Performance tips for Alpha C programmers

Paul J. Drongowski
Hewlett Packard Corporation

30 August 2001

1. Introduction

This technical note describes some considerations and techniques to take into account when building computationally intensive applications for Tru64 UNIX and Linux on Alpha. It provides a little background on the Alpha architecture and goes on to describe compiler options and coding practices that will help you to get the most from Alpha. We will use a small example program to illustrate a few of these practices.

On Alpha, programmers are lucky because there are two C compilers: the GNU (EGCS) C compiler and the Compaq C compiler. Both the GNU C and Compaq C compilers provide options to help produce tuned code for Alpha and this note provides tips for both compilers.

Why should you use the Compaq C compiler for your application instead of GNU C? Both compilers are mature and produce correct code. The GNU C compiler is designed to produce good code for a wide variety of machine architectures and it accomplishes this goal quite successfully. The Compaq C compiler, on the other hand, is the product of intensive R&D to generate insanely great code for the Alpha architecture in particular. The RISC (Reduced Instruction Set Computer) strategy calls for architecture, compiler and operating system to work together to obtain high levels of performance. The Compaq C compiler is your key to performance, and here's the first tip.

> **Tip #1** Use the Compaq C compiler. It has been specially tuned for Alpha.

2. Architectural considerations

2.1 A little history

The most prevalent, and commercially successful, computer architecture is attributed to John von Neumann. In the simplest case, von Neumann's architecture consists of a processing unit and a memory unit, which communicate with each other through a bus (or "channel.")

> Processor <- - -> Memory

Both instructions and data (programs) are stored in memory. The processor fetches instructions from memory, interprets the instructions and possibly reads data from or writes data to memory. The processor specifies a particular data item in memory by sending the address of the item to the memory unit.

Well, most of us know this through either formal education or osmosis. But, what may not be obvious is that the processor-bus-memory architecture has been a considerable source of "tension" and engineering design trade-offs ever since von

Neumann's time. Users demand both higher processor speed (more operations per second) and memory capacity to accommodate bigger data sets and programs. Unfortunately, access speed must usually be sacrificed in favor of capacity when building large memory units and thus, processor speed has far outstripped the ability of big memory units to provide instructions and data.

Computer designers have resolved this dilemma by providing virtual memory and a physical memory hierarchy consisting of two or more levels. With *Virtual memory*, each program executes in its own logical address space. Logical addresses are mapped to physical memory locations by a combination of hardware and operating system software. The data items are located in blocks of physical memory called "pages" (or sometimes "segments.") Only the current working subset of pages in kept in memory. Thus, the logical address space may be very much bigger than the physical memory space.

Memory hierarchy takes advantage of different memory technologies to satisfy users' demand for big memory at a reasonable cost. Memory units are arranged and interconnected so that the fastest, but smallest units are close to the processor and the slowest, but largest units are far away from the processor:

Processor registers <- - - Cache memory <- - - Paged primary memory <- - - Secondary storage

Data and instructions are brought closer to the processor when they are needed. For example, unused pages are kept on secondary storage (disk) until "demanded" by the program through a page fault. The operating system handles the page fault for the program by reading the page data into primary memory, updating the logical to physical address mapping information, and restarting the program from the point of the page fault. Through virtual memory and hierarchy, disk and limited primary memory can be used to simulate a very large, logical primary memory structure. The low cost per byte of the disk makes this approach economically attractive.

Cache memory bears a similar relationship to primary memory as primary memory has to secondary storage. However, the time pressure is even more critical. Time is so critical that the processor hardware handles the caching of primary memory data -- there's no time here for software intervention. The memory blocks are also much smaller, usually 64 to 128 bytes of data instead of 4,096 to 8,192 bytes of page data. Cache block size is often limited by the number of bytes that can be moved in a few processor cycles on a limited width, parallel bus.

The viability of this whole scheme depends upon the assumption that data (or instructions) will be immediately available to the processor when the processor needs it. If the data is not available, then the processor must wait until the needed information is brought forward through the memory hierarchy. If the data is far away (two or more levels), then the wait will be a long one. In the case of a page fault, the processor can be kept busy by switching to another ready-to-run program. This is *not* an option when data is missing from cache memory.

Computer and operating system designers improve the odds that data will be available by exploiting predictable or statistically likely patterns of access to data and instructions. Locality of reference is the *most* important principle guiding design decisions. The principle of locality states that the next most likely data item or instruction to be needed is closest to the current one. For example, the next block in a file is most probably the one that will be needed next. The operating system, being clever, can read the next block before it is needed and have it on hand when the actual read request is issued.

Efficient cache memory operation depends on locality, too. When a read misses in the cache memory, the cache memory controller brings in the missing data item plus the next several items under the assumption that those items will be needed, too. When fetching instructions, this behavior caters to programs with long sequential runs of instructions whose flow is not broken by (taken) branch instructions. This cache behavior also caters to programs that read array data sequentially without jumping around "randomly" in the address space.

2.2 Alpha architecture and its implementations

Alpha is a 64-bit reduced instruction set computer architecture. The architects envisioned a long life for Alpha (15-25 years) that would span many design innovations and changes in technology. The architecture needed to accommodate:

- Fast cycle times,
- Multiple instruction issue (superscalar), and
- Multiple instruction streams (multiprocessors and multithreading).

These considerations led to an instruction set architecture (ISA) with just a few instruction formats, simple instruction operations, and simple interactions between instructions.

The architecture is the basis for a whole family of implementations. Each implementation embodies a quantum step forward in the quest for ever higher levels of performance. Implementations may differ in pipeline design, cache design and size, circuit technology, etc. As we'll see later, compiler writers have exploited these differences to tune code for particular implementations. Some family members, for example, have extensions to the basic Alpha ISA.

Unfortunately, two naming systems have been used to refer to family members: the EV/LCA/PCA naming convention and the roughly equivalent numeric name. Here is a small table denoting family members:

Numeric name	EV/LCA/PCA name
21064	EV4
21064A	EV45
21066A/21068A	LCA45
21164	EV5
21164A	EV56
21164PC	PCA56
21264	EV6
21264A	EV67

If all these names seem confusing, don't panic! What is important is to know which Alpha implementation is in *your* machine. And, in some cases, the compiler can even figure that out for you. The Alpha architecture has two instructions, `AMASK` and `IMPLVER`, that can be used to identify processor type (IMPLVER) and instruction set features (AMASK.) Clever library routines can use these instructions to discover features of the host implementation and to select code that has been tuned to exploit those features. If you're curious about your own Alpha, compile the C program at the end of this note with GNU C and execute it. The program, **whatami.c**, uses AMASK and IMPLVER to identify the instruction features and major Alpha processor type of the host. Here is the information about my machine, an XP1000 workstation with a 21264 at its heart.

```
Compaq Alpha 21264 or variant (EV6 core)
Alpha architectural features/extensions
  BWX byte-word extension
  FIX floating point extension
  CIX count extension
  MVI multimedia extension
  Precise arithmetic trap reporting
```

To keep things simple, I'll discuss the XP1000 and the 21264 specifically.

2.3 Instruction set extensions

The 21264 is the newest member of the Alpha family. All members of the Alpha family execute the same basic instruction set. But, the 21264 has several extensions to the basic instruction set that help it to excel at certain kinds of computations and applications. The output from **whatami** shows that the 21264 in my XP1000 has the BWX, FIX, CIX and MVI extensions.

Abbreviation	Extension
BWX	Byte-word extension
FIX	Floating point extension
CIX	Count extension
MVI	Multimedia extension

The basic Alpha ISA has instructions to load and store data in 32-bit and 64-bit words. 8-bit bytes and 16-bit word data can be read by reading the 32- or 64-bit word that encloses the byte or 16-bit word and then extracting the byte or short word from the enclosing data item. Writing a byte or 16-bit word is similar, but the byte or short word is inserted into the enclosing data item which is then written to memory. The BWX or "byte-word" extension adds instructions to read (write) bytes and 16-bit words directly to (from) memory. Of course, this speeds up applications that manipulate bytes and short integers.

The Alpha architecture has two sets of registers: integer and floating point. Each set has 31 registers to hold values of the appropriate type and a special register to generate the constant zero. Early Alpha implementations moved a value between register sets by writing the value to a temporary memory location and then reading the value from the location into the destination register. The floating point extension, FIX, adds instructions to move values between the two register sets. FIX also adds instructions to compute the square root of a floating point value.

The CIX extension adds instructions to count leading zeroes, trailing zeroes and the number of ones in an integer value. These instructions accelerate certain kinds of bit twiddling operations.

The MVI extension is targeted for multimedia applications. They assist saturated arithmetic on pixels and the packing and unpacking of byte and 16-bit word data within 64-bit quantities.

As we'll see later, compilers can sometimes exploit the extra instructions to speed up computations.

> **Tip #2** Identify the Alpha implementation in your machine. Tell the compiler to produce code for that particular implementation.

This tip must be used with caution if you intend to distribute the application in compiled, binary form. Not all Alpha implementations support extensions and runtime errors may result when the processor encounters unimplemented instructions. (See section 3.2.1 for more details.)

2.4 The modern day saga of John Henry

Some programmers feel compelled to "beat the machine" and write out common idioms (e.g., comparing two strings for equality) when a call to a library function will do. Not only are these programmers making more work for themselves, they may be cheating themselves and users from top performance.

The best way to take advantage of hand-coded assembly language is to call a library function that has been written and optimized for the Alpha architecture. Here is a short program to compare strings for equality:

```
#include "tuned.h"

int result ;

static int str_equal(char *a, char *b)
{
  while (*a == *b) {
    if (*a == '\0') return( 1 ) ;
    a++ ;
    b++ ;
  }

  return( 0 ) ;
}

main()
{
  int i ;
  char **pA, **pB ;

  for (i = 0 ; i < 10000000 ; i++) {
    pA = a_strings ;
    pB = b_strings ;
    while(*pA != 0) {
      result = str_equal(*pA, *pB) ;
      pA++ ;
      pB++ ;
    }
  }
}
```

The include file `tuned.h` contains definitions for the strings and the arrays pointing to the strings (`a_strings` and `b_strings`.) The programmer provided a function, `str_equal` to compare two strings for equality. The program was compiled using the Compaq C compiler with the options `-arch ev6 -O4`. The compiler used the BWX instructions and expanded the function `str_equal` in line, thereby eliminating function call overhead. The program took 20.08 seconds to execute.

By way of comparison, a different version of the program was written. The function `str_equal` was changed to:

```
static int str_equal(char *a, char *b)
{
  return( strcmp(a, b) == 0 ) ;
}
```

so that the library function `strcmp` does the comparison. Everything else remained the same and the program was compiled using the same switches and compiler. The program using `strcmp` executed in 10.28 seconds -- roughly half the time. The string sizes likely exaggerated the difference between the two results, but, the message is plain.

Tip #3 Use hand-tuned library functions wherever possible.

Why is `strcmp` faster? Routines that handle strings adjust pointers until they are aligned and subsequent comparisons can be made 8 bytes at a time using 64-bit quadwords. Aligned, parallel comparisons are a big performance win over byte-at-time algorithms, especially when handling long strings.

2.5 Branch prediction

The 21264 is a superscalar processor that executes instructions out-of-order. What does this last statement mean? First, the 21264 is a processing pipeline with seven stages. An instruction passes through each of these stages during its execution. Like other pipeline processors, the 21264 may have several instructions in different stages of execution. Since these stages operate concurrently, many computations may be in progress at any time yielding higher throughput than a strictly sequential processor. Multiple computational units is another important source of parallelism and performance in the 21264. The 21264 has four integer units and two floating point units. At peak, as many as six operations can be performed in a single CPU cycle. Of course, not all parts of a program can or will attain this peak (there are no floating point operations to perform, there is not enough integer arithmetic to compute, and so forth.)

Next, a conventional computer fetches and executes instructions in program order. This strict sequencing discipline limits the ability of the CPU to exploit additional fine-grained parallelism that is available in a program. The 21264 breaks out of the strict discipline in several important ways. Although instructions are fetched in program order, they may be executed in a different order. This is called "out-of-order" execution. Instructions execute when their arguments are ready and available. The flow of instructions through the execution units is guided by the data dependencies between instructions and the availability of data. Instructions complete or "retire" in order, so program correctness is maintained. Thanks to a very large pool of physical registers (the temporary values have to go somewhere!), pipelining and out-of-order execution, the 21264 may have as many as 80 instructions "in flight" at a time.

What this long-winded discussion (!) means is that the 21264 has a voracious appetite for instructions. To satisfy this hunger, the CPU fetches instructions very aggressively. It uses *branch prediction* to speculate about the flow of control through the program by predicting the direction (taken or not-taken) of conditional branches. Branch prediction lets the CPU fetch as many instructions as possible even though the specific outcome of a branch may not yet be known.

Clearly, high performance depends upon the ability of the branch predictor to correctly speculate on the direction of conditional branches. If the predictor speculates incorrectly, some instructions must be discarded and possibly some intermediate results must also be thrown away. The designers studied real world branch prediction carefully and the 21264 branch prediction is quite good on real programs. But, even so, there are still a few things that a programmer can do to help the branch predictor and instruction fetching process.

First off, fewer conditional branches is better. Long stretches of straight line, computational code will execute better than code with many conditional branches. One way to increase the length of straight line compiled code is to ask the compiler to unroll loops. With loop unrolling, the compiler will expand inner loops so that two or more iterations will be performed in a straight line instead of only one iteration. This increases the number of instructions that can be fetched without any speculation at all.

Another way to increase the length of straight line compiled code is to ask the compiler to generate functions in line. Instead of generating a subroutine call, the compiler will expand the function at the call site, almost like expanding a macro.

In-line expansion eliminates subroutine call overhead and also creates opportunities to combine and optimize code between the caller and the subroutine.

Be forewarned, though, that function in-lining sometimes has a dark side, too. If the function to be expanded is a long one, the final straight line code may not fit into the instruction cache. (We haven't discussed 21264 cache memory yet, but its instruction cache is a two-way, 64Kbyte cache, or 16,384 instructions.) Function in-lining *may* result in poor instruction cache behavior. Unfortunately, it's hard to accurately predict dynamic cache behavior, so it may be necessary to try in-lining on a small test data set first to see if it provides a benefit.

The Alpha conditional move instruction (`cmove`) can sometimes be used to avoid a branch or two. Let's take a look at the following example:

```
int f(int a, int b, int c, int d)
{
  if ((a < b) && (c < d)) {
    return(a - d) ;
  } else {
    return(b - c) ;
  }
}
```

The classic machine code for an `if` statement evaluates the conditional expression and branches around the "then" clause to the `else` clause. The Compaq C compiler produced the following assembler code for the `if` statement:

```
# 4 if ((a < b) && (c < d)) {
    cmplt $16, $17, $1       # R1 <- a < b
    cmplt $18, $19, $2       # R2 <- c < d
    and $1, $2, $1           # condition <- R1 and R2
# 5 return(a - d) ;
    cmoveq $1, $17, $16      # if ¬condition, R16 <- b
    cmoveq $1, $18, $19      # if ¬condition, R19 <- c
    nop
    subl $16, $19, $0        # return ( R16 - R19 )
# 6 } else {
# 7 return(b - c) ;
# 8 }
# 9 }
```

The lines beginning with # are comments in the assembly listing to help understand the compiled code. The compiler used two conditional move instructions (`cmoveq`) to eliminate branches.

Tip #4 Ask the compiler to unroll loops.
Tip #5 Ask the compiler to generate procedures in line.
Tip #6 Use C constructs that encourage the compiler to generate conditional move instructions instead of branches.

2.6 Replay traps

The 21264 must track producer-consumer relationships between instructions in order to execute instructions out of program

order and still keep programs architecturally correct. Register scoreboarding and renaming help maintain producer-consumer relationships between instructions that use only registers for computation. *Replay traps* preserve program producer-consumer semantics between load and store instructions. A replay trap is an internal processor mechanism and is not the same as a software interrupt. Replay traps can rob a program of performance. An HLL programmer should understand the conditions that trigger a replay trap and be able to identify bottlenecks in real code. There are also a few programming practices to help reduce the likelihood of replays.

The Compiler Writer's Guide says it best. "Replay traps occur when there are multiple concurrent loads and/or stores in progress to the same address or the same cache index. [...] The best way to avoid replay traps is to keep values in registers so that multiple references to the same address are not in progress at the same time." Most cache memory schemes use a portion of the effective address to select a particular cache memory entry. This field forms the cache index. It is possible for two different addresses to have the same cache index field, thereby triggering a replay trap as well.

Loads and stores to the same address occur in several ways. Here are a few example situations. The compiler may run short of available registers and is forced to keep the value of a variable in memory instead of in a register. The compiler may need extra storage for temporary values and is forced to keep the temporaries on the stack in memory. Finally, the compiler may not be able to analyze pointer values and determine if they refer to the same data object ("aliasing.") The data object must be kept in memory to assure correctness. If the pointers do refer to the same memory data object, then loads and stores to the same address may occur.

There are five kinds of instruction replay trap: store-load replay trap, load-load order replay trap, wrong-size replay trap, load-miss load replay trap, and mapping to same cache line replay trap. Here is a brief description of each kind of replay trap.

Store-load replay trap. Consider the following sequence of store and load instructions as they appear in the program:

```
STQ  R3,12(R4)
LDQ  R2,12(R4)
```

The instructions store to and load from the same memory address. The load instruction uses the data value that is written to the memory location determined by the effective address. If the 21264 issues the load before the store, the load will produce an incorrect value because the result of the store instruction has not yet been written to memory. The 21264 memory system detects this hazard when the store executes and it performs a replay trap, restarting execution at the load instruction. The load will then receive the correct value from memory since program order is effectively restored. The 21264 includes hardware to "learn" store-load dependencies that lead to a replay trap. This reduces the frequency of store-load replay traps, but it is better still to avoid this kind of trap.

Load-load order replay trap. This replay occurs when two loads from the same address issue out of program order. The memory system uses this replay to ensure that load instructions that read the same physical bytes issue in the correct order. The 21264 restarts execution from the newer load. (Instruction A is "newer" than instruction B, if A follows B in program order.) The 21264 does not have hardware to reduce the frequency of load-load order replay traps. Fortunately, instructions tend to issue in program order unless they are not data ready.

Wrong-size replay trap. This replay trap occurs when a store is followed by a load that reads the same data and the load data type is larger than the store. The processor must get the load data from two different internal sources: the store queue, which temporarily holds data to be written to memory, and the data cache. The load replays until the store data is written to the data cache and then all data is obtained from the cache to

satisfy the load.

Load-miss load replay trap. This replay occurs when a load misses and it is followed by another load from the same address. The processor replays until the first load completes and data is loaded into its destination register.

Mapping to the same cache line. The 21264 ensures that there are never multiple outstanding misses to different physical addresses mapping to the same D-cache or L2 (B-) cache line. Thus, loads and stores that are in progress at the same time and that map to the same cache line (32Kb spacing/stride) will cause a replay trap. The processor replays the instruction that triggered the replay trap until the earlier memory operations have completed.

Modern compilers try to keep as much information in registers as possible. Try to use local variables wherever possible instead of global variables. If a variable is used outside of a function, but is not used in another source module, declare the variable as `static`. These suggestions help the compiler to analyze how a variable is used.

Tip #7 Use local variables to help the compiler analyze and optimize code.

If possible, avoid loops where a single iteration or nearby iterations touch data that is 32Kbytes apart. Avoid creating arrays with dimensions that are multiples of 32Kbytes; pad the array with extra cache blocks if necessary. Also note that prefetches can cause replay traps and out-of-order execution can cause multiple iterations of a loop to overlap in execution, so when padding or spacing data references apart, one should consider factors such as the prefetch distance and store delay in computing a safe distance.

Tip #8 Consider data cache size when sizing and stepping through arrays.

2.7 Data alignment

Alpha favors programs with data structures that are aligned on long word (4 bytes) or quadword (8 bytes) address boundaries. Aligned data items can be read (written) with a single load (store) instruction. Unaligned data items, which are split across longword or quadword boundaries, must be read or written using a multi-instruction sequence.

Here is an example to illustrate this point. The following function takes a pointer to an int (longword) as an argument and returns the value of the int:

```
unsigned int f(unsigned int *pInt)
{
  return( *pInt ) ;
}
```

When this function is compiled using the Compaq C compiler, the following code is produced for the return expression:

```
# 6 return( *pInt ) ;
    ldl $0, ($16)
```

This one instruction (`ldl`) performs an indirect, aligned 32-bit load through register 16, which contains the pointer to the longword.

The Compaq C compiler assumes, as a default, that data items are aligned and generates fast code for memory access. The

compiler can be forced to assume misaligned data objects (for indirect load and store instructions) by specifying the option:

`-assume noaligned_objects`, or `-misalign`

When compiled with `-misalign`, the following code is generated:

```
# 6 return( *pInt ) ;
    ldq_u  $2, ($16)        # Load lower quadword
    ldq_u  $3, 3($16)       # Load upper quadword
    extll  $2, $16, $2      # Extract lower part
    extlh  $3, $16, $3      # Extract upper part
    bis    $2, $3, $0       # Join lower and upper part
    sextl  $0, $0           # Extend sign bit
```

The address of the longword (int) is in register 16. If the longword is unaligned, it may straddle two aligned quadwords. The load instructions (`ldq_u`) read the lower and upper quadwords containing the longword. The extract instructions (`extll` and `extlh`) isolate the lower and upper parts of the long word from the quadwords. The logical OR instruction (`bis`) joins the lower and upper parts together.

Tip #9 Use aligned data structures and data access.

It's easy to see the benefit of aligned data structures and data access: one instruction versus six instructions and one memory access versus two accesses. When should unaligned be used? Check out the next section.

2.8 Alignment faults

An *alignment fault* is generated by the Alpha hardware when an attempt is made to load or store a word, longword, or a quadword to/from a register using an address that does not have the natural alignment of the particular data reference (instruction.) Both Tru64 UNIX and Linux fix up the unaligned access, as we'll see in a moment, but the two operating systems handle alignment fault reporting differently.

Let's examine what happens when a pointer refers to a data item which is not aligned. The following program allocates two 64-bit words in an array of 16 unsigned characters (bytes.) The array is aligned on a quadword address boundary. The main program calls the function `f`, which we discussed in the previous section.

```
unsigned char two_64bit_words[16] ;

unsigned int f(unsigned int *pInt)
{
  return( *pInt ) ;
}

main()
{
  int i ;
  unsigned char *pChar ;
  unsigned int *pInt ;
  unsigned int vInt ;
```

```
  pChar = two_64bit_words ;

  for (i = 0 ; i < 8 ; i++) two_64bit_words[i] = 0xFF ;
  for (i = 8 ; i < 16 ; i++) two_64bit_words[i] = 0x0 ;

  for (i = 0 ; i < 1000000 ; i++) {
    pInt = (unsigned int *)(pChar+6) ;
    vInt = f(pInt) ;
  }

  printf("pInt is %8x\n", pInt) ;
  printf("vInt is %8x\n", vInt) ;
}
```

The main program calls `f` 1,000,000 times in order to get a meaningful, measured execution time. The assignment statements in the body of the loop:

```
pInt = (unsigned int *)(pChar+6) ;
vInt = f(pInt) ;
```

deliberately force the pointer address to a non-aligned address and call `f` to read the longword from non-aligned address. The two `printf` statements display the low order 32-bits of the pointer value (address of the longword) and the value of the longword as returned by `f`. The program produces the following output when run:

```
pInt is 20100c9e
vInt is ffff
```

Note that the pointer value (address) is indeed unaligned.

When this program is run on Tru64 UNIX, execution immediately produces a steady stream of messages like:

```
Unaligned access pid=35798 va=0x1400001a6 pc=0x1200011b0 ra=0x120001260
inst=0xa0100000
```

On Tru64, the default action when an alignment fault occurs is to report the alignment fault, fix up the memory reference, and continue. The Tru64 kernel fixes up an unaligned read operation by reading the data, putting the data item into the destination register, and restarting the program after the load instruction. This is called an "alignment fix-up." The error messages display the process ID (pid), the virtual address causing the fault (va), the program counter (pc) and the offending instruction (inst.) This information can be used to find the location of the alignment fault in the code and to remove its cause. The default action can be changed using the `uac` command.

When an instruction causes an alignment trap on Alpha Linux, the trap switches control to the Linux kernel. The Linux kernel *silently* fixes up the unaligned access and restarts the program after the offending instruction. However, the Linux kernel does write a log message to `/var/log/messages` when it performs an alignment fix-up on behalf of a process. Here is the log entry for the misaligned example above:

```
Oct 18 09:22:49 amt157 kernel: fast_code(633):
unaligned trap at 00000001200008c0: 0000000120100c9e 28 0
```

The first address is the address of the offending instruction. (Yes, Alpha address/pointer values are 64-bits, so watch casts

between integer and pointer types.) The second address is the offending effective address.

The first step in eliminating alignment faults is to relate the offending instruction back to the source code. One method (shown below for Linux) is to run the program using `gdb` and then disassemble the function containing the offending instruction. Here is an example of the method in action:

```
26 > gdb fast_code
This GDB was configured as "alpha-redhat-linux"...
(gdb) run
Starting program: /home/pjd/misc/unaligned/fast_code
pInt is 20100c9e
vInt is ffff
Program exited normally.
(gdb) disassemble 0x01200008c0
Dump of assembler code for function f:
0x1200008c0 <f>:      ldl v0,0(a0)
0x1200008c4 <f+4>:    ret zero,(ra),0x1
0x1200008c8 <f+8>:    unop
0x1200008cc <f+12>:   unop
End of assembler dump.
(gdb) quit
```

In this case, it's easy to relate the disassembly back to the source because the function `f` is so short. For long functions, you may need to ask the compiler to generate assembler code for the program and then compare the disassembly against the generated code. We'll show how to generate assembler code later. The alignment fault is quite probably due to a misused pointer or incorrect pointer arithmetic.

We compiled this program in two different ways using the Compaq C compiler.

- The first compile assumed aligned data items and produced "fast" code, a single `ldl` instruction to read the longword.
- The second compile assumed non-aligned data items and produced conservative code using the full five instruction sequence to read a non-aligned longword.

The "fast" version ran in 0.41 seconds while the "conservative" version ran in 0.01 seconds (elapsed time.)

What happened to produce this seemingly unexpected result? The portion of time spent in user mode (the application) versus system mode (the operating system) partially tells the tale.

Assumption	User time	System time	Elapsed time
Aligned data ("fast")	0.067 sec	0.347 sec	0.410 sec
Unaligned data ("conservative")	0.009 sec	0.000 sec	0.010 sec

The "fast" version not only spent more time in user mode, but spent a (relatively) whopping 0.347 seconds in the operating system.

Tip #10 Detect, identify and remove performance robbing alignment faults.

Alignment faults should be investigated and removed even if they do not occur within the performance critical parts of a program. The presence of an alignment fault indicates a possible mismatch between the intentions of the programmer and the reality of the execution engine and may be a bug. The implications in a multithreaded program are even worse.

- The unaligned access is not atomic. The program will fail if it depends on atomicity.
- An unaligned access touches more bytes and the lock granularity is bigger. This may lead to a data race between two threads even when explicit synchronization is used.

Thus, an alignment fault often indicates a coding problem that needs to be considered and corrected.

2.9 XP1000 memory hierarchy

The memory hierarchy in the XP1000 workstation has five levels as summarized in the table below.

Level	Size	Latency
Registers	31 integer, 31 floating	Immediate
L1 cache	64Kb instruction, 64Kb data	2 cycles
L2 cache	4Mb	26 cycles
Main memory	Up to 2Gb	130 cycles
Virtual memory	Many Gb's	Millions of cycles

Latency is the time needed to satisfy a request for data. The general registers are located within the processor and are available for immediate use. Data present in the L1 cache is available in just two CPU cycles. So, clearly, there is a substantial performance benefit when data is available in either the general registers or L1 cache, both of which are located on-chip.

> **Tip #11** Help the compiler to keep data items as close to the processor as possible.

In general, this tip means helping the compiler keep frequently used values in registers instead of memory, and writing programs using data structures and data access patterns with good cache memory behavior. We'll make some specific suggestions on how to accomplish this later.

Latency increases rapidly as the data becomes farther from the central processor. Latency from the L2 cache and primary memory are 26 and 130 cycles, respectively. If no other useful work can be done while waiting for data to arrive, the processor must sit idle. The processor and compiler try to reduce the effect of long latency memory operations in two ways:

- Compilers try to arrange ("schedule") instructions to keep the processor busy even if data is not readily available when it is requested by a load instruction. The compiler will try to place unrelated instructions after a load and before the instruction that consumes the result of the load.
- The 21264 provides read and write queues to hold pending load and store requests. Coupled with out-of-order execution, this lets the CPU execute ahead in the instruction stream without waiting for loads and stores to complete. If at some point the processor needs a data item and cannot execute further, the processor will stall.

Two other important techniques are prefetching and software pipelining. The 21264 provides special operations to identify to the memory subsystem those data items (cache blocks) which will be required next. If these operations can be performed early enough, then the memory system will have the data items close at hand when they are actually requested by a load instruction. This is called "prefetching." The Compaq C compiler will automatically insert prefetches as long as the

programmer keeps memory references "simple," so that the compiler can recognize repeated references with a loop-invariant stride.

The Alpha 21264 initiates a prefetch operation by executing one of the load instructions as summarized in the table below. Note that the destination register is `R31` or `F31`. When used as a source register, `R31` and `F31` return integer zero and floating point zero, respectively. When used as a destination register as shown below, `R31` and `F31` denote the purpose of these instructions as a prefetch operation. Earlier Alpha implementations ignore these instructions. Some care must be taken as a prefetch with an invalid address must be dismissed by firmware and a prefetch can cause an alignment trap.

Prefetch	Description
`ldl R31,ea`	Normal cache line prefetch; load cache block into L1 and L2
`lds R31,ea`	Prefetch with intent to modify; load block in dirty state so subsequent stores can immediately update the block
`ldq R31,ea`	Prefetch, evict next; evict block from L1 cache as soon as there is another block loaded at the same cache index
`wh64 ea`	Write hint; set block contents to indeterminate state and obtain write access to cache block without reading its old contents

Prefetch instructions and write hint (Alpha 21264)

The write hint instruction, `WH64`, is used when the program intends to completely write an aligned 64-byte block of memory. The old contents of the block are not read, saving memory bandwidth. The write hint instruction sets the contents of the block to a random, unknown state.

Software pipelining is a technique that is usually applied to loops by the compiler. The generated loop code prefetches data during iteration *N* that it will actually require during iteration *N+1* or perhaps even further in the future. Software pipelining can be applied when the compiler can analyze dependencies and discern a regular, predictable pattern to array access. The Compaq C compiler can arrange loops for software pipelining.

> **Tip #12** Help the compiler exploit software pipelining by keeping inner loop code straightforward and by keeping dependencies simple.

2.10 Cache memory

The XP1000 has a two levels of cache memory, L1 and L2. Here is a quick summary of cache characteristics.

- L1 cache memory
  - Implemented on-chip
  - Separate instruction and data caches ("Harvard" architecture)
  - 64Kbyte instruction cache, 64Kbyte data cache
  - Virtual-addressed, two-way set associative organization
  - 64 byte block size
- L2 cache memory

- Implemented off-chip
- Unified instruction and data cache
- 4Mbyte capacity
- Direct-mapped organization
- 64 byte block size

Cache memory design is a difficult science. The designers ran many simulations to find the right trade-off between cache size and organization. A larger cache size increases the probability that a data item will be found in the cache at the expense of more chip real estate or memory devices. Set associative caches increase conflict tolerance, that is, the number of addresses that can map to the same cache index. A direct-mapped cache cannot tolerate any conflicts while a two-way set associative cache tolerates two-way conflicts. To further cloud the choice of size and organization, a larger direct-mapped cache can often perform as well as a two-way set associative cache.

The effect of cache size and organization on program design and behavior is what is most important to programmers. From the last section on XP1000 memory hierarchy, it's a clear win to get and keep data items in the L1 cache. Caches work most effectively when a program exhibits good *temporal* and *spatial* locality.

- Most recently used data items or instructions are more likely to be available in cache (temporal locality.)
- Data items or instructions that are close together in memory are more likely to be in cache when needed (spatial locality.)

Temporal locality can be increased by processing a particular data item completely in one continuous step rather than a piecemeal approach which processes the data item in many steps with long periods of unrelated activity between steps. Spatial locality can be increased by stepping through arrays sequentially using a unit stride or by placing frequently accessed data items within the same cache block. If a program uses a large record structure (`struct`), locality can be improved by grouping related fields together:

```
struct {
    // Fields used during phase 1 of the program
    int a ;
    int b ;
    ...
    // Fields used during phase 2 of the program
    int x ;
    ...
}
```

Spatial locality is destroyed by memory access patterns that hop around in memory. Subroutines that call each other frequently should be close to each other in memory in order to improve their instruction cache behavior.

Tip #13 Design and code for good temporal and spatial locality.

Since the space of effective addresses is bigger than the cache size, the cache hardware must map an address into a cache entry in a very short period of time. The cache uses a field within the effective address as an index into its memory. Thus, many addresses within the address space will all map to the same cache entry. The remaining bits in the full address are kept in the cache entry and are checked during the read or write operation. If the stored bits match the remaining bits of the effective address, then the address hits and information can be updated or returned for that address. If the load or store misses in the cache (i.e., there is not a cache entry for the effective address), the old entry must be discarded and a new one must be created. Program performance is best when the hit rate is high.

It's possible to have pathological cases with a low hit rate and bad program performance. Let's consider a program with three separate data streams. In the code below, data from the three arrays, `a, b` and `c` can be thought of as three data streams flowing into and out of the inner loop of function `compute`:

```
static int a[8192], b[8192], c[8192] ;

void compute(
{
  int i ;

  for (i = 0 ; i < 8192 ; i++) {
    c[i] = a[i] + b[i] ;
  }
}

main()
{
  int i ;

  for (i = 0 ; i < 100000 ; i++) {
    compute() ;
  }
}
```

The three arrays are each the size of one set in the 21264 L1 cache (32Kbytes.) As the inner loop of `compute` steps through the three arrays, each memory access will map to the same cache entry and will miss. This program takes 22.051 seconds to execute. After padding the array bounds:

```
#define PADDING 32
static int a[8192+PADDING], b[8192+PADDING], c[8192+PADDING] ;
```

the execution time is reduced to 9.760 seconds, better than half the execution time of the unpadded version. The padding is enough to shift the position of the arrays in primary memory such that the loads from `a` and `b` and the store to `c` map to different cache entries during a given iteration.

Frequent, repeated conflicts ("thrashing") of this type can sap program performance. Thrashing may also occur when stepping through an array with a stride that is a multiple of the cache size.

Tip #14 Avoid array sizes and strides that are multiples of the cache size.

The examples above were compiled with the Compaq C compiler at the lowest level of optimization to obtain the most straightforward machine code for the inner loop. When compiled at the highest level (`-O4`), however, the compiler unrolled the inner loop, employed software pipelining, offset the array accesses and used the write hint instruction (`wh64`) to improve store behavior. After these optimizations, the unpadded and padded programs ran in 7.886 and 7.622 seconds, respectively. Still want to beat the compiler?

2.11 Virtual memory

As mentioned earlier, virtual memory provides a large logical (virtual) address space for a each process. The logical address space can be much bigger than the physical memory of the host platform.

Both Tru64 UNIX and Linux divide physical memory into 8Kbyte pages. The OS manages these pages and makes them available to a process as they are needed by the program. The pages that a program needs to execute, the *working set*, must be in physical memory when the program actually runs. The working set is usually small subset of all pages that make up the program's virtual memory space. Multiple programs (working sets) can often be fit into physical memory letting the OS switch between ready-to-run programs when the currently executing program blocks due to I/O, a page fault, etc. This kind of resource sharing makes for efficient use of processor time and memory.

Of course, every virtual address issued by a running program must be mapped to the corresponding physical memory page containing the data for that location. This mapping is handled by the operating system and the processor. Linux maintains a table describing the mapping from virtual to physical pages for each process. (This is really quite a simplification!) If the OS had to intervene in every load or store instruction, execution performance would be slow. So, the 21264 provides two translation look-aside buffers: the Instruction-stream Translation Buffer (ITB) and the Data-stream Translation Buffer (DTB.) The ITB and DTB contain the most recently used page mapping information. The ITB and DTB feed the processor hardware that translates virtual addresses to physical addresses, so the actual mapping takes place very fast.

ITB and DTB sizes are limited since they are physical, on-chip circuits. Both the ITB and DTB contain 128 entries. Each entry describes the mapping for one 8Kbyte page. (The 21264 hardware is more flexible than this, but this is how Linux uses the ITB and DTB.) Therefore, the ITB and DTB can each cover 1Mbyte of virtual address space. If the program issues a virtual address that misses in the ITB or DTB, a new entry is allocated for the missing mapping information and the operating system is called to load the entry. Allocation is round robin. Programs that jump "randomly" in a very large virtual address space will not perform well if the operating system must frequently fill ITB or DTB entries.

> *Note:* Some versions of Tru64 UNIX support "large pages," i.e., pages that are bigger than 8Kbytes. Please see the documentation for your version of Tru64 UNIX for more details. Large pages can be used to reduce the number of DTB misses.

The performance situation is even more serious when the needed page itself is not available in physical memory, a "page fault." Now the operating system must read the page from secondary storage, usually a disk, and fill the ITB or DTB entry. The time needed to retrieve the page is several milliseconds depending upon the speed of the disk. Since page fault processing is so slow, the operating system will switch the processor to a ready-to-run process in order to keep the CPU busy. The faulting program waits for the page to be read into physical memory and performance suffers.

Good performing programs take virtual memory into account. Temporal and spatial locality play their roles again, but this time, the unit of memory allocation is the 8Kbyte page instead of individual data items. Programs should manipulate data within a small working set and minimize page faults. Data structures (especially arrays) should be laid out such that the access pattern will be uniform and will not skip around between pages. Frequently executed functions should reside on the same few pages. Infrequently executed functions like error handlers should reside on their own pages -- pages that are only rarely brought into the working set.

> **Tip #15** Use data structures and code organization that are "virtual memory ready."

Linux and UNIX on other architectures use a 4Kbyte page size. This may be an issue when porting programs to Alpha.

3. Example program

The program that we will use as the running example in this section is taken from the realm of image processing. The program applies the morphological dilation operation to an image. Dilation is performed by applying a structuring element to each pixel in the image. The effect of the dilation operation on a particular pixel is given by the formula:

$$Z_{i,j} = \text{Maximum } [A_{i-k,j-l} + B_{k,l}]$$

where -15<=k<=15 and -15<=l<=15 for a 31 by 31 pixel structuring element. In the example program, the input and result images are each 512 by 512 pixels where each pixel is 16-bits deep and the structuring element is 31 by 31 pixels. The input image, structure element and result image are stored in the following global arrays:

```
short int A[IMAGE_ROWS+30][IMAGE_COLS+30] ; /* Input image */
short int B[SE_ROWS][SE_COLS] ;              /* Structuring element */
short int Z[IMAGE_ROWS+30][IMAGE_COLS+30] ; /* Output image */
```

The symbolic constants `IMAGE_ROWS`, `IMAGE_COLS`, `SE_ROWS`, and `SE_COLS` are defined to be:

```
#define IMAGE_ROWS  512
#define IMAGE_COLS  512
#define SE_ROWS      31
#define SE_COLS      31
```

The algorithm handles the pixels around the edges of the input image by oversizing the input image, a "picture frame," and by initializing the frame to the minimum signed pixel value, -32767. The algorithm applies the structuring element to the input image beginning with the first real pixel in the input image. The values in the frame area handle the edge cases without the use of conditional statements. The minimum pixel value nullifies the effect of the picture frame.

By using explicit static arrays and array bounds, the compiler will be able to take advantage of known information to produce highly optimized code. It is generally a good idea to provide as much information to the compiler as possible so that it can analyze the source and take advantage of its optimization techniques. Sometimes, if specific information about memory use is not available, perhaps through the use of dynamic storage and pointers, the compiler must produce conservative code that is guaranteed to preserve correctness and performance is lost.

The program contains functions to read the input image and structuring element, process the input image and write the result image. The following two C functions are the "heart" of the dilation program and perform the dilation operation.

```
DilatePixel(r, c) int r, c ;

  {
  register int i, j ;
  int sum, the_max ;

  the_max = MINUS_INFINITY ;

  for (i = 0 ; i < 31 ; i++)
    {
    for (j = 0 ; j < 31 ; j++)
      {
      sum = A[r+i][c+j] + B[i][j] ;
      if (sum > the_max) the_max = sum ;
      }
    }

  Z[r][c] = the_max ;
```

```
  }

DilateImage()

  {
  register int r, c ;

  for (r = 0 ; r < IMAGE_ROWS ; r++)
    {
    for (c = 0 ; c < IMAGE_COLS ; c++)
      {
      DilatePixel(r, c) ;
      }
    }
  }
```

The program spends most of its time in these two functions and we will be taking a look at the Alpha code which is generated for `DilatePixel` in particular.

3.1 Measuring execution time

In order to compare the effect of applying compiler optimizations, etc., we need a way to time the execution of a program. The command line shell (csh, bash) often has a built-in command, time, that displays information about the execution of a program. Here is a sample of the time command (from csh) in action as applied to the dilation program as it runs.

```
38 > time dilate heart.ima ball out.ima
Dilation (dilate) untimed `spec' version
Image file (heart.ima) read.
Structuring element file (ball) read.
Output image file (out.ima) written.
Leaving program (dilate)
2.879u 0.010s 0:02.910 99.8%     0+0k 0+0io 79pf+0w
```

The first four numbers in the last line provide information on execution time.

- The first number is the number of seconds spent in user mode. This is the amount of CPU time spent running the actual application code.
- The second number is the number of seconds spent in system mode. This is the amount of CPU time spent by the operating system on behalf of the application program (e.g., handling I/O, memory management, etc.)
- The third number is the elapsed time, that is, how much "wall clock" time expired from starting the program to completion.
- The fourth number is the utilization, or CPU time (both user and system) as a percentage of elapsed time.

The number preceding "`pf`" is the number of page faults, by the way. Utilization may not be 100% as the system may spend time waiting for I/O operations to complete or perhaps other users and activities may take resources away from the program. In this example, the dilation program is running on a relatively quiet workstation by itself and it spends most of its time executing application code with little I/O activity.

3.2 Compaq C compiler

The Compaq C compiler provides many options to help you to tune up your program for its best performance. This section will illustrate the use of just a few options. You should definitely read the `man` page (`cc(1)` on Tru64 UNIX and `ccc(1)` on Linux.) Options have been added to the Linux version of the Compaq C compiler to enhance its compatibility with GCC (language, makefiles, etc.) These options should ease the process of introducing the Compaq C compiler into Linux-based builds.

Compiler option	Description
`-Wf,-S`	Generate assembler language file
`-arch` *x*	Generate code for Alpha implementation *x*
`-host`	Generate code for the Alpha implementation of the host
`-O`*x*	Sets the optimization level to *x*
`-fast`	Sets a number of options to increase performance

One way to learn more about what the compiler does and produces for certain source language constructs is to examine the code generated by the compiler. Object code is just binary, of course, and disassembly often loses valuable symbolic information. The C compiler provides an option to help us out. In the earlier examples, we used the `-Wf,-S` option to have the Compaq C compiler produce a file containing the assembler language version of the compiled code. The resulting file has the `.s` extension and the root of the file name is the root name of the source file. For example, the Linux compile command:

```
ccc -Wf,-S dilate.c
```

produces the file `dilate.s`. The file `dilate.s` can be printed, opened in an editor, etc. since the code inside is just ASCII text. Addendum B below provides an introduction to reading Alpha assembler code. If you intend to really dig into Alpha, we recommend reading *Alpha RISC Architecture for Programmers* by James S. Evans and Richard H. Eckhouse or the *Alpha Architecture Reference Manual (Third Edition)*.

> *Note:* On Linux, the Compaq C compiler is invoked using "`ccc`" as "`cc`" invokes the GNU C compiler. The examples below were produced on Linux. Substitute "`cc`" for "`ccc`" on Tru64 UNIX.

3.2.1 Compiling for Alpha implementation

Let's compile the dilation program without any compiler options to establish a baseline execution time. The measured user, system and elapsed times are:

Compilation command	User time	System time	Elapsed time
`ccc dilate.c`	2.879 sec	0.010 sec	2.910 sec

The default optimization level is `-O1`, which performs basic local and global optimizations, but does not in-line functions or perform software pipelining.

`DilatePixel` inner loop - baseline

```
#    232    the_max = MINUS_INFINITY ;
```

```
#    233
#    234   for (i = 0 ; i < 31 ; i++)
#    235     {
#    236     for (j = 0 ; j < 31 ; j++)
#    237       {
#    238       sum = A[r+i][c+j] + B[i][j] ;
       ldq     $2, ($gp)                  # 000238
       addq    $16, $16, $0
       sextl   $17, $17                   # 000228
       s8addq  $0, $16, $0                # 000238
       ldq     $3, ($gp)
       addq    $0, $0, $0
       mov     -32767, $1                 # 000232   $1 = the_max
       s8subq  $0, $16, $0                # 000238
       clr     $16                        # 000234
       s4addq  $0, $2, $2                 # 000238
       addq    $17, $17, $17
L$11:                                     # 000234   Top of outer loop
       addq    $2, $17, $4                # 000238
       clr     $5                         # 000236   Set j to zero ($5 = j)
       mov     $3, $6                     # 000238
L$12:                                     # 000236   Top of inner loop
       ldq_u   $8, 1($4)                  # 000238   Load quadword with A
       ldq_u   $19, 1($6)                            Load quadword with B
       unop
       lda     $7, 2($4)
       lda     $18, 2($6)
       extqh   $8, $7, $8                            Extract A from quadword
       addl    $5, 1, $5                  # 000236   Increment j
       extqh   $19, $18, $19              # 000238   Extract B from quadword
       lda     $4, 2($4)                  # 000236   Advance pointer to A
       sra     $8, 48, $7                 # 000238   Align A right
       lda     $6, 2($6)                  # 000236   Advance pointer to B
       sra     $19, 48, $18               # 000238   Align B right
       cmplt   $5, 31, $19                # 000236   Compare j with 31
       addq    $7, $18, $7                # 000238   Compute Sum = A + B
#    239       if (sum > the_max) the_max = sum ;
       cmplt   $1, $7, $8                 # 000239   Compare the_max with Sum
       cmovne  $8, $7, $1
       bne     $19, L$12                  # 000236   Bottom of inner loop
       addl    $16, 1, $16                # 000234   Increment i ($16 = i)
       lda     $2, 1084($2)
       cmplt   $16, 31, $7                           Compare i with 31
       lda     $3, 62($3)
       bne     $7, L$11                              Bottom of outer loop
#    240       }
#    241     }
#    242
#    243   Z[r][c] = the_max ;
       ldq     $8, ($gp)                  # 000243
       s4addq  $0, $8, $0
```

```
        addq    $0, $17, $0
        ldq_u   $5, ($0)                                Load Z[r][c]
        inswl   $1, $0, $6                              Insert the_max (short)
        mskwl   $5, $0, $5
        bis     $5, $6, $5
        stq_u   $5, ($0)                                Store Z[r][c]
#    244   }
        ret     ($26)                       # 000244
```

The code for the inner loop of `DilatePixel` (shown above) is straightforward. The compiler has created new induction variables for array indexing to simplify address calculation. The compiler has assumed that short integers do not cross quadword address boundaries. However, the default case is to generate code for a generic Alpha (i.e., the core instruction set) and the compiler has generated quadword loads, stores, extracts and inserts to read and write 16-bit words.

As the dilation program stores pixels as 16-bit short integers, it would likely benefit from the 21264 byte/word extensions (BWX.) There are two ways to select the Alpha implementation at compile time: `-arch` *x* and `-host`. The `-arch` *x* option sets the Alpha implementation to *x*. Possible values for *x* include `generic`, `host`, `ev56`, `ev6`, etc. The `-arch host` option directs the compiler to generate code for the Alpha implementation of the host computer that is running the compiler. `-host` is shorthand for `-arch host`.

There is an important *gotcha*, however, if you intend to distribute your program in pre-compiled, binary form. If a 21164 (ev56) processor executes a program that was compiled for a 21264 (ev6), for example, the processor will produce an **illegal instruction trap** when it encounters any unsupported instruction. The safest way to compile a program for any Alpha target is to select the `generic` target implementation. Another option is to use `-tune` *x*, which behaves like `-arch` *x*, but instructions specific to *x* are guarded by `AMASK` instructions. Illegal instruction traps will be avoided at the cost of a larger binary image.

Here are the results after recompiling with `-arch ev6`. The execution time has improved substantially.

Compilation command	User time	System time	Elapsed time
`ccc dilate.c`	2.879 sec	0.010 sec	2.910 sec
`ccc -arch ev6 dilate.c`	1.990 sec	0.008 sec	1.998 sec

The compiled code for the inner loop of `DilatePixel` is shown below. The compiler has used byte/word instructions (`ldwu`, `sextw` and `stw`) to streamline the inner loop. `Unop` instructions were added to align the beginning of each loop on an octaword (16-byte) address boundary. Each label at a loop top is the target of a conditional branch at the end of the corresponding loop. Fetch efficiency is improved by aligning the targets because the 21264 fetches instructions in groups of four (so-called "quadpacks," each of which begins on a 16-byte address boundary.)

`DilatePixel` inner loop - EV6 implementation

```
#    232   the_max = MINUS_INFINITY ;
        mov     -32767, $1              # 000232      $1 = the_max
#    233
```

```
 #    234   for (i = 0 ; i < 31 ; i++)
 #    235     {
 #    236     for (j = 0 ; j < 31 ; j++)
 #    237       {
 #    238       sum = A[r+i][c+j] + B[i][j] ;
        addq    $16, $16, $0          # 000238
        ldq     $2, ($gp)
        ldq     $3, ($gp)
        addq    $17, $17, $17
        s8addq  $0, $16, $0
        addq    $0, $0, $0
        s8subq  $0, $16, $0
        clr     $16                   # 000234
        s4addq  $0, $2, $2            # 000238
        unop                                      Align to octaword boundary
        unop
L$11:                                 # 000234      Top of outer loop
        addq    $2, $17, $4           # 000238      $4 = ptr to A
        clr     $5                    # 000236      Clear j ($5 = j)
        mov     $3, $6                # 000238      $6 = ptr to B
        unop                                      Align to octaword boundary
L$12:                                 # 000236      Top of inner loop
        ldwu    $8, ($4)              # 000238      Load pixel from A
        ldwu    $19, ($6)                         Load pixel from B
        addl    $5, 1, $5             # 000236      Increment j
        lda     $4, 2($4)                         Advance ptr to A
        lda     $6, 2($6)                         Advance ptr to B
        sextw   $8, $8                # 000238      Sign extend A value
        sextw   $19, $19                          Sign extend B value
        addq    $8, $19, $8                       Form sum ($8)
        cmplt   $5, 31, $19           # 000236      Is j < 31?
 #    239       if (sum > the_max) the_max = sum ;  $8 = sum, $1 = the_max
        cmplt   $1, $8, $20           # 000239      Compare sum and the_max
        cmovne  $20, $8, $1                       Conditional move
        bne     $19, L$12             # 000236      Bottom of inner loop
        addl    $16, 1, $16           # 000234      Add 1 to i ($16 = i)
        lda     $2, 1084($2)                      Advance to next row of A
        cmplt   $16, 31, $20                      Is i < 31?
        lda     $3, 62($3)                        Advance to next row of B
        bne     $20, L$11                         Bottom of outer loop
 #    240       }
 #    241     }
 #    242
 #    243   Z[r][c] = the_max ;
        ldq     $7, ($gp)             # 000243
        s4addq  $0, $7, $0
        addq    $0, $17, $0
        stw     $1, ($0)                          Store Z[r][c]
 #    244   }
        ret     ($26)                 # 000244
```

3.2.2 Loop unrolling

The last optimization that we will demonstrate is loop unrolling. This optimization comes into play at level four (`-O4`.) The table below summarizes the optimization levels and the optimizations that each level enables. As noted earlier, level `-O1` is the default level.

Level	Optimization
`-O0`	None
`-O1`	Local optimizations and recognition of common subexpressions. Global optimizations including code motion, strength reduction and test replacement, split lifetime analysis and code scheduling
`-O2, -O`	Inline expansion of static procedures. Additional global optimizations that improve speed (at the cost of extra code size) such as expansion of integer multiplication and division using shifts, loop unrolling and code replication to eliminate branches
`-O3`	Inline expansion of global procedures
`-O4`	Software pipelining using dependency analysis, vectorization of some loops on 8-bit and 16-bit data (`char` and `short`) and insertion of `NOP`instructions to improve scheduling

At level `-O4` the compiler performs deeper analysis of loop bodies and looks for opportunities to perform software pipelining. Compiling at `-O4` produced an additional speed-up as shown in the next table.

Compilation command	User time	System time	Elapsed time
`ccc dilate.c`	2.879 sec	0.010 sec	2.910 sec
`ccc -arch ev6 dilate.c`	1.990 sec	0.008 sec	1.998 sec
`ccc -arch ev6 -O4 dilate.c`	1.642 sec	0.006 sec	1.649 sec

The compiled code (below) looks *much* bigger. The compiler has unrolled the inner loop four times, so that each time around the compiled loop is equal to four normal iterations around the original loop. The compiler has created three loops with the structure:

```
Outer loop (induction variable: i, starting at L$11)
  Inner loop #1 (induction variable: j, starting at L$12)
  Inner loop #2 (induction variable: j, starting at L$14)
```

The outer loop iterates through the rows of both the image neighborhood and the structuring element as before. The original inner loop has been split into two loops. The first inner loop has been unrolled and handles *j=0..27*. The second inner loop handles the "leftovers" or *j=28..30*. Notice that everything in the first inner loop comes in fours: four `ldwu`, four `sextw`, four `complt`, four `cmovne`, etc. The compiler keeps all of the bookkeeping and register allocations straight.

Although the code is much larger, the extra instructions buy an additional performance boost. If the number of iterations through the inner loop was larger, the relative performance gain would be even greater.

`DilatePixel` inner loop - loop unrolling

```
 #    234   for (i = 0 ; i < 31 ; i++)
 #    235     {
 #    236     for (j = 0 ; j < 31 ; j++)
 #    237       {
 #    238       sum = A[r+i][c+j] + B[i][j] ;
       addq    $16, $16, $0        # 000238
       ldq     $2, ($gp)
       ldq     $3, ($gp)
       addq    $17, $17, $17
       s8addq  $0, $16, $0
       addq    $0, $0, $0
       s8subq  $0, $16, $0
       clr     $16                 # 000234    Clear loop counter ($16 = i)
       s4addq  $0, $2, $2          # 000238
       unop
       unop
L$11:                              # 000234    Top of outer loop
       addq    $2, $17, $4         # 000238
       clr     $5                  # 000236    Clear loop counter ($5 = j)
       mov     $3, $6              # 000238
       unop
L$12:                              # 000236    Top of inner loop
       ldwu    $8, ($4)            # 000238    $8 and $19 (sum1)
       ldwu    $19, ($6)
       addl    $5, 4, $5           # 000236    Add four to loop counter
       lda     $4, 8($4)           # 000238
       ldwu    $20, -6($4)                   $20 and $21 (sum2)
       ldwu    $21, 2($6)
       lda     $6, 8($6)
       ldwu    $22, -4($4)                   $22 and $23 (sum3)
       ldwu    $23, -4($6)
       sextw   $8, $8
       sextw   $19, $19
       sextw   $20, $20
       sextw   $21, $21
       addq    $8, $19, $8                   Form sum1
       ldwu    $19, -2($4)                   $19 and $23 (sum4)
       addq    $20, $21, $20                 Form sum2
 #    239       if (sum > the_max) the_max = sum ;
       cmplt   $1, $8, $21         # 000239    Compare sum1 and the_max
       sextw   $22, $22            # 000238
       sextw   $23, $23
       cmovne  $21, $8, $1         # 000239
       addq    $22, $23, $22       # 000238    Form sum3
       ldwu    $23, -2($6)
       sextw   $19, $19
```

```
        cmplt   $1, $20, $24        # 000239    Compare sum2 and the_max
        cmovne  $24, $20, $1
        sextw   $23, $23            # 000238
        cmplt   $5, 28, $20         # 000236    Note new loop bound (28)
        addq    $19, $23, $19       # 000238    Form sum4
        cmplt   $1, $22, $8         # 000239    Compare sum3 and the_max
        cmovne  $8, $22, $1
        cmplt   $1, $19, $21                  Compare sum4 and the_max
        cmovne  $21, $19, $1
        bne     $20, L$12           # 000236    Bottom of inner loop
        unop
        unop
        unop
L$14:                                         Top of inner loop #2
        ldwu    $8, ($4)            # 000238    Get A
        ldwu    $23, ($6)                     Get B
        addl    $5, 1, $5           # 000236    Increment j
        lda     $4, 2($4)                     Advance pointer to A
        cmplt   $5, 31, $18                   Compare j with 31
        lda     $6, 2($6)                     Advance pointer to B
        sextw   $8, $8              # 000238    Sign extend A
        sextw   $23, $23                      Sign extend B
        addq    $8, $23, $8                   Form sum
        cmplt   $1, $8, $7          # 000239    Compare Sum with the_max
        cmovne  $7, $8, $1                    Conditional move
        bne     $18, L$14           # 000236    Bottom of inner loop #2
        addl    $16, 1, $16         # 000234    Increment i
        lda     $2, 1084($2)
        cmplt   $16, 31, $21                  Compare i with 32
        lda     $3, 62($3)
        bne     $21, L$11                     Bottom of outer loop
#    240        }
#    241      }
#    242
#    243   Z[r][c] = the_max ;
        ldq     $20, ($gp)          # 000243
        s4addq  $0, $20, $0
        addq    $0, $17, $0
        stw     $1, ($0)                      Store Z[r][c]
#    244   }
        ret     ($26)               # 000244
```

3.3 GNU C compiler

The GNU C compiler offers a similar set of optimization options. The optimization levels supported by the GNU C compiler are summarized in the table below.

Level	Optimization
None	Debugging produces expected results; statements are independent
`-O0`	Do not optimize
`-O1, -O`	Basic optimizations are enabled including thread jumps, defer pop and omit frame pointer
`-O2`	Perform optimizations that do not involve a space-speed trade-off including loop strength reduction and instruction scheduling
`-O3`	Perform function in-lining

Unlike the Compaq C compiler, the GNU compiler does not optimize code by default. When `dilate.c` is compiled without specifying any options:

```
cc dilate.c
```

the resulting executable program runs in 9.845 seconds (user time.) Why is this time so much longer than the default case for the Compaq C compiler (9.845 versus 2.879 seconds?) First off, the code is compiled for a generic Alpha implementation, but, more importantly, the GNU compiler produces code to assist debugging by maintaining statement structure. Thus, you can place breakpoints between the code for source statements, change and examine variables, etc. and get the results that you would expect. By being faithful to the original source, the compiler passes on opportunities for optimization.

Level `-O2` performs many optimizations, but does not unroll loops or expand functions in-line. Level `-O3` enables function in-lining, but not loop unrolling. Loop unrolling must be enabled by using one or the two switches: `-funroll-loops` or `-funroll-all-loops`. The first option unrolls only those loops for which the number of iterations can be determined at compile time.

The GNU compiler offers three different sets of options to select and set characteristics of the target Alpha implementation. The first set of options enable or disable use of instruction set extensions:

```
-mbwx
-mno-bwx
-mcix
-mno-cix
-mmax
-mno-max
```

The second option sets the instruction set and scheduling parameters for a particular Alpha implementation. The option has the form:

`-mcpu=`*`cpu_type`*

where *cpu_type* may be: `ev4, ev5, ev56, pca56, ev6 21064, 21164, 21164pc, 21164PC` or `21264`. The compiler gives precedence to the `mcpu` option first, instruction set specifications second, and by default, the type of Alpha on which the compiler was built. The third option lets you set the scheduling latency for memory references:

`-mmemory-latency=`*`time`*

where *time* may be a decimal number specifying clock cycles, or the symbols `L1, L2, L3` or `main`. The symbols represent level 1 cache, level 2 cache, level 3 cache and main memory, respectively. This option lets the programmer tune instruction scheduling for an expected memory access pattern (or hit rate.)

The table below summarizes execution times for `dilate.c` compiled under a number of different options.

Compilation command	User time	System time	Elapsed time
`cc dilate.c`	9.845 sec	0.010 sec	9.855 sec
`cc -O2 dilate.c`	4.546 sec	0.005 sec	4.551 sec
`cc -O2 -mcpu=ev6 dilate.c`	3.814 sec	0.011 sec	3.825 sec
`cc -O2 -mcpu=ev6 -funroll-all-loops dilate.c`	3.396 sec	0.009 sec	3.405 sec
`cc -O3 -mcpu=ev6 -funroll-all-loops dilate.c`	3.376 sec	0.007 sec	3.383 sec

Compiled code for the third result (`-mcpu=ev6 -O2`) is shown below. The number of instructions in the innermost loop produced by GNU C is 20 while the innermost loop generated by the Compaq C compiler under similar conditions (`-arch ev6`) has 12 instructions. The GNU C version executes in 3.814 seconds (user time) versus 1.990 seconds for the Compaq C version. The loop produced by GNU C performs more address arithmetic at runtime.

`DilatePixel` - GNU C compiler

```
DilatePixel:
        .frame $30,0,$26,0
        ldgp $29,0($27)
$DilatePixel..ng:
        .prologue 1
        lda $23,-32767          $23 = the_max
        bis $31,$31,$22         Clear i ($24 = i)
        lda $25,A               Load base address of A[][]
        lda $28,B               Load base address of B[][]
        bis $31,$31,$24         Clear internal induction variable
        .align 4
$L90:                           Top of outer loop
        bis $31,$31,$7          Clear j ($7 = j)
        addq $24,$28,$8         Compute address of B[r][0]
        .align 4
$L94:                           Top of inner loop
        addl $22,$16,$2
        ldwu $3,0($8)           Load B
        addl $7,$17,$4
        sll $2,4,$1
        addq $4,$4,$4
        addl $7,1,$7            Increment j
        addq $8,2,$8            Advance pointer to B[r][j]
        addq $1,$2,$1
        cmple $7,30,$6          Compare j with 30
        sll $1,4,$1
```

```
sextw $3,$5           Sign extend item from B
subq $1,$2,$1
s4addq $1,$4,$1
addq $1,$25,$1        Form address of item in A
ldwu $3,0($1)         Load A
sextw $3,$2           Sign extend item from A
addq $2,$5,$2         Compute the Sum
cmple $2,$23,$1       Compare Sum with the_max
cmoveq $1,$2,$23      Condition move to the_max
bne $6,$L94           Bottom of inner loop
addl $22,1,$22        Increment i
addq $24,62,$24       Advance pointer to next row of B
cmple $22,30,$1       Compare i with 30
bne $1,$L90           Bottom of outer loop
sll $16,4,$1
lda $3,Z
addq $17,$17,$2
addq $1,$16,$1
sll $1,4,$1
subq $1,$16,$1
s4addq $1,$2,$1
addq $1,$3,$1
stw $23,0($1)         Store Z[r][c]
ret $31,($26),1       Return
```

---

4. Conclusion

We've made several suggestions for achieving higher levels of performance when programming in C on Alpha. We focused mainly on integer computation and should mention here that there are techniques for improving the performance of floating point performance as well. Broadly, these techniques include: trading off accuracy for faster execution speed, non-strict handling of error cases, and non-strict handling of denormalized numbers, NaNs and infinities. Indeed, even much more can be said about pointer aliasing and its effect on compiler optimizations. The references below include many Alpha-specific and generic resources on program optimization.

References

Alpha Architecture Committee, *Alpha Architecture Reference Manual (Third Edition)*, Butterworth-Heinemann, Woburn, MA, 1998.

Compaq Computer Corporation, *Alpha Architecture Handbook,* Order number EC-QD2KC-TE, October 1998.

Compaq Computer Corporation, *Alpha 21064 and Alpha 21064A Microprocessors Hardware Reference Manual,* Order number EC-Q9ZUC-TE, June 1996.

Compaq Computer Corporation, *Alpha 21164 Hardware Reference Manual,* Order number EC-QP97D-TE,

March 1997.

Compaq Computer Corporation, *Alpha 21264 Hardware Reference Manual,* Order number EC-RJRZA-TE, July 1999.

Compaq Computer Corporation, *Compiler Writer's Guide for the Alpha 21264,* Order number EC-RJ66A-TE, June 1999.

Compaq Computer Corporation, *Linux Alpha - Power tools from Compaq* .

Compaq Computer Corporation, *Compaq C for Tru64 UNIX,* July 1999.

Compaq Computer Corporation, *Tru64 UNIX Programmer's Guide,* August 1999.

Isom Crawford and Kevin Wadleigh, *Software Optimization for High Performance Computing: Creating Faster Applications*, Prentice Hall, May 2000.

Kevin Dowd and Charles Severance, *High Performance Computing (Second Edition)*, O'Reilly & Associates, 1993.

Addendum A - More tips and techniques

The following tips and techniques are from the *Compaq Tru64 Programmer's Guide*.

If you are willing to modify your application, use profiling tools to determine where your application spends most of its time. Many applications spend most of their time in a few routines. Concentrate your efforts on improving the speed of those heavily used routines.

After you identify the heavily used portions of your application, consider the algorithms used by that code. Is it possible to replace a slow algorithm with a more efficient one? Replacing a slow algorithm with a faster one often produces a larger performance gain than tweaking an existing algorithm.

The following sections identify performance opportunities involving data types, I/O handling, cache usage and data alignment, and general coding issues.

A.1 Data Type Considerations

The smallest unit of efficient access on Alpha systems is 32 bits. Accessing an 8- or 16-bit scalar can result in a sequence of machine instructions to access the data. A 32- or 64-bit data item can be accessed with a single, efficient machine instruction.

If performance is a critical concern, avoid using integer and logical data types that are less than 32 bits, especially for scalars that are used frequently. In C programs, consider replacing `char` and `short` declarations with `int` and `long` declarations.

Division of integer quantities is slower than division of floating-point quantities. If possible, consider replacing such integer operations with equivalent floating-point operations.

Integer division operations are not native to the Alpha processor and must be emulated in software, so they can be slow. Other non-native operations include transcendental operations (for example, sine and cosine) and square root, although square root is supported by the FIX extension.

A.2 Cache Usage and Data Alignment Considerations

If your application has a few heavily used data structures, try to allocate those data structures on cache line boundaries in the secondary cache. Doing so can improve the efficiency of your application's use of cache. See Appendix A of the Alpha Architecture Reference Manual for additional information.

Look for potential data cache collisions between heavily used data structures. Such collisions occur when the distance between two data structures allocated in memory is equal to the size of the primary (internal) data cache. If your data structures are small, you can avoid this by allocating them contiguously in memory. You can use the uprofile tool to determine the number of cache collisions and their locations. See Appendix A of the Alpha Architecture Reference Manual for additional information on data cache collisions.

Data alignment can also affect performance. By default, the C compiler aligns each data item on its natural boundary; that is, it positions each data item so that its starting address is an even multiple of the size of the data type used to declare it. Data not aligned on natural boundaries is called misaligned data. Misaligned data can slow performance because it forces the software to make necessary adjustments (fix-ups) at run time.

In C programs, misalignment can occur when you type cast a pointer variable from one data type to a larger data type; for example, type casting a `char` pointer (1-byte alignment) to an int pointer (4-byte alignment) and then dereferencing the new pointer may cause unaligned access. Also in C, creating packed structures using the #pragma pack directive can cause unaligned access.

To correct alignment problems in C programs, you can use the `-align` option or you can make necessary modifications to the source code. If instances of misalignment are required by your program for some reason, use the `__unaligned` data-type qualifier in any pointer definitions that involve the misaligned data. When data is accessed through the use of a pointer declared `__unaligned`, the compiler generates the additional code necessary to copy or store the data without generating alignment errors. (Alignment errors have a much more costly impact on performance than the additional code that is generated.)

Warning messages identifying misaligned data are not issued during the compilation of C programs.

During execution of any program, the Tru64 UNIX kernel issues warning messages ("unaligned access") for most instances of misaligned data. The messages include the program counter (PC) value for the address of the instruction that caused the misalignment.

On Tru64, you can use either of the following two methods to access code that causes the unaligned access fault:

- By using a debugger to examine the PC value presented in the "unaligned access" message, you can find the routine name and line number for the instruction causing the misalignment. (In some cases, the "unaligned access" message results from a pointer passed by a calling routine. The return address register (ra) contains the address of the calling routine -- if the contents of the register have not been changed by the called routine.)
- By turning off the `-align` option on the command line and running your program in a debugger session, you can examine your program's stack and variables at the point where the debugger stops due to the unaligned access.

For additional information on data alignment, see Appendix A in the Alpha Architecture Reference Manual. See ccc(1) for details on alignment-control options that you can specify on compilation command lines.

A.3 General Coding Considerations

Use libc functions (for example: strcpy, strlen, strcmp, bcopy, bzero, memset, memcpy) instead of writing similar routines or your own loops. These functions are hand coded for efficiency.

Use the `unsigned` data type for variables wherever possible because:

- The variable is always greater than or equal to zero, which enables the compiler to perform optimizations that would not otherwise be possible, and
- The compiler generates fewer instructions for all unsigned divide operations.

Consider the following example:

```
int long i;
unsigned long j;
...
return i/2 + j/2;
```

In the example, `i/2` is an expensive expression; however, `j/2` is inexpensive.

The compiler generates three instructions for the signed `i/2` operations:

```
cmplt   $1,  0, $2
addq    $1, $2, $1
sra     $1,  1, $1
```

The compiler generates only one instruction for the unsigned `j/2` operation:

```
srl     $3,  1, $3
```

Also, consider using the `-unsigned` option to treat all `char` declarations as `unsigned char`.

If your application uses large amounts of data for a short period of time, consider allocating the data dynamically with the `malloc` function instead of declaring it statically. When you have finished using the memory, free it so it can be used for other data structures later in your program. Using this technique to reduce the total memory usage of your application can substantially increase the performance of applications running in an environment in which physical memory is a scarce resource.

If an application uses the `malloc` function extensively, you may be able to improve the application's performance (processing speed, memory utilization, or both) by using `malloc`'s control variables to tune memory allocation. See malloc(3) for details.

If your application uses local arrays whose sizes are unknown at compile time, you can gain a performance advantage by allocating them with the `alloca` function, which uses very few instructions and is very efficient. Storage allocated by the `alloca` function is automatically reclaimed when an exit is made from the routine in which the allocation is made.

The `alloca` function allocates space on the stack, not the heap, so you must make sure that the object being allocated does not exhaust all of the free stack space. If the object does not fit in the stack, a core dump is issued.

Programs that issue calls to the `alloca` function should include the `alloca.h` header file. If the header file is not included, the program will execute properly, but it will run much slower.

Minimize type casting, especially type conversion from integer to floating point and from a small data type to a larger data type.

To avoid cache misses, make sure that multidimensional arrays are traversed in natural storage order; that is, in row major order with the rightmost subscript varying fastest and striding by one. Avoid column major order, which is used by Fortran.

If your application fits in a 32-bit address space and allocates large amounts of dynamic memory by allocating structures that contain many pointers, you may be able to save significant amounts of memory by using the -xtaso option. To use the option, you must modify your source code with a C-language pragma that controls pointer size allocations.

Do not use indirect calls in C programs (that is, calls that use routines or pointers to functions as arguments). Indirect calls introduce the possibility of changes to global variables. This effect reduces the amount of optimization that can be safely performed by the optimizer.

Use functions to return values instead of reference parameters.

Use `do while` instead of `while` or `for` whenever possible. With `do while`, the optimizer does not have to duplicate the loop condition in order to move code from within the loop to outside the loop.

Use local variables and avoid global variables. Declare any variable outside of a function as `static`, unless that variable is referenced by another source file. Minimizing the use of global variables increases optimization opportunities for the compiler.

Use value parameters instead of reference parameters or global variables. Reference parameters have the same degrading effects as pointers.

Write straightforward code. For example, do not use `++` and `--` operators within an expression. When you use these operators for their values instead of their side-effects, you often get bad code. For example, the following coding is not recommended:

```
while (n--)
  {
  ...
  }
```

The following coding is recommended:

```
while (n != 0)
  {
  n--;
  ...
  }
```

Avoid taking and passing addresses (that is, & values). Using & values can create aliases, make the optimizer store variables from registers to their home storage locations, and significantly reduce optimization opportunities.

Avoid creating functions that take a variable number of arguments. A function with a variable number of arguments causes the optimizer to unnecessarily save all parameter registers on entry.

Declare functions as `static` unless the function is referenced by another source module. Use of static functions allows the optimizer to use more efficient calling sequences.

Also, avoid aliases where possible by introducing local variables to store dereferenced results. (A dereferenced result is the value obtained from a specified address.) Dereferenced values are affected by indirect operations and calls, whereas local variables are not; local variables can be kept in registers. The example below shows how the proper placement of pointers and the elimination of aliasing enable the compiler to produce better code.

```
int len = 10;
char a[10];

void zero()
  {
  char *p;
  for (p = a; p != a + len; ) *p++ = 0;
  }
```

Consider the use of pointers in the example above. Because the statement `*p++=0` might modify `len`, the compiler must load it from memory and add it to the address of `a` on each pass through the loop, instead of computing `(a + len)` in a register once outside the loop.

Two different methods can be used to increase the efficiency of the code used in the example above.

Method #1: Use subscripts instead of pointers. As shown in the following example, the use of subscripting in the `azero` procedure eliminates aliasing; the compiler keeps the value of `len` in a register, saving two instructions, and still uses a pointer to access `a` efficiently, even though a pointer is not specified in the source code:

```
char a[10];
int len;

void azero()
  {
  int i;
  for (i = 0; i != len; i++) a[i] = 0;
  }
```

Method #2: Use local variables. As shown in the following example, specifying `len` as a local variable or formal argument ensures that aliasing cannot take place and permits the compiler to place `len` in a register:

```
char a[10];

void lpzero(len)
  int len;
  {
```

```
char *p;
for (p = a; p != a + len; ) *p++ = 0;
}
```

Addendum B - Alpha assembler language

This section is by no means a thorough introduction to Alpha assembler language, but it should provide enough information to read the examples in this paper.

Most Alpha instructions fall into one of three categories: computational, load/store and control. A computational instruction usually takes two arguments, performs an operation on the argument values and writes the result. The arguments and result are stored in general registers. Data flows in computational instructions from left to right. So, the add instruction below:

```
addq $2,$3,$16
```

adds the two quadwords in registers 2 and 3, and places the result in register 16. The second argument may also be a short constant.

The set of operations that can be performed by the computational instructions is rich. Operations include arithmetic, logic, shifts, insertion, extraction, masking, comparison and of course, floating point operations.

Load and store instructions read and write memory values, of course. Both load and store instructions form the effective address in the same way -- adding an offset with the contents of a register. Data in store instructions flows from left to right, so the store instruction:

```
stq $18,40($3)
```

stores the data in register 18 to the effective address formed by the sum of the constant 40 and the contents of register 3. Load instructions have a similar syntax, but data flows from right to left. The following load instruction:

```
ldq $18,40($3)
```

loads a quadword from the memory location at the effective address to register 18.

Control instructions include unconditional branch (`BR`), conditional branch (`BEQ`, `BGE`, etc.), branch to subroutine (`BSR`), and jump instructions (`JMP`, `JSR`, `RET`, `JMP_COROUTINE`.) All control instructions specify the target of the control transfer. Conditional branches also have a register argument whose value is compared against zero to form the branch condition.

The table below summarizes the Alpha instructions that appeared in the examples. This is just a small subset of the complete instruction set. More complete and detailed information can be found in the Alpha Architecture Reference Manual.

Mnemonic	Description
addl	Add longword (32-bits)
addq	Add quadword (64-bits)
and	Logical product (AND)

bis	Logical sum (OR or "bit set")
bne	Branch if not equal to zero
clr	Clear integer register (pseudo-op)
cmoveq	Conditional move if equal to zero
cmplt	Compare signed quadword less than
extlh	Extract longword high
extll	Extract longword low
extqh	Extract quadword high
extql	Extract quadword low
inswl	Insert word low
lda	Load address
ldah	Load address high
ldbu	Load zero-extended byte from memory to register
ldl	Load sign-extended longword from memory to register
ldq	Load quadword from memory to register
ldwu	Load zero-extended word from memory to register
ldq_u	Load unaligned quadword
mov	Move literal/register (pseudo-op)
mskwl	Mask word low
nop	No operation
ret	Return from subroutine
s4addq	Scaled quadword add by 4
s8addq	Scaled quadword add by 8
s8subq	Scaled quadword subtract by 8
sextl	Sign extended longword (pseudo-op)
sextw	Sign extended word
sra	Shift right arithmetic
subl	Subtract longwords
stb	Store byte from register to memory
stl	Store longword from register to memory
stq	Store quadword from register to memory
stw	Store word from register to memory
unop	No operation

Addendum C - whatami.c: Linux / GCC version

```
/*
 * whatami.c - Linux/GNU C version
```

```
 * Display the Alpha microprocessor family and its features
 *
 * Author: Paul J. Drongowski
 *         Alpha Technology Solutions Group
 *         550 King Street
 *         Littleton, MA 01460
 * Date:   8 September 2000
 *
 * This program is based on original code by Steve Hofmann.
 *
 * Please consult the Alpha Architecture Reference Manual for
 * any new additions.
 */

#include <stdio.h>

/*
 * Use Alpha Linux code lifted from .../include/asm-alpha/system.h
 */

/*
 * IMPLVER value assignments
 *
 *    Value   Meaning
 *      0     21064 (EV4)
 *            21064A (EV45)
 *            21066a/21068A (LCA45)
 *      1     21164 (EV5)
 *            21164A (EV56)
 *            21164PC (PCA56)
 *      2     21264 (EV6)
 */

enum implver_enum {
        IMPLVER_EV4,
        IMPLVER_EV5,
        IMPLVER_EV6
};

#define implver()                               \
({ unsigned long __implver;                     \
   __asm__ ("implver %0" : "=r"(__implver));    \
   (enum implver_enum) __implver; })

enum amask_enum {
        AMASK_BWX = (1UL << 0),
        AMASK_FIX = (1UL << 1),
        AMASK_MVI = (1UL << 2),
        AMASK_MAX = (1UL << 8),
        AMASK_PRECISE_TRAP = (1UL << 9),
};
```

```
#define amask(mask)                                                 \
({ unsigned long __amask, __input = (mask);                         \
   __asm__ ("amask %1,%0" : "=r"(__amask) : "rI"(__input));         \
   __amask; })

main()
{
  int my_implver ;
  int my_amask ;
  int definedbits = 0;

  my_implver = implver() ;
  my_amask = amask(-1) ;

  switch( my_implver ) {
    case IMPLVER_EV4: {
      printf("Compaq Alpha 21064 or variant (EV4 core)\n") ;
      break ;
    }
    case IMPLVER_EV5: {
      printf("Compaq Alpha 21164 or variant (EV5 core)\n") ;
      break ;
    }
    case IMPLVER_EV6: {
      printf("Compaq Alpha 21264 or variant (EV6 core)\n") ;
      break ;
    }
  }

  /*
   * An amask bit is clear if the feature is present
   */
  printf("Alpha architectural features/extensions\n") ;

  if ( !~my_amask ) printf("  None\n") ;
  definedbits |= 1UL<<0 ;
  if ( ~my_amask & 1LU<<0 ) printf("  BWX byte-word extension\n") ;
  definedbits |= 1UL<<1 ;
  if ( ~my_amask & 1UL<<1 ) printf("  FIX floating point extension\n") ;
  definedbits |= 1UL<<2 ;
  if ( ~my_amask & 1UL<<2 ) printf("  CIX count extension\n") ;
  definedbits |= 1UL<<8 ;
  if ( ~my_amask & 1UL<<8 ) printf("  MVI multimedia extension\n") ;
  definedbits |= 1UL<<9 ;
  if ( ~my_amask & 1UL<<9 ) printf("  Precise arithmetic trap reporting\n") ;

  if ( ~my_amask & ~definedbits ) printf("  Unknown bit(s)/feature\n") ;

  return( 1 ) ;
}
```

Addendum D - whatami.c: Tru64 UNIX / Compaq C version

```
/*
 * whatami_tru.c -- Tru64 UNIX version / Compaq C compiler
 * Display the Alpha microprocessor family and its features
 *
 * Author: Paul Drongowski
 *         110 Spit Brook Road
 *         Nashua, NH 03062
 * Date:   8 September 2000 (revised 11 April 2001)
 *
 * This program is based on original code by Steve Hofmann.
 *
 * Please consult the Alpha Architecture Reference Manual for
 * any new additions.
 */

#include <stdio.h>
#include <c_asm.h>
#include <stdio.h>
#include <sys/sysinfo.h>
#include <machine/hal_sysinfo.h>

/*
 * IMPLVER value assignments
 *
 *    Value   Meaning
 *      0     21064 (EV4)
 *            21064A (EV45)
 *            21066a/21068A (LCA45)
 *      1     21164 (EV5)
 *            21164A (EV56)
 *            21164PC (PCA56)
 *      2     21264 (EV6)
 */

enum implver_enum {
        IMPLVER_EV4,
        IMPLVER_EV5,
        IMPLVER_EV6
};

enum amask_enum {
        AMASK_BWX = (1UL << 0),
        AMASK_FIX = (1UL << 1),
        AMASK_MVI = (1UL << 2),
        AMASK_MAX = (1UL << 8),
        AMASK_PRECISE_TRAP = (1UL << 9)
};

main()
```

```
{
  int my_implver ;
  int my_amask ;
  int definedbits = 0;

  my_implver = asm("implver %r0;") ;
  my_amask = asm("amask %a0,%r0;", -1);

  switch( my_implver ) {
    case IMPLVER_EV4: {
      printf("Compaq Alpha 21064 or variant (EV4 core)\n") ;
      break ;
    }
    case IMPLVER_EV5: {
      printf("Compaq Alpha 21164 or variant (EV5 core)\n") ;
      break ;
    }
    case IMPLVER_EV6: {
      printf("Compaq Alpha 21264 or variant (EV6 core)\n") ;
      break ;
    }
  }

  /*
   * An amask bit is clear if the feature is present
   */
  printf("Alpha architectural features/extensions\n") ;

  if ( !~my_amask ) printf("  None\n") ;
  definedbits |= 1UL<<0 ;
  if ( ~my_amask & 1LU<<0 ) printf("  BWX byte-word extension\n") ;
  definedbits |= 1UL<<1 ;
  if ( ~my_amask & 1UL<<1 ) printf("  FIX floating point extension\n") ;
  definedbits |= 1UL<<2 ;
  if ( ~my_amask & 1UL<<2 ) printf("  CIX count extension\n") ;
  definedbits |= 1UL<<8 ;
  if ( ~my_amask & 1UL<<8 ) printf("  MVI multimedia extension\n") ;
  definedbits |= 1UL<<9 ;
  if ( ~my_amask & 1UL<<9 ) printf("  Precise arithmetic trap reporting\n") ;

  if ( ~my_amask & ~definedbits ) printf("  Unknown bit(s)/feature\n") ;

  return( 1 ) ;
}
```

Disclaimer: The execution times cited in this technical note are solely intended to compare the relative merits of certain coding practices, optimizations, etc. Your mileage may vary.

Special thanks are due to Mike Burrows at the Compaq Systems Research Center for his careful review, suggestions and corrections. Sharon Smith and Mark Davis also provide some excellent suggestions. Thanks are certainly due to the authors whose work I have used in writing this paper, especially Steve Root.