << 3) | ((L >> 3) & (mask << 2)) |
          ((L & (mask << 3)) << 1) | ((L >> 1) & (mask << 3)));
}
```

[1] The careful reader will notice that the data TLB on the 21164 is 64 entries large; the jump in access time occurs when going from 63 to 64 pages because one TLB entry is needed to access the virtually mapped page table.

Note that, except for the initialization of `mask`, this code is completely independent of the width of a long! So, to make this work on a 32 bit machine, all we need to do is initialize `mask` with `0x01010101` instead.

Now let's see how the implementations compare. The table below gives the results for the Alpha, the Pentium Pro (P6), and a 120MHz Pentium Notebook (P5). In addition to the two implementations shown above, the table also includes a row that shows the performance of the naive implementation when coded in x86 assembly code (implementation `byterev_x86`). This routine comes straight from the XFree86 v3.2 distribution.

	Alpha		P6	P5
Implementation	abs	rel	rel	rel
`byterev_naive`	55MB/s	1.00	1.00	1.00
`byterev_long`	72MB/s	1.31	1.13	1.17
`byterev_x86`	n/a	n/a	0.56	1.51

As the table shows, the byte-reversal routine that avoids memory accesses is over 30% faster on the Alpha. Interestingly, on the P6 this routine is also fastest. The benefit there is less ("only" 13%), but given that it's more portable and more intuitive, there is no reason not to use it. What's stunning is the complete failure of the assembly version on the P6. That routine is only half as fast as the version that avoids memory accesses. This may be due the assembly routine's extensive use of byte accesses to CPU registers. For the Pentium, as shown in the P5 column, the assembly coded routine is the fastest version. But don't be misled by the relative performance numbers: in terms of absolute performance, the Pentium is just half as fast as the P6.

To summarize, not only is `byterev_long()` by far the fastest version on the Alpha, but even on a P6 it appears like the right solution.

4.4 Lending `gcc` a Hand: Separating Loads/Stores from Computation

One reason why `gcc` sometimes generates significantly worse code than Digital's GEM compiler is that it does not perform inter-procedural alias analysis. What this means is that `gcc`'s alias analysis is sometimes unnecessarily conservative. To illustrate the problem, consider the function below. It is a simple unrolled loop that reverses all the bytes in array `src` and stores the result in array `dst`.

```
void vecrev_naive (long *src, long *dst, long size)
{
    long i;
    for (i = 0; i < size/sizeof(long); i += 4) {
        dst[i + 0] = byterev_long(src[i + 0]);
        dst[i + 1] = byterev_long(src[i + 1]);
        dst[i + 2] = byterev_long(src[i + 2]);
        dst[i + 3] = byterev_long(src[i + 3]);
    }
}
```

The problem with this code is that the C compiler has no way of knowing how the `src` and `dst` pointer relate to each other. For all it knows, `dst` could point to the second element of the array pointed to by `src`. When this happens, the two pointers refer to overlapping regions of memory and they are said to alias each other. For the compiler, it is important to know whether the two pointers overlap since that determines the degree of freedom it has in reordering instructions. For example, if the regions overlap, then storing a value to `dst[i+0]` may affect the value of `src[i+1]`. Thus, to be on the safe side, the compiler must generate the loads and stores in the above function strictly in the order in which they occur in the source code.

Now, if it is known that the two arrays passed to the function never alias each other, then we can lend `gcc` a hand by explicitly telling it so. We can do this by first reading all the values from the memory, then doing all the computation and finally storing the results back to memory. Thus, the above code would be transformed into the code shown below.

```
void vecrev_sep (long *src, long *dst, long size) {
  unsigned long L0, L1, L2, L3; long i;
  for (i = 0; i < size/sizeof(long); i += 4) {
    /* loads: */
    L0 = src[i + 0]; L1 = src[i + 1]; L2 = src[i + 2]; L3 = src[i + 3];

    /* computation: */
    L0 = byterev_long(L0); L1 = byterev_long(L1);
    L2 = byterev_long(L2); L3 = byterev_long(L3);

    /* stores: */
    dst[i + 0] = L0; dst[i + 1] = L1; dst[i + 2] = L2; dst[i + 3] = L3;
  }
}
```

Since the stores all happen at the end, `gcc` knows right away that none of the stores may affect any of the preceding loads. This provides it complete freedom in generating the best possible code for the loads and computation (the assumption here is that `byterev_long` is an inlined function).

On the Alpha and most other architectures with lots of registers (e.g., most RISCs), this kind of code never (or at least very rarely) hurts performance and usually improves performance for `gcc`. Unfortunately, the same isn't true for the x86 architecture. The problem there is that only few registers are available. So the code that's better for RISCs is usually worse for the x86 due to additional stores and loads that are necessary to access the temporaries that ended up on the stack. To witness, performance numbers for the two versions are given below:

	Alpha		P6
Implementation	abs	rel	rel
`vecrev_naive`	82MB/s	1.00	1.00
`vecrev_sep`	93MB/s	1.13	0.73

As the table shows, merely by separating loads from computation from stores gains 13% with `gcc` on the Alpha. The same code looses 17% on the P6, in contrast. This illustrates that while most of the techniques in this paper apply to both architectures, there are important differences that sometimes necessitate different coding to achieve the best performance.

4.5 Improving the Memory Access Pattern

By now you may be wondering where the promised order of magnitude improvements are. Let's consider a simple problem: matrix multiplication. While simple, it is typical of a problem that is considered floating-point intensive. In reality, most floating-point intensive problems are also very memory intensive. E.g., they process large vectors or matrices. Thus, the memory access pattern often plays a crucial role in achieving the best possible performance. The textbook implementation of matrix multiplication looks roughly like this:

```
void matmul0 (int dim, float *a, float *b, float *c) {
  int i, j, k; float dot;

  for (i = 0; i < dim; ++i)
    for (j = 0; j < dim; ++j) {
      dot = 0.0;
      for (k = 0; k < dim; ++k)
        dot += a[i*dim + k] * b[k*dim + j];
      c[i*dim + j] = dot;
    }
}
```

Here, the matrix pointed to by a gets multiplied by the matrix pointed to by b and the result is stored in c. The matrix dimension is passed in argument dim. On the Alpha, a 512×512 matrix multiply with this routine executes at about thirteen million floating-point operations per second (MFlops). This is not too shabby, but let's see whether we can squeeze more out of the machine. Having learned our lesson in performance optimization, we might try to unroll the inner loop and avoid all multiplications due to indexing. This does indeed result in a faster version: now, matrix multiply executes at about 15MFlops.

Rather than declaring the problem solved, let's think about the memory access pattern for a minute. Each element in the result matrix c is a dot product of a row in a and a column in b. For example, the element c[0][0] is computed as the dot product of the first row in a and the first column in b. This is illustrated in Figure 2. In our naive matrix multiply routine, this means that the accesses to a causes a nice, dense, linear memory access pattern. Unfortunately, things do not look quite as good for b. There the memory access pattern is sparse: first, the element at offset 0 is read, then the one at offset dim and so on. Such sparse access patterns are bad for many reasons. Suffice it to say that it's easiest to optimize a machine for dense, linear accesses, so it is likely that those accesses will always be the fastest ones. Fortunately, there is a simple trick that avoids the bad access pattern for matrix b: before doing the actual matrix multiply, we can simply transpose matrix b. Then, all memory accesses are dense. Of course, transposing b causes extra work, but since that matrix is accessed dim times, this may well be worth the trouble.

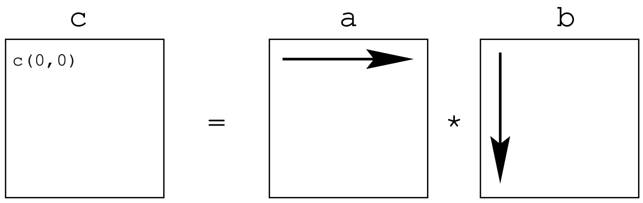

Figure 2: Matrix Multiply

So, let's change matmul0 into matmul1 by adding a matrix transposition in front of the main loop. The code below assumes that tb is an appropriately sized temporary variable to hold the transposition of b.

```
void matmul1 (int dim, float *a, float *b, float *c)
{
  int i, j, k; float dot;

  /* transpose b: */
  for (i = 0; i < dim; ++i)
    for (j = 0; j < dim; ++j)
      tb[i*dim + j] = b[j*dim + i];

  ...rest like matmul0, except that b is replaced by tb...
}
```

If you thought 15MFlops is fast, think again: matmul1 executes at a blazing 45Mflops! Are we done yet? How about we try some loop unrolling, etc.? For matmul0, this bought us 25%, which isn't bad at all. If we unroll the loop eight times and do some other straight forward optimizations, the code may look like the one shown below. For compactness, it assumes that dim is an integer multiple of eight.

```
void matmul2 (int dim, float *a, float *b, float *c)
{
  float dot0, dot1, dot2, dot3, a0, a1, a2, a3, b0, b1, b2, b3;
  float dot4, dot5, dot6, dot7, a4, a5, a6, a7, b4, b5, b6, b7;
  long I, J, i, j, k;

  ...transpose b into tb like in matmul1...;
```

```
    for (I = 0; I < dim*dim; I += dim)
      for (j = J = 0; j < dim; ++j, J += dim) {
        dot0 = dot1 = dot2 = dot3 = dot4 = dot5 = dot6 = dot7 = 0.0;
        for (k = 0; k < dim; k += 8) {
          a0 =  a[I + k + 0]; b0 = tb[J + k + 0];
          a1 =  a[I + k + 1]; b1 = tb[J + k + 1];
          a2 =  a[I + k + 2]; b2 = tb[J + k + 2];
          a3 =  a[I + k + 3]; b3 = tb[J + k + 3];
          a4 =  a[I + k + 4]; b4 = tb[J + k + 4];
          a5 =  a[I + k + 5]; b5 = tb[J + k + 5];
          a6 =  a[I + k + 6]; b6 = tb[J + k + 6];
          a7 =  a[I + k + 7]; b7 = tb[J + k + 7];
          dot0 += a0 * b0; dot1 += a1 * b1; dot2 += a2 * b2; dot3 += a3 * b3;
          dot4 += a4 * b4; dot5 += a5 * b5; dot6 += a6 * b6; dot7 += a7 * b7;
        }
        c[I + j] = dot0 + dot1 + dot2 + dot3 + dot4 + dot5 + dot6 + dot7;
      }
  }
```

Was it worth the trouble? By all means yes: `matmul2` clocks at a full 80MFlops! Whether you like this kind of code or not may be a matter of taste, but it certainly is fast!

The table below presents a summary of the performance results. For comparison, it also includes the results obtained when compiling the same code with Digital's GEM C compiler and the relative results for the P6.

	Alpha				P6
	gcc		GEM C		
Implementation	abs	rel	abs	rel	rel
matmul0	11MFlops	1.00	13MFlops	1.00	1.00
matmul1	47MFlops	4.27	66MFlops	5.08	2.53
matmul2	80MFlops	7.27	105MFlops	8.08	3.97

As the table shows, with `gcc` we achieved a performance improvement by over a factor of seven. More importantly, the optimizations paid off in all three cases. In fact, with Digital's compiler, the improvement was more than a factor of eight. The performance gap between Digital C and `gcc` is rather large. For integer code, the gap is usually smaller, but it certainly looks as if `gcc` needs some work in the floating point area. Finally, notice that even on the P6 performance increased by almost a factor of four. This is encouraging since unrolling the loop eight times is a tad aggressive for the x86 architecture (since relatively few registers are available). Presumably, the code could be sped up some more, but the point here is that all three cases benefit from memory access optimizations in the same way.

4.6 Data-parallel Processing: MPEG Core Loop

Multimedia applications are the rage these days. All mainstream CPU architectures (with the notable exception of the PowerPC) have so-called multimedia extensions. The Alpha is ideally suited for such applications since it has been a 64 bit architecture right from the start. In fact, the Alpha multimedia extension is completely trivial: it adds only four new instruction types (vector minimum/maximum, pixel error, pack and unpack). Since some of these instructions can operate on different data types (byte or word; signed or unsigned), the total number of instructions added is 13, which is much smaller than the corresponding number for other architectures. The reason the Alpha got away with so few additions is because its original instruction set already contains many of the instructions needed for multimedia applications. For example, there is an instruction that allows eight bytes to be compared in parallel—a seemingly simple instruction that can prove surprisingly powerful in a number of applications.

We illustrate this using `mpeg_play`, the Berkeley MPEG decoder.[2] Since there is not enough space to illustrate all

[2] `mpeg_play` is available at `http://sunsite.unc.edu/pub/Linux/apps/graphics/viewers/` as `mpeg_play-2.3-src.tar.gz`.

the optimizations that can be applied to this program, we focus on one of the most important operations in the MPEG decoder. This operation involves computing the average of two byte vectors. This is a frequent operation since video often contains images that can be represented as the average of an earlier and a later image. The straight-forward way of averaging a byte vector is shown below:

```
void byte_avg0 (u_long size, u_char *a, u_char *b, u_char *c) {
  int i;
  for (i = 0; i < size; ++i) c[i] = (a[i] + b[i]) / 2;
}
```

This loop executes at about 94ns per byte-average (iteration) when compiled with `gcc`. Unrolling this loop twice and reading ahead the input values needed in the next iteration yields code that is probably close to optimal with this byte-oriented approach. Indeed, with `gcc`, performance increases to about 60ns per byte-average (let's call this unrolled version of the function `byte_avg1`).

To get even higher performance, we need to be a bit more aggressive. Considering that the Alpha is a 64 bit architecture, it sure would be nice if we could calculate the average of eight bytes in parallel. Reformulating byte-oriented algorithms in such a data parallel format is often trivial (see Section 4.3). For byte-averaging, it's not quite as simple: the straight forward implementation requires nine bits of precision, since $255+255=510$. But if we pack 8 bytes into a 64 bit word, there is no extra ninth bit. How can we get around this? Obviously, we can divide the operands by two *before* adding them. That way, the sum is at most $127+127=254$ which conveniently fits into 8 bits. But the catch is that the result may be wrong: if both operands are odd, it will be one too small. Fortunately, it's easy to correct for this: if bit 0 in both operands is set, a correction by one is necessary. In other words, we can make space for that extra ninth bit by using an additional long register that is used to hold the correction bits. Since all intermediary results now fit into 8 bits, the obstacles to a data-parallel implementation of byte-averaging have been removed. The resulting code is shown below. For simplicity, it assumes that the input vectors are long aligned and have a size that is an integer multiple of the size of a long:

```
#define CAT(v,x)  ((u_long)(v)<<8 | (x))
#define VEC(x)    CAT(CAT(CAT(CAT(CAT(CAT(CAT(x,x),x),x),x),x),x),x)

void byte_avg2 (u_long size, u_char *ca, u_char *cb, u_char *cc) {
  u_long *a = (u_long*) ca; u_long *b = (u_long*) cb; u_long *c = (u_long*) cc;
  u_long A0 = a[0], B0 = b[0], A1, B1, CC, i, count = size / sizeof(u_long);
  for (i = 1; i < count; ) {
    A1 = a[i]; B1 = b[i]; ++i;  /* read ahead */
    CC = (A0 & B0) & VEC(0x01);
    A0 = (A0 >> 1) & VEC(0x7f); B0 = (B0 >> 1) & VEC(0x7f);
    c[i - 2] = A0 + B0 + CC;

    A0 = a[i]; B0 = b[i]; ++i;  /* read ahead */
    CC = (A1 & B1) & VEC(0x01);
    A1 = (A1 >> 1) & VEC(0x7f); B1 = (B1 >> 1) & VEC(0x7f);
    c[i - 2] = A1 + B1 + CC;
  }
}
```

Note that macro `VEC()` takes an eight bit value and replicates it once for each byte in a long—it's much more convenient and less error-prone to write `VEC(0x01)` instead of `0x0101010101010101`. Maybe it's helpful to explain the core of the averaging a bit: variable `CC` holds the correction bits, so it's simply the bitwise AND of vectors `A0` and `B0`, masked with a vector of `0x01`. We divide `A0` and `B0` by two by shifting them to the right by one position and masking the resulting long with a vector of `0x7f`. This masking is necessary since otherwise bit 0 of the byte "above" a byte would sneak in and become bit 7 of that byte, which would cause gross errors. The average is computed by simply adding the vectors `A0`, `B0`, and `CC`. This addition does not cause any overflows since, per byte, the largest possible value is $127+127+1=255$.

Despite its look, this code is actually very portable. For a 32-bit architecture, all that needs to change is macro `VEC` (and even that is necessary only to get rid of compiler warnings). Byte-order is not an issue since even though the data is accessed a long at a time, each byte is still processed individually. This is all nice and fun, but lets not forget the real reason why we tried this: performance. This data parallel version of the byte-averaging loop runs at 5.3ns per byte-average—more than an order of magnitude faster than the unrolled loop!

A summary of the three averaging routines is given in the table below. The relative performance is in terms of throughput (number of byte-averages per second) since that's both more intuitive and more impressive. Results for the Alpha are presented both for `gcc` and Digital's GEM C compiler; as usual, for the P6, `gcc` was used.

	Alpha				P6
	`gcc`		GEM C		
Implementation	abs	rel	abs	rel	rel
`byte_avg0`	94ns/avg	1.00	69ns/avg	1.00	1.00
`byte_avg1`	60ns/avg	1.57	56ns/avg	1.23	0.82
`byte_avg2`	5ns/avg	18.80	4ns/avg	17.25	1.93

Note how GEM C generates much better code for the stupid version (`byte_avg0`) but just slightly better code for the clever version (`byte_avg2`). This is a common theme (at least for integer code): for well-structured code, `gcc` usually generates code that is on par with the GEM C compiler. The other interesting result is that the read ahead and loop-unrolling hurt on the P6. This means that `byte_avg2` probably could be optimized more for the P6 (since it uses read ahead and loop-unrolling, too), but even so the P6 is twice as fast with the data-parallel version. This is impressive since the relative overheads are much higher for a 32 bit chip (the Alpha can amortize all the masking and shifting over eight bytes, whereas a 32 bit architecture has only four bytes).

But how does all this affect performance of `mpeg_play`? This is best illustrated by comparing the original Berkeley version with the one optimized using the techniques described in this section (particularly data-parallel processing and avoiding integer divisions).[3] Comparing MPEG performance is a bit tricky since a large fraction of the time is spent displaying images. This can be factored out by using option "`-dither none`" which has the effect that nothing at all gets displayed (while the MPEG stream is still decoded as usual). The table below shows the result for this mode as well as when using an ordered dither. The ordered dither itself was also optimized using data-parallel processing, which resulted in a version called ordered4. The exact set of options used for `mpeg_play` was "`-controls none -framerate 0 -dither` *D*". *D* was either `none`, `ordered`, or `ordered4` as indicated in column "Dither" below. The movie that was used in these measurements was a 320×240 pixel sized computer animation called `RedsNightmare.mpg`.[4]

Version	Dither	Framerate
Original	`none`	45.3 frames/sec (1.00)
Optimized	`none`	139.8 frames/sec (3.09)
Original	`ordered`	26.9 frames/sec (1.00)
Optimized	`ordered`	58.9 frames/sec (2.19)
Optimized	`ordered4`	98.2 frames/sec (3.65)

As the table shows, the optimization techniques do indeed result in tremendous performance improvements even at the application level. Of course, few people would enjoy watching a movie at 98 frames per second, but what this means is that with the optimized code, you can either watch much larger videos in real-time, or you could have CNN on while watching your favorite movie. Who needs picture-in-picture capability when we've got real windowing systems?

[3] The full source to the Alpha optimized version is available at `ftp://ftp.azstarnet.com/pub/linux/axp/misc/` as `axpsrc-mpeg_play-2.3axp-2.tar.gz`

[4] The movie `RedsNightmare.mpg` is available at many ftp sites. E.g., `http://sunsite.unc.edu/pub/multimedia/animation/mpeg/berkeley-mirror/movies/`.

5 Conclusions

The Alpha architecture is designed for performance and its implementations do indeed make for very fast systems. Since its chips run at very high clock frequencies, the Alpha usually benefits the most from simple techniques that improve the memory system behavior of a given program or algorithm. A few of these techniques have been demonstrated in this paper and shown to achieve performance improvements anywhere in the range from 10% to 1700%. Fortunately, the same techniques also seem to benefit the other CPU architectures. This is good news since it means that usually one optimized implementation will perform well across a broad range of CPUs.

The biggest hurdle to developing high-performance applications under Linux is the current lack of sophisticated performance analysis tools. Section 3 demonstrated two simple tools and alluded to what hardware capabilities might be exploited in the future. The relative lack of such tools is not surprising: while most commercial Unix vendors have tools for their own architecture, few, if any, are multi-platform. To some degree this is inherent to the problem, but there is no question that it would not be very difficult to create better yet portable performance analysis tools.

Linux is what makes low-cost Alpha-based Unix workstations a reality. While Digital Unix currently comes with better compilers, runtime-libraries and more tools for the Alpha, the price difference is such that one can easily make up for the performance difference by spending a little more money on a faster machine. Also, development of `gcc` and better libraries doesn't stand still. However, since most work is done on a voluntary basis, it does take some time. Even so, Linux is already a highly competitive platform for integer intensive applications. For floating-point intensive and especially FORTRAN applications things are not as mature yet. Fortunately, if one cannot afford to wait for a better compiler, there is always the option of purchasing one of the commercial FORTRAN compilers available for Linux/Alpha.

Acknowledgments

The author would like to thank Richard Henderson of Texas A&M University and Erik Troan of Red Hat Software for reviewing this paper on short notice. Their feedback greatly improved its quality. Errors and omissions are the sole responsibility of the author.

References

[Alv91] Robert Alverson. Integer division using reciprocals. In *10th Symposium on Computer Arithmetic*, pages 186–190. IEEE, 1991. Available at `http://www.tera.com/tera/ftp.html`.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*, number 17, pages 120–126, June 1982.

[SE94] Amitabh Sristava and Alan Eustace. ATOM: A system for building customizable program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, number 6, pages 196–205, June 1994.

[SW92] Richard L. Sites and Richard T. Witek, editors. *Alpha Architecture Reference Manual*. Digital Press, Burlington, Massachusetts, 1992. Available at `http://ftp.digital.com/pub/Digital/-info/semiconductor/literature/dsc-library.html`.