David P. Hunter
Eric B. Betts

# Measured Effects of Adding Byte and Word Instructions to the Alpha Architecture

**The performance of an application can be expressed as the product of three variables: (1) the number of instructions executed, (2) the average number of machine cycles required to execute a single instruction, and (3) the cycle time of the machine. The recent decision to add byte and word manipulation instructions to the DIGITAL Alpha Architecture has an effect upon the first of these variables. The performance of a commercial database running on the Windows NT operating system has been analyzed to determine the effect of the addition of the new byte and word instructions. Static and dynamic analysis of the new instructions' effect on instruction counts, function calls, and instruction distribution have been conducted. Test measurements indicate an increase in performance of 5 percent and a decrease of 4 to 7 percent in instructions executed. The use of prototype Alpha 21164 microprocessor-based hardware and instruction tracing tools showed that these two measurements are due to the use of the Alpha Architecture's new instructions within the application.**

The Alpha Architecture and its initial implementations were limited in their ability to manipulate data values at the byte and word granularity. Instead of allowing single instructions to manipulate byte and word values, the original Alpha Architecture required as many as sixteen instructions. Recently, DIGITAL extended the Alpha Architecture to manipulate byte and word data values with a single instruction. The second generation of the Alpha 21164 microprocessor, operating at 400 megahertz (MHz) or greater, is the first implementation to include the new instructions.

This paper presents the results of an analysis of the effects that the new instructions in the Alpha Architecture have on the performance, code size, and dynamic instruction distribution of a consistent execution path through a commercial database. To exercise the database, we modified the Transaction Processing Performance Council's (TPC) obsolete TPC-B benchmark. Although it is no longer a valid TPC benchmark, the TPC-B benchmark, along with other TPC benchmarks, has been widely used to study database performance.[1–5]

We began our project by rebuilding Microsoft Corporation's SQL Server product to use the new Alpha instructions. We proceeded to conduct a static code analysis of the resulting images and dynamic link libraries (DLLs). The focus of the study was to investigate the impact that the new instructions had upon a large application and not their impact upon the operating system. To this end, we did not rebuild the Windows NT operating system to use the new byte and word instructions.

We measured the dynamic effects by gathering instruction and function traces with several profiling and image analysis tools. The results indicate that the Microsoft SQL Server product benefits from the additional byte and word instructions to the Alpha microprocessor. Our measurements of the images and DLLs show a decrease in code size, ranging from negligible to almost 9 percent. For the cached TPC-B transactions, the number of instructions executed per transaction decreased from 111,288 to 106,521 (a 4 percent reduction). For the scaled TPC-B transactions, the number of instructions executed per

transaction decreased from 115,895 to 107,854 (a 7 percent reduction).

The rest of this paper is divided as follows: we begin with a brief overview of the Alpha Architecture and its introduction of the new byte and word manipulation instructions. Next, we describe the hardware, software, and tools used in our experiments. Lastly, we provide an analysis of the instruction distribution and count.

## Alpha Architecture

The Alpha Architecture is a 64-bit, load and store, reduced instruction set computer (RISC) architecture that was designed with high performance and longevity in mind. Its major areas of concentration are the processor clock speed, the multiple instruction issue, and multiple processor implementations. For a detailed account of the Alpha Architecture, its major design choices, and overall benefits, see the paper by R. Sites.[6] The original architecture did not define the capability to manipulate byte- and word-level data with a single instruction. As a result, the first three implementations of the Alpha Architecture, the 21064, the 21064A, and the 21164 microprocessors, were forced to use as many as sixteen additional instructions to accomplish this task. The Alpha Architecture was recently extended to include six new instructions for manipulating data at byte and word boundaries. The second implementation of the 21164 family of microprocessors includes these extensions.

The first implementation of the Alpha Architecture, the 21064 microprocessor, was introduced in November 1992. It was fabricated in a 0.75-micrometer (μm) complementary metal-oxide semiconductor (CMOS) process and operated at speeds up to 200 MHz. It had both an 8-kilobyte (KB), direct-mapped, write-through, 32-byte line instruction cache (I-cache) and data cache (D-cache). The 21064 microprocessor was able to issue two instructions per clock cycle to a 7-stage integer pipeline or a 10-stage floating-point pipeline.[7] The second implementation of the 21064 generation was the Alpha 21064A microprocessor, introduced in October 1993. It was manufactured in a 0.5-μm CMOS process and operated at speeds of 233 MHz to 275 MHz. This implementation increased the size of the I-cache and D-cache to 16 KB. Various other differences exist between the two implementations and are outlined in the product data sheet.[8]

The Alpha 21164 microprocessor was the second-generation implementation of the Alpha Architecture and was introduced in October 1994. It was manufactured in a 0.5-μm CMOS technology and has the ability to issue four instructions per clock cycle. It contains a 64-entry data translation buffer (DTB) and a 48-entry instruction translation buffer (ITB) compared to the 21064A microprocessor's 32-entry DTB and 12-entry ITB. The chip contains three on-chip caches. The level one (L1) caches include an 8-KB, direct-mapped I-cache and an 8-KB, dual-ported, direct-mapped, write-through D-cache. A third on-chip cache is a 96-KB, three-way set-associative, write-back mixed instruction and data cache. The floating-point pipeline was reduced to nine stages, and the CPU has two integer units and two floating-point execution units.[9]

## The Exclusion of Byte and Word Instructions

The original Alpha Architecture intended that operations involved in loading or storing aligned bytes and words would involve sequences as given in Tables 1 and 2.[10] As many as 16 additional instructions are required to accomplish these operations on unaligned data. These same operations in the MIPS Architecture involve only a single instruction: LB, LW, SB, and SW.[11] The MIPS Architecture also includes single instructions to do the same for unaligned data. Given a situation in which all other factors are consistent, this would appear to give the MIPS Architecture an advantage in its ability to reduce the number of instructions executed per workload.

Sites has presented several key Alpha Architecture design decisions.[6] Among them is the decision not to include byte load and store instructions. Key design assumptions related to the exclusion of these features include the following:

- The majority of operations would involve naturally aligned data elements.

**Table 1**
Loading Aligned Bytes and Words on Alpha

| | **Load and Sign Extend a Byte** |
|---|---|
| LDL | R1, D.lw(Rx) |
| EXTBL | R1, #D.mod, R1 |
| | **Load and Zero Extend a Byte** |
| LDL | R1, D.lw(Rx) |
| SLL | R1, #56-8*D.mod, R1 |
| SRA | R1, #56, R1 |
| | **Load and Sign Extend a Word** |
| LDL | R1, D.lw(Rx) |
| EXTWL | R1, #D.mod, R1 |
| | **Load and Zero Extend a Word** |
| LDL | R1, D.lw(Rx) |
| SLL | R1, #48-8*D.mod, R1 |
| SRA | R1, #48, R1 |

**Table 2**
Storing Aligned Bytes and Words on Alpha

| **Store a Byte** | |
|---|---|
| LDL | R1, D.lw(Rx) |
| INSBL | R5,#D.mod, R3 |
| MSKBL | R1, #D.mod, R1 |
| BIS | R3, R1, R1 |
| STL | R1, D.1w(Rx) |

| **Store a Word** | |
|---|---|
| LDL | R1, D.lw(Rx) |
| INSWL | R5,#D.mod, R3 |
| MSKWL | R1, #D.mod, R1 |
| BIS | R3, R1, R1 |
| STL | R1, D.1w(Rx) |

- In the best possible scheme for multiple instruction issue, single byte and write instructions to memory are not allowed.
- The addition of byte and write instructions would require an additional byte shifter in the load and store path.

These factors indicated that the exclusion of specific instructions to manipulate bytes and words would be advantageous to the performance of the Alpha Architecture.

The decision not to include byte and word manipulation instructions is not without precedents. The original MIPS Architecture developed at Stanford University did not have byte instructions.[12] Hennessy et al. have discussed a series of hardware and software trade-offs for performance with respect to the MIPS processor.[13] Among those trade-offs are reasons for not including the ability to do byte addressing operations. Hennessy et al. argue that the additional cost of including the mechanisms to do byte addressing was not justified. Their studies showed that word references occur more frequently in applications than do byte references. Hennessy et al. conclude that to make a word-addressed machine feasible, special instructions are required for inserting and extracting bytes. These instructions are available in both the MIPS and the Alpha Architectures.

## Reversing the Byte and Word Instructions Decision

During the development of the Alpha Architecture, DIGITAL supported two operating systems, OpenVMS and ULTRIX. The developers had as a goal the ability to maintain both customer bases and to facilitate their transitions to the new Alpha microprocessor-based machines. In 1991, Microsoft and DIGITAL began work on porting Microsoft's new operating system, Windows NT, to the Alpha platform. The Windows NT operating system had strong links to the Intel x86 and the MIPS Architectures, both of which included instructions for single byte and word manipulation.[14] This strong connection influenced the Microsoft developers and independent software vendors (ISVs) to favor those architectures over the Alpha design.

Another factor contributed to this issue: the majority of code being run on the new operating system came from the Microsoft Windows and MS-DOS environments. In designing software applications for these two environments, the manipulation of data at the byte and word boundary is prevalent. With the Alpha microprocessor's inability to accomplish this manipulation in a single instruction, it suffered an average of 3:1 and 4:1 instructions per workload on load and store operations, respectively, compared to those architectures with single instructions for byte and word manipulation.

To assist in running the ISV applications under the Windows NT operating system, a new technology was needed that would allow 16-bit applications to run as if they were on the older operating system. Microsoft developed the Virtual DOS Machine (VDM) environment for the Intel Architecture and the Windows-on-Windows (WOW) environment to allow 16-bit Windows applications to work. For non-Intel architectures, Insignia developed a VDM environment that emulated an Intel 80286 microprocessor-based computer. Upon examining this emulator more closely, DIGITAL found opportunities for improving performance if the Alpha Architecture had single byte and word instructions.

Based upon this information and other factors, a corporate task force was commissioned in March 1994 to investigate improving the general performance of Windows NT running on Alpha machines. The further DIGITAL studied the issues, the more convincing the argument became to extend the Alpha Architecture to include single byte and word instructions.

This reversal in position on byte and word instructions was also seen in the evolution of the MIPS Architecture. In the original MIPS Architecture developed at Stanford University, there were no load or store byte instructions.[12] However, for the first commercially produced chip of the MIPS Architecture, the MIPS R2000 RISC processor, developers added instructions for the loading and storing of bytes.[11] One reason for this choice stemmed from the challenges posed by the UNIX operating system. Many implicit byte assumptions inside the UNIX kernel caused performance problems. Since the operating system being implemented was UNIX, it made sense to add the byte instructions to the MIPS Architecture.[15]

In June 1994, one of the coarchitects of the Alpha Architecture, Richard Sites, submitted an Engineering

Change Order (ECO) for the extension of the architecture to include byte and word instructions. It was speculated at the time that an increase of as much as 4 percent in overall performance would be achieved using the new instructions. In June 1995, six new instructions were added to the Alpha Architecture. The new instructions are outlined in Table 3. The first implementation to include support for the new instructions was the second generation of the Alpha 21164 microprocessor series. This reimplementation of the first Alpha 21164 design was manufactured in a 0.35-μm CMOS process and was introduced in October 1995.

### *Testing Environment*

We set up tests to measure the performance of equipment with and without the new instructions. To conduct our experiments, we used prototype hardware that included the second-generation Alpha 21164 microprocessor, and we devised a method to enable and disable the new instructions in hardware. At the same time, we investigated the projected performance of the software emulation mechanism to execute the new instructions on older processors. Finally, we built two separate versions of the Microsoft SQL Server application, one that used the new instructions and one that did not. For the purposes of discussing the different scenarios under study, we summarize the three execution schemes in Table 4. We use the associated nomenclature given there in the rest of this paper. In the remainder of this section, we describe each of the hardware, software, compiler, and analysis tools.

### *Prototype Hardware*

As previously mentioned, our machine was capable of operating with and without the new instructions. By using the same machine, we were able to minimize effects that could be introduced from variations in machine designs or processor families that could cause an increase in the executed code path through the operating system. All experiments were run on a prototype of the AlphaStation 500 workstation that was based upon the second-generation 21164 microprocessor operating at 400 MHz. (The AlphaStation 500 is a family of high-performance, mid-range graphics workstations.) The prototype was configured with 128 megabytes (MB) of memory and a single, 4-gigabyte (GB) fast-wide-differential (FWD) small computer systems interface (SCSI-2) disk.

**Table 3**
New Byte and Word Manipulation Instructions

| Mnemonic | Opcode | Function |
|---|---|---|
| stb | 0E | Store byte from register to memory |
| stw | 0D | Store word from register to memory |
| ldbu | 0A | Load zero-extended byte from memory to register |
| ldwu | 0C | Load zero-extended word from memory to register |
| sextb | 1C.0000 | Sign extend byte |
| sextw | 1C.0001 | Sign extend word |

**Table 4**
Three Methods for Execution of the New Instructions

| Nomenclature | Description |
|---|---|
| Original | Compiled with instructions that can execute on all Alpha implementations |
| Byte/Word | Compiled using the new instructions that will execute on second-generation 21164 implementations at full speed |
| Emulation | Compiled with new instructions and emulated through software |

New firmware allowed us to alternate between direct hardware execution and software emulation of the new byte and word instructions. We modified the Advanced RISC Consortium (ARC) code to allow us to switch between the two firmware versions through a simple power-cycle utility, called the fail-safe loader.[16] When the machine is powered on, it loads code from a serial read-only memory (SROM) storage device. This code then loads the ARC firmware from nonvolatile flash ROM. The fail-safe loader allowed the ARC firmware to be loaded into physical memory and not into the flash ROM. The new firmware was initialized by a reset of the processor and was executed as if it were loaded from the flash ROM. When the machine was turned off and then back on, the version of firmware that was stored in nonvolatile memory was loaded and executed.

### *Operating System*

We used a beta copy of the Microsoft Windows NT version 4.0 operating system. We chose this operating system for its capability to allow us to examine the impact of emulating the new byte and word instructions in the operating system.

By default, version 4.0 of the Windows NT operating system disables the trap and emulation capability for the new instructions. This approach is similar to the one Windows NT provides for the Alpha microprocessor to handle unaligned data references. For testing purposes, we enabled and disabled the trap and emulation capability of the new instructions. When this option is enabled, the operating system treats each new instruction listed in Table 3 as an illegal instruction and emulates the instruction. The trap and emulate strategy takes approximately 5 to 7 microseconds

per emulated instruction. When it is disabled or not present, the action taken depends upon the hardware support for the new instructions. If disabled in hardware, the instruction is treated as an illegal instruction; if enabled, it is executed like any other instruction.

### Microsoft SQL Server

To observe the effects of the new instructions, we chose the Microsoft SQL Server, a relational database management system (RDBMS) for the Windows NT operating system. Microsoft SQL Server was engineered to be a scalable, multiplatform, multithreaded RDBMS, supporting symmetric multiprocessing (SMP) systems. It was designed specifically for distributed client-server computing, data warehousing, and database applications on the Internet.

In an earlier investigation, Sites and Perl present a profile of the Microsoft SQL Server running the TPC-B benchmark.[4] They identify the executables and DLLs that are involved in running the benchmark and break down the percentage of time that each contributes to the benchmark. Their results, summarized in Figure 1, show that only a few SQL Server executables and DLLs were heavily exercised during the benchmark. After verifying these results with the SQL Server development group at Microsoft, we decided to rebuild only the images and DLLs identified in Figure 1 to use the new byte and word instructions.

Table 5 lists the executables and DLLs that we modified and their correlation to the ones identified by Sites and Perl. The variations exist because of name changes of DLLs or the use of a different network protocol. We changed network protocols for performance reasons. Sites and Perl used an early version of the Microsoft SQL Server version 6.0, in which the fastest network transport available at that time was Named Pipes. In the final release of SQL Server version 6.0 and subsequent versions of the product, the Transmission Control Protocol/Internet Protocol (TCP/IP) replaced Named Pipes in this category. Based upon this, we rebuilt the libraries associated with TCP/IP instead of those associated with Named Pipes. Other networking libraries, such as those for DECnet and Internetwork Packet Exchange/Sequenced Packet Exchange (IPX/SPX), were not rebuilt.

**Table 5**
Images and DLLs Modified for the Microsoft SQL Server

| Sites DLL/EXE | V6.0 DLL/EXE | Function |
|---|---|---|
| sqlserver.exe | sqlservr.exe | SQL Server Main Executable |
| ntwdblib.dll | ntwdblib.dll | Network Communications Library |
| opends50.dll | opends60.dll | Open Data Services Networking Library |
| dbnmpntw.dll | N/A | V4.21A Client Side Named Pipes Library |
| ssnmpntw.dll | N/A | V4.21A Named Pipes Library |
| N/A | dbmssocn.dll | V6.5 Client Side TCP/IP Library |
| N/A | ssmsso60.dll | V6.5 Netlibs TCP/IP Library |

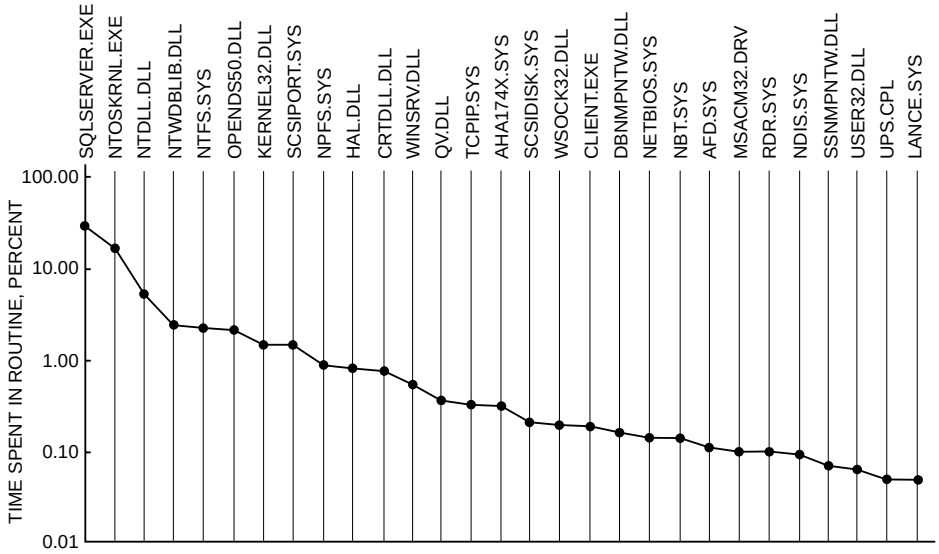


**Figure 1**
Images/DLLs Involved in a TPC-B Transaction for Microsoft SQL Server Based on Sites and Perl's Analysis

## Compiling Microsoft SQL Server to Use the New Instructions

Our goal was to measure only the effects introduced by using the new instructions and not effects introduced by different versions or generations of compilers. Therefore, we needed to find a way to use the same version of a compiler that differed only in its use or nonuse of the new instructions. To do this, we used a compiler option available on the Microsoft Visual C++ compiler. This switch, available on all RISC platforms that support Visual C++, allows the generation of optimized code for a specific processor within a processor family while maintaining binary compatibility with all processors in the processor family. Processor optimizations are accomplished by a combination of specific code-pattern selection and code scheduling. The default action of the compiler is to use a blended model, resulting in code that executes equally well across all processors within a platform family.

Using this compiler option, we built two versions of the aforementioned images within the SQL Server application, varying only their use of the code-generation switch. The first version, referred to as the Original build, was built without specifying an argument for the code-generation switch. The second one, referred to as Byte/Word, set the switch to generate code patterns using the new byte and word manipulation instructions. All other required files came from the SQL Server version 6.5 Beta II distribution CD-ROM.

### *The Benchmark*

The benchmark we chose was derived from the TPC-B benchmark. As previously mentioned, the TPC-B benchmark is now obsolete; however, it is still useful for stressing a database and its interaction with a computer system. The TPC-B benchmark is relatively easy to set up and scales readily. It has been used by both database vendors and computer manufacturers to measure the performance of either the computer system or the actual database. We did not include all the required metrics of the TPC-B benchmark; therefore, it is not in full compliance with published guidelines of the TPC. We refer to it henceforth simply as the application benchmark.

The application benchmark is characterized by significant disk I/O activity, moderate system and application execution time, and transaction integrity. The application benchmark exercises and measures the efficiency of the processor, I/O architecture, and RDBMS. The results measure performance by indicating how many simulated banking transactions can be completed per second. This is defined as transactions per second (tps) and is the total number of committed transactions that were started and completed during the measurement interval.

The application benchmark can be run in two different modes: cached and scaled. The cached, or in-memory mode, is used to estimate the system's maximum performance in this benchmark environment. This is accomplished by building a small database that resides completely in the database cache, which in turn fits within the system's physical random-access memory (RAM). Since the entire database resides in memory, all I/O activity is eliminated with the exception of log writes. Consequently, the benchmark only performs one disk I/O for each transaction, once the entire database is read off the disk and into the database cache. The result is a representation of the maximum number of tps that the system is capable of sustaining.

The scaled mode is run using a bigger database with a larger amount of disk I/O activity. The increase in disk I/O is a result of the need to read and write data to locations that are not within the database cache. These additional reads and writes add extra disk I/Os. The result is normally characterized as having to do one read and one write to the database and a single write to the transaction log for each transaction. The combination of a larger database and additional I/O activity decreases the tps value from the cached version. Based upon our previous experience running this benchmark, the scaled benchmark can be expected to reach approximately 80 percent of the cached performance.

For the scaled tests, we built a database sized to accommodate 50 tps. This was less than 80 percent of the maximum tps produced by the cached results. We chose this size because we were concentrating on isolating a single scaled transaction under a moderate load and not under the maximum scaled performance possible.

### *Image Tracing and Analysis Tools*

Collecting only static measurements of the executables and DLLs affected was insufficient to determine the applicability of the new instructions. We collected the actual instruction traces of SQL Server while it executed the application benchmark. Furthermore, we decided that the ability to trace the actual instructions being executed was more desirable than developing or extending a simulator. To obtain the traces, we needed a tool that would allow us to

- Collect both system- and user-mode code.
- Collect function traces, which would allow us to align the starting and stopping points of different benchmark runs.
- Work without modifying either the application or the operating system.

In the past, the only tool that would provide instruction traces under the Windows NT operating system was the debugger running in single-step mode.

Obtaining traces through either the ntsd or the windbg debugger is quite limited due to the following problems:

- The tracing rate is only about 500 instructions per second. This is far too slow to trace anything other than isolated pieces of code.
- The trace fails across system calls.
- The trace loops infinitely in critical section code.
- Register contents are not easily displayed for each instruction.
- Real-time analysis of instruction usage and cache misses are not possible.

Instruction traces can also be obtained using the PatchWrks trace analysis tool.[4] Although this tool operates with near real-time performance and can trace instructions executing in kernel mode, it has the following limitations:

- It operates only on a DIGITAL Alpha AXP personal computer.
- It requires an extra 40 MB of memory.
- All images to be traced must be patched, thus slightly distorting text addresses and function sizes.
- Successive runs of application code are not repeatable due to unpredictable kernel interrupt behavior (the traces are too accurate).

The solution was Ntstep, a tool that can trace user-mode instruction execution of any image in the Windows NT/Alpha environment through an innovative combination of breakpointing and "Alpha-on-Alpha" emulation. It has the ability to trace a program's execution at rates approaching a million instructions per second. Ntstep can trace individual instructions, loads, stores, function calls, I-cache and D-cache misses, unaligned data accesses, and anything else that can be observed when given access to each instruction as it is being executed. It produces summary reports of the instruction distribution, cache line usage, page usage (working set), and cache simulation statistics for a variety of Alpha systems.

Ntstep acts like a debugger that can execute single-step instructions except that it executes instructions using emulation instead of single-step breakpoints whenever possible. In practice, emulation accounts for the majority of instructions executed within Ntstep. Since a single-step execution of an instruction with breakpoints takes approximately 2 milliseconds and emulation of an Alpha instruction requires only 1 or 2 microseconds, Ntstep can trace approximately 1,000 times faster than a debugger. Unlike most emulators, the application executes normally in its own address space and environment.

## Results

We collected data on three different experiments. In the first investigation, we looked at the relative performance of the three different versions of the Microsoft SQL Server outlined in Table 4. We compared the three variations using the cached version of the application benchmark.

In the second experiment, we observed how the new instructions affect the instruction distribution in the static images and DLLs that we rebuilt. We compared the Byte/Word versions to the Original versions of the images and DLLs. We also attempted to link the differences in instruction counts to the use of the new instructions.

Lastly, we investigated the variation between the Original and the Byte/Word versions with respect to instruction distribution on the scaled version of the benchmark. This comparison was based upon the code path executed by a single transaction.

### *Cached Performance*

In the first experiments, we compared the relative performance impact of using the new instructions. We chose to measure performance of only the cached version of the application benchmark because the I/O subsystem available on the prototype of the AlphaStation 500 was not adequate for a full-scaled measurement. We ensured that the database was fully cached by using a ramp-up period of 60 seconds and a ramp-down period of 30 seconds. This was verified as steady state by observing that the SQL Server buffer cache hit ratio remained at or above 95 percent. The measurement period for the benchmark was 60 seconds. We ran the benchmark several times and took the average tps for each of the three variations outlined in Table 4.

The results of the three schemes are as follows: 444 tps for the Original version, 460 tps for the Byte/Word version, and 116 tps for the Emulation version. The new instructions contributed a 3.5 percent gain in performance. The impact of emulating the instructions is a loss of 73.9 percent of the potential performance.

### *Static Instruction Counts*

To analyze the mixture of instructions in the images and DLLs, we disassembled each image and DLL in the Original and Byte/Word versions. We then looked at only those instructions that exhibited a difference between the two versions within the images or DLLs. The variations in instruction counts of these are shown in Table 6.

To examine the images more closely, we disassembled each image and DLL and collected counts of code

**Table 6**
Instruction Deltas (Normal Minus Byte/Word) for the SQL Server Images and DLLs

| Instruction | dbmssocn.dll | ntwdblib.dll | opends60.dll | sqlservr.exe | ssmsso60.dll | Instruction | dbmssocn.dll | ntwdblib.dll | opends60.dll | sqlservr.exe | ssmsso60.dll |
|---|---|---|---|---|---|---|---|---|---|---|---|
| lda | 0 | −3 | −247 | −8524 | −4 | xor | 0 | 0 | −2 | +119 | 0 |
| ldah | 0 | 0 | −27 | 18−18 | 0 | sll | 0 | 0 | +2 | −2359 | 0 |
| ldl | −9 | −11 | −597 | −13133 | −46 | sra | 0 | 0 | −15 | −3534 | −4 |
| ldq | 0 | 0 | −29 | −2980 | 0 | srl | 0 | 0 | 0 | −295 | 0 |
| ldq_l | 0 | 0 | 0 | −9 | 0 | cmpbge | 0 | 0 | −1 | −18 | 0 |
| ldq_u | −10 | −2 | −311 | −8529 | −18 | mskbl | −3 | −1 | −196 | −3647 | −8 |
| stl | −5 | −11 | −278 | −7932 | −11 | mskwl | 0 | −5 | −41 | −1604 | 0 |
| stb | +3 | +1 | +216 | +3969 | +7 | zapnot | −5 | 0 | −115 | −2135 | −33 |
| stw | +2 | +5 | +59 | +2798 | +3 | addl | 0 | 0 | 0 | −8 | 0 |
| stq | 0 | 0 | −4 | −53 | 0 | addq | 0 | 0 | 0 | +3 | 0 |
| stq_c | 0 | 0 | 0 | −9 | 0 | s4addl | 0 | 0 | 0 | −4 | 0 |
| beq | 0 | 5 | +1 | −1236 | 0 | cmovge | 0 | 0 | 0 | +1 | 0 |
| bge | 0 | 0 | 0 | +8 | 0 | cmovne | 0 | 0 | 0 | +2 | 0 |
| bgt | 0 | 0 | 0 | +3 | 0 | cmovlt | 0 | 0 | 0 | −1 | 0 |
| blbc | 0 | 0 | −1 | −19 | 0 | cmovlbc | 0 | 0 | 0 | −2 | 0 |
| blbs | 0 | 0 | 0 | −4 | 0 | callsys | 0 | 0 | 0 | 0 | 0 |
| blt | 0 | 0 | 0 | 0 | 0 | extqh | 0 | 0 | −14 | −426 | −4 |
| bne | 0 | 0 | +1 | +24 | 0 | ldwu | +4 | 0 | +193 | +6320 | +35 |
| br | 0 | −4 | +1 | −1120 | 0 | ldbu | +9 | +3 | +464 | +10231 | +18 |
| bsr | 0 | 0 | 0 | −6 | 0 | mull | 0 | 0 | 0 | +1 | 0 |
| ret | 0 | 0 | +4 | +15 | 0 | subl | 0 | 0 | +1 | +6 | 0 |
| cmpeq | 0 | 0 | 0 | +9 | 0 | subq | 0 | 0 | 0 | +3 | 0 |
| cmplt | 0 | 0 | 0 | +15 | 0 | insll | 0 | 0 | 0 | 1−1 | 0 |
| cmple | 0 | 0 | 0 | +5 | 0 | inswl | −2 | −3 | −54 | −2647 | −3 |
| cmpult | 0 | 0 | −1 | 1−1 | 0 | call_pal | +2 | +1 | +1 | +161 | 0 |
| cmpule | 0 | −5 | −2 | 1183–1183 | 0 | extlh | 0 | 0 | 0 | −14 | 0 |
| and | −2 | −6 | −364 | −6435 | −8 | insbl | −2 | −1 | −135 | −3163 | −6 |
| bic | −3 | −11 | −287 | −7242 | −8 | extll | 0 | 0 | 0 | −20 | 0 |
| bis | −4 | −7 | −208 | −7097 | −9 | extbl | −10 | −6 | −367 | −10656 | −14 |
| ornot | 0 | 0 | 0 | +4 | 0 | extwl | −1 | 0 | −84 | −324 | −1 |

size, the number of functions, the number and type of new byte and word instructions, and lastly, nop and trapb instructions. The results are presented in Tables 7 through 10.

We expected that the instructions used to manipulate bytes and words in the original Alpha Architecture (Tables 1 and 2) would decrease proportionally to the usage of the new instructions. These assumptions held true for all the images and DLLs that used the new instructions. For example, in the original Alpha Architecture, the instructions MSKBL and MSKWL are used to store a byte and word, respectively. In the sqlservr.exe image, these two instructions showed a decrease of 3,647 and 1,604 instructions, respectively. Compare this with the corresponding addition of 3,969 STB and 2,798 STW instructions in the same image. Looking further into the sqlservr.exe image, we also saw that 10,231 LDBU instructions were used and the usage of the EXTBL instruction was reduced by 10,656. Although these numbers do not correlate on a one-for-one basis, we believe this is due to other usage of these instructions. Other usage might include the compiler scheme for introducing the new instructions in places where it used an LDL or an LDQ in the Original image.

Of the rebuilt images and DLLs, sqlservr.exe and opends60.dll showed the most variations, with the new instructions making up 3.73 percent and 3.9 percent of these files. The most frequently occurring new instruction was ldbu, followed by ldwu. The least-used instructions were sextb and sextw. The size of the images was reduced in three out of five images. The image size reduction ranged from negligible to just over 4 percent. In all cases, the size of the code section was reduced and ranged from insignificant to approximately 8.5 percent. There was no change in the number of functions in any of the files.

### *Dynamic Instruction Counts*

We gathered data from the application benchmark running in both cached and scaled modes. We ran at least one iteration of the benchmark test prior to gathering trace data to allow both the Windows NT operating system and the Microsoft SQL Server database to reach a steady state of operation on the system under test (SUT). Steady state was achieved when the SQL Server cache-hit ratio reached 95 percent or greater, the number of transactions per second was constant, and the CPU utilization was as close to 100 percent as possible. The traces were gathered over a sufficient

**Table 7**
Byte/Word Images and DLLs

| Image/DLL | Total File Bytes | Total Text Bytes | Total Code Bytes | Number of Functions | Total Byte/ Word | % Byte/ Word | LDBU Count | LDBU % | LDWU Count | LDWU % | STB Count | STB % | STW Count | STW % | SEXTB Count | SEXTB % | SEXTW Count | SEXTW % | Total NOPs | Total TRAPB |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| sqlservr.exe | 8053624 | 2981148 | 2884776 | 3364 | 26869 | 3.73 | 10231 | 38.077 | 6320 | 23.5215 | 3969 | 14.7717 | 2798 | 10.4135 | 139 | 0.517325 | 3412 | 12.6986 | 5929 | 2219 |
| dbmssocn.dll | 13824 | 5884 | 5520 | 13 | 18 | 1.3 | 9 | 50 | 4 | 22.2222 | 3 | 16.6667 | 2 | 11.1111 | 0 | 0 | 0 | 0 | 21 | 0 |
| ntwdblib.dll | 318464 | 246316 | 231688 | 429 | 9 | 0.02 | 3 | 33.333 | 0 | 0 | 1 | 11.1111 | 5 | 55.5556 | 0 | 0 | 0 | 0 | 767 | 10 |
| opends60.dll | 212992 | 104204 | 97240 | 243 | 948 | 3.9 | 464 | 48.945 | 193 | 20.3586 | 216 | 22.7848 | 59 | 6.22363 | 9 | 0.949367 | 7 | 0.738397 | 391 | 128 |
| ssmsso60.dll | 70760 | 9884 | 9128 | 19 | 67 | 2.94 | 18 | 26.866 | 35 | 52.2388 | 7 | 10.4478 | 3 | 4.47761 | 4 | 5.97015 | 0 | 0 | 25 | 0 |

**Table 8**
Original Build of Images and DLLs

| Image/DLL | Total File Bytes | Total Text Bytes | Total Code Bytes | Number of Functions | Total Byte/ Word | % Byte/ Word | LDBU Count | LDBU % | LDWU Count | LDWU % | STB Count | STB % | STW Count | STW % | SEXTB Count | SEXTB % | SEXTW Count | SEXTW % | Total NOPs | Total TRAPB |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| sqlservr.exe | 8337248 | 3264108 | 3153480 | 3364 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6207 | 2252 |
| dbmssocn.dll | 13824 | 6012 | 5656 | 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 16 | 0 |
| ntwdblib.dll | 318464 | 246620 | 231904 | 429 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 770 | 10 |
| opends60.dll | 222720 | 114012 | 105536 | 243 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 405 | 128 |
| ssmsso60.dll | 71284 | 10300 | 9424 | 19 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 18 | 0 |

**Table 9**
Numerical Differences of Original Minus Byte/Word Images and DLLs

| Image/DLL | Total File Bytes | Total Text Bytes | Total Code Bytes | Number of Functions | Total Byte/ Word | % Byte/ Word | LDBU Count | LDBU % | LDWU Count | LDWU % | STB Count | STB % | STW Count | STW % | SEXTB Count | SEXTB % | SEXTW Count | SEXTW % | Total NOPs | Total TRAPB |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| lsqlservr.exe | −283624 | −282960 | −268704 | 0 | +26869 | +4 | +10231 | +38 | +6320 | +24 | +3969 | +15 | +2798 | +10 | +139 | +1 | +3412 | +13 | −278 | −33 |
| dbmssocn.dll | 0 | −128 | −136 | 0 | +18 | +1 | +9 | +50 | +4 | +22 | +3 | +17 | +2 | +11 | 0 | 0 | 0 | 0 | +5 | 0 |
| ntwdblib.dll | 0 | −304 | −216 | 0 | +9 | 0 | +3 | +33 | 0 | 0 | +1 | +11 | +5 | +56 | 0 | 0 | 0 | 0 | −3 | 0 |
| opends60.dll | −9728 | −9808 | −8296 | 0 | +948 | +4 | +464 | +49 | +193 | +20 | +216 | +23 | +59 | +6 | +9 | +1 | +7 | +1 | −14 | 0 |
| ssmsso60.dll | −524 | −416 | −296 | 0 | +67 | +3 | +18 | +27 | +35 | +52 | +7 | +10 | +3 | +4 | +4 | +6 | 0 | 0 | +7 | 0 |

**Table 10**
Percentage Variation of Original Minus Byte/Word Images and DLLs

| Image/DLL | Total File Bytes | Total Text Bytes | Total Code Bytes | Number of Functions | Total Byte/ Word | % Byte/ Word | LDBU Count | LDBU % | LDWU Count | LDWU % | STB Count | STB % | STW Count | STW % | SEXTB Count | SEXTB % | SEXTW Count | SEXTW % | Total NOPs | Total TRAPB |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| sqlservr.exe | −3.402% | −8.669% | −8.521% | 0.000% | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | −4.479% | −1.465% |
| dbmssocn.dll | 0.000% | −2.129% | −2.405% | 0.000% | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | +31.250% | N/A |
| ntwdblib.dll | 0.000% | −0.123% | −0.093% | 0.000% | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | −0.390% | 0.000% |
| opends60.dll | −4.368% | −8.603% | −7.861% | 0.000% | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | −3.457% | 0.000% |
| ssmsso60.dll | −0.735% | −4.039% | −3.141% | 0.000% | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | +38.889% | N/A |

period of time to ensure that we captured several transactions. The traces were then edited into separate individual transactions. The geometric mean was taken from the resulting traces and used for all subsequent analysis.

We used Ntstep to gather complete instruction and function traces of both versions of the SQL Server database while it executed the application benchmark. Figure 2 shows an example output for an instruction trace, and Figure 3 shows an example output for a function trace from Ntstep. Since Ntstep can attach to a running process, we allowed the application benchmark to achieve steady state prior to data collection. This approach ensured that we did not see the effects of warming up either the machine caches or the SQL Server database cache. Each instruction trace consisted of approximately one million instructions, which was sufficient to cover multiple transactions. The data was

```
 0  ** Breakpoint (Pid 0xd1, Tid 0xb2) SQLSERVR.EXE pc 77f39b34
 0  ** Trace begins at 242698
    opends60!FetchNextCommand
 1  00242698: 23deffb0   lda        sp, -50(sp)         // sp now 72bff00
 2  0024269c: b53e0000   stq        s0, 0(sp)           // @072bff00 = 148440
 3  002426a0: b55e0008   stq        s1, 8(sp)           // @072bff08 = 0
 4  002426a4: b57e0010   stq        s2, 10(sp)          // @072bff10 = 5
 5  002426a8: b59e0018   stq        s3, 18(sp)          // @072bff18 = 1476a8
 6  002426ac: b5be0020   stq        s4, 20(sp)          // @072bff20 = 2c4
 7  002426b0: b5de0028   stq        s5, 28(sp)          // @072bff28 = 41
 8  002426b4: b5fe0030   stq        fp, 30(sp)          // @072bff30 = 0
 9  002426b8: b75e0038   stq        ra, 38(sp)          // @072bff38 = 242398
10  002426bc: 47f00409   bis        zero, a0, s0        // s0 now 148440
11  002426c0: 47f1040a   bis        zero, a1, s1        // s1 now 72bffa0
12  002426c4: 47f2040b   bis        zero, a2, s2        // s2 now 72bffa8
13  002426c8: d3404e67   bsr        ra, 00256068        // ra now 2426cc
    opends60!netI0ReadData
14  00256068: 23deffa0   lda        sp, -60(sp)         // sp now 72bfea0
15  0025606c: 43f10002   addl       zero, a1, t1        // t1 now 72bffa0
16  00256070: b53e0000   stq        s0, 0(sp)           // @072bfea0 = 148440
17  00256074: b55e0008   stq        s1, 8(sp)           // @072bfea8 = 72bffa0
18  00256078: b57e0010   stq        s2, 10(sp)          // @072bfeb0 = 72bffa8
19  0025607c: b59e0018   stq        s3, 18(sp)          // @072bfeb8 = 1476a8
20  00256080: b5be0020   stq        s4, 20(sp)          // @072bfec0 = 2c4
21  00256084: b5de0028   stq        s5, 28(sp)          // @072bfec8 = 41
22  00256088: b5fe0030   stq        fp, 30(sp)          // @072bfed0 = 0
23  0025608c: b75e0038   stq        ra, 38(sp)          // @072bfed8 = 2426cc
24  00256090: a1d01140   ldl        s5, 1140(a0)        // @00149580 1479e8
25  00256094: 47f00409   bis        zero, a0, s0        // s0 now 148440
26  00256098: a1f001d0   ldl        fp, 1d0(a0)         // @00148610 dbba0
27  0025609c: 47e0340d   bis        zero, #1, s4        // s4 now 1
28  002560a0: a0620000   ldl        t2, 0(t1)           // @072bffa0 155c58
29  002560a4: b23e004c   stl        a1, 4c(sp)          // @072bfeec = 72bffa0
30  002560a8: b25e0050   stl        a2, 50(sp)          // @072bfef0 = 72bffa8
31  002560ac: b27e0054   stl        a3, 54(sp)          // @072bfef4 = 1476a8
32  002560b0: e460001d   beq        t2, 00256128        // (t2 is 155c58)
33  002560b4: 220303e0   lda        a0, 3e0(t2)         // a0 now 156038
34  002560b8: 47f00404   bis        zero, a0, t3        // t3 now 156038
35  002560bc: 63ff4000   mb                             //
36  002560c0: 47e03400   bis        zero, #1, v0        // v0 now 1
37  002560c4: a8240000   ldl_l      t0, 0(t3)           // @00156038 0
38  002560c8: b8040000   stl_c      v0, 0(t3)           // @00156038 = 1
39  002560cc: e40000b6   beq        v0, 002563a8        // (v0 is 1)
40  002560d0: 63ff4000   mb                             //
41  002560d4: e4200001   beq        t0, 002560dc        // (t0 is 0)
    opends60!netI0ReadData+0x74:
42  002560dc: a1be004c   ldl        s4, 4c(sp)          // @072bfeec 72bffa0
43  002560e0: a00d0000   ldl        v0, 0(s4)           // @072bffa0 155c58
44  002560e4: a04003dc   ldl        t1, 3dc(v0)         // @00156034 0
45  002560e8: 20800404   lda        t3, 404(v0)         // t3 now 15605c
46  002560ec: 405f05a2   cmpeq      t1, zero, t1        // t1 now 1
47  002560f0: e4400003   beq        t1, 00256100        // (t1 is 1)
48  002560f4: a0600404   ldl        t2, 404(v0)         // @0015605c 15605c
49  002560f8: 406405a3   cmpeq      t2, t3, t2          // t2 now 1
50  002560fc: 47e30402   bis        zero, t2, t1        // t1 now 1
51  00256100: 47e2040d   bis        zero, t1, s4        // s4 now 1
```

**Figure 2**
Example of Instruction Trace Output from Ntstep

```
52  00256104: e4400005   beq       t1, 0025611c       // (t1 is 1)
53  00256108: a0a00000   ldl       t4, 0(v0)          // @00155c58 204200
54  0025610c: 24df0080   ldah      t5, 80(zero)       // t5 now 800000
55  00256110: 48a07625   zapnot    t4, #3, t4         // t4 now 4200
56  00256114: 40a60005   addl      t4, t5, t4         // t4 now 804200
57  00256118: b0a00000   stl       t4, 0(v0)          // @00155c58 = 804200
58  0025611c: a0fe004c   ldl       t6, 4c(sp)         // @072bfeec 72bffa0
59  00256120: a0e70000   ldl       t6, 0(t6)          // @072bffa0 155c58
60  00256124: b3e703e0   stl       zero, 3e0(t6)      // @00156038 = 0
61  00256128: e5a00061   beq       s4, 002562b0       // (s4 is 1)
62  0025612c: 257f0026   ldah      s2, 26(zero)       // s2 now 260000
63  00256130: 216b62f8   lda       s2, 62f8(s2)       // s2 now 2662f8
64  00256134: 5fff041f   cpys      f31, f31, f31      //
65  00256138: a21e0054   ldl       a0, 54(sp)         // @072bfef4 1476a8
66  0025613c: 225e0040   lda       a2, 40(sp)         // a2 now 72bfee0
67  00256140: a00b0000   ldl       v0, 0(s2)          // @002662f8 77e985a0
68  00256144: 227e0048   lda       a3, 48(sp)         // a3 now 72bfee8
69  00256148: a23e0050   ldl       a1, 50(sp)         // @072bfef0 72bffa8
70  0025614c: 47ef0414   bis       zero, fp, a4       // a4 now dbba0
71  00256150: a2100000   ldl       a0, 0(a0)          // @001476a8 2c0
72  00256154: 6b404000   jsr       ra, (v0),0         // ra now 256158
    KERNEL32!GetQueuedCompletionStatus:
73  77e985a0: 23deffc0   lda       sp, -40(sp)        // sp now 72bfe60
74  77e985a4: b53e0000   stq       s0, 0(sp)          // @072bfe60 = 148440
75  77e985a8: b55e0008   stq       s1, 8(sp)          //@072bfe68 = 72bffa0
76  77e985ac: b57e0010   stq       s2, 10(sp)         // @072bfe70 = 2662f8
77  77e985b0: b59e0018   stq       s3, 18(sp)         // @072bfe78 = 1476a8
78  77e985b4: b75e0020   stq       ra, 20(sp)         // @072bfe80 = 256158
79  77e985b8: 47f00409   bis       zero, a0, s0       // s0 now 2c0
80  77e985bc: 47f1040a   bis       zero, a1, s1       // s1 now 72bffa8
81  77e985c0: 47f2040b   bis       zero, a2, s2       // s2 now 72bfee0
82  77e985c4: 47f3040c   bis       zero, a3, s3       // s3 now 72bfee8
83  77e985c8: 47f40411   bis       zero, a4, a1       // a1 now dbba0
84  77e985cc: 221e0038   lda       a0, 38(sp)         // a0 now 72bfe98
85  77e985d0: d3405893   bsr       ra, 77eae820       // ra now 77e985d4
```

**Figure 2 (continued)**
Example of Instruction Trace Output from Ntstep

then reduced to a series of single transactions and analyzed for instruction distribution. For both the cached- and the scaled-transaction instruction counts, we combined at least three separate transactions and took the geometric mean of the instructions executed, which caused slight variations in the instruction counts. All resulting instruction counts were within an acceptable standard deviation as compared to individual transaction instruction counts.

We collected the function traces in a similar fashion. Once the application benchmark was at a steady state, we began collecting the function call tree. Based on previous work with the SQL Server database and consultation with Microsoft engineers, we could pinpoint the beginning of a single transaction. We then began collecting samples for both traces at the same instant, using an Ntstep feature that allowed us to start or stop sample collection based upon a particular address.

The dynamic instruction counts for both the scaled and the cached transactions are given in Tables 11 and 12. We also show the variation and percentage variation between the Original and the Byte/Word versions of the SQL Server. Two of the six new instructions, sextb and sextw, are not present in the Byte/Word trace. The remaining four instructions combine to make up 2.6 percent and 2.7 percent of the instructions executed per scaled and cached transaction, respectively. Other observations include the following:

- The number of instructions executed decreased 7 percent for scaled and 4 percent for cached transactions.
- The number of ldl_l/stl_c sequences decreased 3 percent for scaled transactions.
- All the instructions that are identified in Tables 1 and 2 show a decrease in usage. Not surprisingly, the instructions mskwl and mskbl completely disappeared. The inswl and insbl instructions decreased by 47 percent and 90 percent, respectively. The sll instruction decreased by 38 percent, and the sra instruction usage decreased by 53 percent. These reductions hold true within 1 to 2 percent for both scaled and cached transactions.
- The instructions ldq_u and lda, which are used in unaligned load and store operations, show a decrease in the range of 20 to 22 percent and 15 to 16 percent, respectively.

```
    0   **   Breakpoint (Pid 0xd7, Tid 0xdb) SQLSERVR.EXE pc 77f39b34
    0   **   Trace begins at 00242698
    0   **   . opends60!FetchNextCommand
   13   **   . . opends60!netIOReadData
   72   **   . . . KERNEL32!GetQueuedCompletionStatus
   85   **   . . . . KERNEL32!BaseFormatTimeOut
   99   **   . . . . ntdll!NtRemoveIoCompletion
  129   **   . . . opends60!netIOCompletionRoutine
  272   **   . . opends60!netIORequestRead
  285   **   . . . KERNEL32!ResetEvent
  290   **   . . . . ntdll!NtClearEvent
  318   **   . . . SSNMPN60!*0x06a131f0*
  348   **   . . . . KERNEL32!ReadFile
  399   **   . . . . . ntdll!NtReadFile
  412   **   . . . . . KERNEL32!BaseSetLastNTError
  417   **   . . . . . . ntdll!RtlNtStatusToDosError
  423   **   . . . . . . . ntdll!RtlNtStatusToDosErrorNoTeb
  509   **   . . . . KERNEL32!GetLastError
  560   **   . opends60!get_client_event
  665   **   . . opends60!processRPC
  682   **   . . . opends60!unpack_rpc
  749   **   . opends60!execute_event
  762   **   . . opends60!execute_sqlserver_event
  802   **   . . . opends60!unpack_rpc
  864   **   . . . SQLSERVR!execrpc
  911   **   . . . . KERNEL32!WaitForSingleObjectEx
  937   **   . . . . . KERNEL32!BaseFormatTimeOut
  950   **   . . . . . ntdll!NtWaitForSingleObject
 1024   **   . . . . SQLSERVR!UserPerfStats
 1038   **   . . . . . KERNEL32!GetThreadTimes
 1055   **   . . . . . . ntdll!NtQueryInformationThread
 1173   **   . . . . SQLSERVR!init_recvbuf
 1208   **   . . . . SQLSERVR!init_sendbuf
 1227   **   . . . . SQLSERVR!port_ex_handle
 1263   **   . . . . SQLSERVR!_0tssetjmp3
 1313   **   . . . . SQLSERVR!memalloc
 1365   **   . . . . . SQLSERVR!_0tsZero
 1405   **   . . . . SQLSERVR!recvhost
 1437   **   . . . . . SQLSERVR!_0tsMove
 1500   **   . . . . SQLSERVR!memalloc
 1577   **   . . . . SQLSERVR!rn_char
 1580   **   . . . . . SQLSERVR!recvhost
 1612   **   . . . . . . SQLSERVR!_0tsMove
 1777   **   . . . . SQLSERVR!parse_name
 1808   **   . . . . . SQLSERVR!dbcs_strnchr
 2115   **   . . . . SQLSERVR!rpcprot
 2131   **   . . . . . SQLSERVR!memalloc
 2183   **   . . . . . . SQLSERVR!_0tsZero
 2252   **   . . . . . SQLSERVR!getprocid
 2319   **   . . . . . . SQLSERVR!procrelink+0x1250
 2546   **   . . . . . . . SQLSERVR!_0tsRemainder32
 2559   **   . . . . . . . SQLSERVR!_0tsDivide32+0x94
 2597   **   . . . . . . SQLSERVR!opentable
 2642   **   . . . . . . . SQLSERVR!parse_name
 2673   **   . . . . . . . . SQLSERVR!dbcs_strnchr
 2979   **   . . . . . . . SQLSERVR!parse_name
 3010   **   . . . . . . . . SQLSERVR!dbcs_strnchr
 3323   **   . . . . . . . SQLSERVR!opentabid
 3363   **   . . . . . . . . SQLSERVR!getdes
 3493   **   . . . . . . . . SQLSERVR!GetRunidFromDefid+0x40
 3510   **   . . . . . . . . . SQLSERVR!_0tsZero
 3658   **   . . . . . . . SQLSERVR!initarg
 3668   **   . . . . . . . SQLSERVR!setarg
 3703   **   . . . . . . . . SQLSERVR!_0tsFieldInsert
 3764   **   . . . . . . . SQLSERVR!setarg
 3799   **   . . . . . . . . SQLSERVR!_0tsFieldInsert
 3857   **   . . . . . . . SQLSERVR!startscan
 3901   **   . . . . . . . . SQLSERVR!getindex2
 3978   **   . . . . . . . . . SQLSERVR!getkeepslot
 4064   **   . . . . . . . . . SQLSERVR!rowoffset
 4109   **   . . . . . . . . . SQLSERVR!rowoffset
 4170   **   . . . . . . . . . SQLSERVR!_0tsMove
 4331   **   . . . . . . . . . SQLSERVR!memcmp
 5323   **   . . . . . . . . . SQLSERVR!bufunhold
 5436   **   . . . . . . . . SQLSERVR!prepscan
 5550   **   . . . . . . . . . SQLSERVR!match_sargs_to_index
```

**Figure 3**
Example of Function Trace Output from Ntstep

```
 5828  **   . . . . . . . . . SQLSERVR!srchindex
 5895  **   . . . . . . . . . . SQLSERVR!getpage
 5942  **   . . . . . . . . . . . SQLSERVR!bufget
 5976  **   . . . . . . . . . . . . SQLSERVR!_OtsDivide
 5985  **   . . . . . . . . . . . . SQLSERVR!_OtsDivide32+0x94
 6090  **   . . . . . . . . . . . . SQLSERVR!getkeepslot
 6356  **   . . . . . . . . . . SQLSERVR!bufrlockwait
 6539  **   . . . . . . . . . . SQLSERVR!srchpage
 6720  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 6912  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 7309  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 7728  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 8125  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 8522  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 8919  **   . . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
 9410  **   . . . . . . . . . . SQLSERVR!index_beforesleep+0x100
 9465  **   . . . . . . . . . . . SQLSERVR!bufrunlock
 9641  **   . . . . . . . . . SQLSERVR!trim_sqoff+0xf0
 9661  **   . . . . . . . . . SQLSERVR!qualpage
 9809  **   . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
10212  **   . . . . . . . . . . SQLSERVR!nc__sqlhilo+0x8b0
10616  **   . . . . . . . . . . SQLSERVR!rowoffset
10702  **   . . . . . . . . SQLSERVR!getnext
10769  **   . . . . . . . . . SQLSERVR!_OtsFieldInsert
10822  **   . . . . . . . . . SQLSERVR!getrow2
10838  **   . . . . . . . . . . SQLSERVR!getpage
10885  **   . . . . . . . . . . . SQLSERVR!bufget
10919  **   . . . . . . . . . . . . SQLSERVR!_OtsDivide
10928  **   . . . . . . . . . . . . SQLSERVR!_OtsDivide32+0x94
11033  **   . . . . . . . . . . . . SQLSERVR!getkeepslot
11359  **   . . . . . . . . SQLSERVR!_OtsMove
11489  **   . . . . . . . . SQLSERVR!endscan
11557  **   . . . . . . . . . SQLSERVR!bufunkeep
11675  **   . . . . . . . . . SQLSERVR!bufunkeep
11853  **   . . . . . . . . SQLSERVR!closetable
11907  **   . . . . . . . . . SQLSERVR!endscan
12044  **   . . . . . . . . . SQLSERVR!get_spinlock
12103  **   . . . . . . . . SQLSERVR!opentabid
12138  **   . . . . . . . . . SQLSERVR!getdes
12291  **   . . . . . . . . . SQLSERVR!_OtsZero
12464  **   . . . . . . . SQLSERVR!closetable
12524  **   . . . . . . . . SQLSERVR!endscan
12661  **   . . . . . . . . SQLSERVR!get_spinlock
12729  **   . . . . . SQLSERVR!protect
12756  **   . . . . . . SQLSERVR!port_ex_handle
12792  **   . . . . . . SQLSERVR!_Otssetjmp3
12845  **   . . . . . . SQLSERVR!prot_search
12887  **   . . . . . . . SQLSERVR!dbtblfind
12958  **   . . . . . . SQLSERVR!check_protect
13025  **   . . . . SQLSERVR!memalloc
13077  **   . . . . . SQLSERVR!_OtsZero
13127  **   . . . . SQLSERVR!memalloc
13179  **   . . . . . SQLSERVR!_OtsZero
13263  **   . . . . SQLSERVR!rn_i2
13267  **   . . . . . SQLSERVR!recvhost
13299  **   . . . . . . SQLSERVR!_OtsMove
13369  **   . . . . SQLSERVR!recvhost
13401  **   . . . . . SQLSERVR!_OtsMove
13477  **   . . . . SQLSERVR!recvhost
13509  **   . . . . . SQLSERVR!_OtsMove
13562  **   . . . . SQLSERVR!recvhost
13594  **   . . . . . SQLSERVR!_OtsMove
13670  **   . . . . SQLSERVR!recvhost
13702  **   . . . . . SQLSERVR!_OtsMove
13755  **   . . . . SQLSERVR!recvhost
13787  **   . . . . . SQLSERVR!_OtsMove
13847  **   . . . . SQLSERVR!bconst
13895  **   . . . . . SQLSERVR!mkconstant
13921  **   . . . . . . SQLSERVR!memalloc
14046  **   . . . . . . SQLSERVR!memalloc
14098  **   . . . . . . . SQLSERVR!_OtsZero
14157  **   . . . . . SQLSERVR!rn_i4
14161  **   . . . . . . SQLSERVR!recvhost
14193  **   . . . . . . . SQLSERVR!_OtsMove
```

**Figure 3 (continued)**
Example of Function Trace Output from Ntstep

**Table 11**
Instruction Count and Variations for Scaled Transaction

| Instruction | Original | Byte/Word | Delta | % Delta | Instruction | Original | Byte/Word | Delta | % Delta |
|---|---|---|---|---|---|---|---|---|---|
| stb | 0 | 174 | +174 | N/A | stt | 334 | 334 | 0 | 0% |
| stw | 0 | 219 | +219 | N/A | cmple | 368 | 358 | 10 | −3% |
| ldwu | 0 | 1215 | +1215 | N/A | inswl | 390 | 207 | 183 | −47% |
| ldbu | 0 | 1216 | +1216 | N/A | srl | 457 | 398 | 59 | −13% |
| cmpbge | 2 | 0 | −2 | −100% | extqh | 441 | 317 | 124 | −28% |
| cmovlbs | 2 | 2 | 0 | 0% | cmpule | 468 | 450 | 18 | −4% |
| addt | 3 | 3 | 0 | 0% | cmpult | 563 | 518 | 45 | −8% |
| cmovlbc | 5 | 4 | −1 | −20% | cmplt | 565 | 534 | 31 | −5% |
| cmovle | 5 | 5 | 0 | 0% | rdteb | 604 | 597 | 7 | −1% |
| insqh | 6 | 6 | 0 | 0% | extwl | 660 | 345 | 315 | −48% |
| cmovgt | 13 | 13 | 0 | 0% | stq_u | 688 | 688 | 0 | 0% |
| callsys | 18 | 14 | −4 | −22% | blt | 784 | 771 | 13 | −2% |
| mulq | 13 | 13 | 0 | 0% | bic | 771 | 347 | 424 | −55% |
| s8subq | 17 | 17 | 0 | 0% | extll | 789 | 761 | 28 | −4% |
| cmovlt | 16 | 16 | 0 | 0% | extlh | 789 | 761 | 28 | −4% |
| ldt | 25 | 25 | 0 | 0% | bge | 828 | 819 | 9 | −1% |
| zap | 34 | 33 | −1 | −3% | mb | 961 | 941 | 20 | −2% |
| umulh | 35 | 35 | 0 | 0% | sll | 949 | 590 | 359 | −38% |
| mull | 60 | 62 | +2 | +3% | subl | 1052 | 1061 | (9) | +1% |
| ornot | 52 | 52 | 0 | 0% | br | 1160 | 1080 | 80 | −7% |
| cmpeq | 64 | 61 | −3 | −5% | sra | 1211 | 562 | 649 | −54% |
| insql | 61 | 61 | 0 | 0% | bsr | 1203 | 1191 | 12 | −1% |
| blbs | 69 | 69 | 0 | 0% | s4addl | 1176 | 1166 | 10 | −1% |
| s8addl | 71 | 74 | +3 | +4% | ret | 1282 | 1264 | 18 | −1% |
| mskwl | 74 | 0 | −74 | −100% | zapnot | 1262 | 910 | 352 | −28% |
| jsr | 98 | 89 | −9 | −9% | addq | 1704 | 1685 | 19 | −1% |
| cpys | 104 | 41 | −63 | −61% | subq | 2159 | 2140 | 19 | −1% |
| mskqh | 155 | 153 | −2 | −1% | ldah | 2793 | 2746 | 47 | −2% |
| cmovne | 147 | 141 | −6 | −4% | extbl | 2902 | 1668 | 1234 | −43% |
| mskbl | 163 | 0 | −163 | −100% | xor | 3426 | 3380 | 46 | −1% |
| cmoveq | 183 | 173 | −10 | −5% | and | 3402 | 2969 | 433 | −13% |
| insbl | 182 | 19 | −163 | −90% | bne | 4537 | 4440 | 97 | −2% |
| extwh | 196 | 196 | 0 | 0% | addl | 4897 | 4855 | 42 | −1% |
| trapb | 203 | 215 | +12 | +6% | ldq_u | 5046 | 3933 | 1113 | −22% |
| mskql | 204 | 202 | −2 | −1% | stl | 5753 | 5301 | 452 | −8% |
| jmp | 208 | 200 | −8 | −4% | lda | 6496 | 5435 | 1061 | −16% |
| cmovge | 291 | 287 | −4 | −1% | stq | 6778 | 6713 | 65 | −1% |
| blbc | 249 | 249 | 0 | 0% | ldq | 7018 | 6519 | −499 | +7% |
| bgt | 331 | 328 | −3 | −1% | beq | 7607 | 7455 | 152 | −2% |
| ldl_l | 344 | 335 | −9 | −3% | bis | 11284 | 10707 | 577 | −5% |
| stl_c | 344 | 335 | −9 | −3% | ldl | 15962 | 14260 | 1702 | −11% |
| extql | 329 | 327 | −2 | −1% | **Totals** | **115895** | **107854** | **8042** | **−7%** |

For the scaled transaction, a decrease in 58 out of 81 instructions types occurred. Of the remaining 25 instructions, 21 had no change and only 4 instructions, mull, s8addl, trapb, and subl, showed an increase. For cached transactions, 22 instruction counts decreased, 29 increased, and 22 remained unchanged.

The performance gain of 3.5 percent measured for the cached version of the application benchmark correlates closely to the decrease in the number of instructions per transaction measured in Table 13. If this correlation holds true, we would expect to see an increase in performance of approximately 7 percent for scaled transactions runs.

### *Dynamic Instruction Distribution*

The performance of the Alpha microprocessor using technical and commercial workloads has been evaluated.[1] The commercial workload used was debit-

**Table 12**
Instruction Count and Variations for Cached Transaction

| Instruction | Original | Byte/Word | Delta | % Delta | Instruction | Original | Byte/Word | Delta | % Delta |
|---|---|---|---|---|---|---|---|---|---|
| stb | 0 | 174 | 1174 | N/A | stt | 334 | 334 | 0 | 0% |
| stw | 0 | 217 | 1217 | N/A | cmple | 367 | 374 | +7 | +2% |
| ldwu | 0 | 1189 | +1189 | N/A | inswl | 381 | 203 | −178 | −47% |
| ldbu | 0 | 1333 | +1333 | N/A | srl | 433 | 383 | −50 | −12% |
| cmpbge | 2 | 0 | −2 | −100% | extqh | 434 | 314 | −120 | −28% |
| cmovlbs | 2 | 2 | 0 | 0% | cmpule | 450 | 440 | −10 | −2% |
| addt | 3 | 3 | 0 | 0% | cmpult | 550 | 572 | +22 | +4% |
| cmovlbc | 4 | 5 | +1 | +25% | cmplt | 561 | 585 | +24 | +4% |
| cmovle | 5 | 5 | 0 | 0% | rdteb | 587 | 590 | +3 | +1% |
| insqh | 6 | 6 | 0 | 0% | extwl | 654 | 340 | −314 | −48% |
| cmovgt | 13 | 13 | 0 | 0% | stq_u | 689 | 687 | −2 | 0% |
| callsys | 15 | 16 | +1 | +7% | blt | 751 | 770 | +19 | +3% |
| mulq | 13 | 13 | 0 | 0% | bic | 759 | 346 | −413 | −54% |
| s8subq | 13 | 14 | +1 | +8% | extll | 784 | 805 | +21 | +3% |
| cmovlt | 16 | 16 | 0 | 0% | extlh | 784 | 805 | +21 | +3% |
| ldt | 25 | 25 | 0 | 0% | bge | 813 | 831 | +18 | +2% |
| zap | 26 | 27 | +1 | +4% | mb | 883 | 901 | +18 | +2% |
| umulh | 32 | 32 | 0 | 0% | sll | 899 | 569 | −330 | −37% |
| mull | 46 | 48 | +2 | +4% | subl | 983 | 995 | +12 | +1% |
| ornot | 46 | 46 | 0 | 0% | br | 1130 | 1100 | −30 | −3% |
| cmpeq | 53 | 53 | 0 | 0% | sra | 1134 | 528 | −606 | −53% |
| insql | 61 | 61 | 0 | 0% | bsr | 1158 | 1165 | +7 | +1% |
| blbs | 63 | 63 | 0 | 0% | s4addl | 1160 | 1170 | +10 | +1% |
| s8addl | 69 | 70 | +1 | +1% | ret | 1232 | 1239 | +7 | +1% |
| mskwl | 73 | 0 | −73 | −100% | zapnot | 1247 | 911 | −336 | −27% |
| jsr | 90 | 92 | +2 | +2% | addq | 1589 | 1631 | +42 | +3% |
| cpys | 87 | 41 | −46 | −53% | subq | 1994 | 2046 | +52 | +3% |
| mskqh | 152 | 157 | +5 | +3% | ldah | 2684 | 2691 | +7 | +0% |
| cmovne | 160 | 165 | +5 | +3% | extbl | 2921 | 1682 | −1239 | −42% |
| mskbl | 163 | 0 | −163 | −100% | xor | 3278 | 3332 | +54 | +2% |
| cmoveq | 182 | 190 | +8 | +4% | and | 3361 | 2990 | −371 | −11% |
| insbl | 182 | 19 | −163 | −90% | bne | 4328 | 4376 | +48 | +1% |
| extwh | 195 | 196 | +1 | +1% | addl | 4734 | 4856 | +122 | +3% |
| trapb | 210 | 211 | +1 | 0% | ldq_u | 5061 | 4046 | −1015 | −20% |
| mskql | 201 | 203 | +2 | +1% | stl | 5418 | 5052 | −366 | −7% |
| jmp | 209 | 215 | +6 | +3% | lda | 6289 | 5344 | −945 | −15% |
| cmovge | 226 | 236 | +10 | +4% | stq | 6464 | 6588 | +124 | +2% |
| blbc | 238 | 238 | 0 | 0% | ldq | 6685 | 6359 | −326 | −5% |
| bgt | 292 | 302 | +10 | +3% | beq | 7355 | 7466 | +111 | +2% |
| ldl_l | 314 | 320 | +6 | +2% | bis | 10890 | 10668 | −222 | −2% |
| stl_c | 314 | 320 | +6 | +2% | ldl | 14964 | 13772 | −1192 | −8% |
| extql | 326 | 329 | +3 | +1% | **Totals** | **111288** | **106521** | **−4767** | **−4%** |

credit, which is similar to the TPC-A benchmark. The TPC-B benchmark is similar to the TPC-A, differing only in its method of execution. Cvetanovic and Bhandarkar presented an instruction distribution matrix for the debit-credit workload. The Alpha instruction type mix is dominated by the integer class, followed by other, load, branch, and store instructions, in descending order.[17] We took a similar approach but divided the instructions into more groups to achieve a finer detailed distribution. Table 13 gives the instruction makeup of each group. Figure 4 shows the percentage of instructions in each group for the four alternatives we studied. In all four cases, INTEGER LOADs make up 32 percent of the instructions executed. In the scaled Byte/Word category, the new ldbu and ldwu instructions compose 1 percent of the integer instructions, and the new stb and stw instructions accounted for 18 percent of the integer store instructions executed.

**Table 13**
Instruction Groupings

| Instruction Group | Group Members |
|---|---|
| Integer loads | ldwu, ldbu, ldl_l, ldah, ldq_u, lda, ldq, ldl |
| Integer stores | stb, stw, stl_c, stq_u, stl, stq |
| Integer control | blbs, jsr, jmp, blbc, bgt, blt, bge, br, bsr, ret, bne, beg |
| Integer arithmetic | cmpbge, s8subq, umulh, mull, cmpeq, s8addl, cmple, cmpule, cmpult, cmplt, subl, s4addl, addq, subq, addl |
| Logical shift | cmovlbs, cmovlbc, cmovle, cmovgt, cmovlt, ornot, cmovne, cmoveq, cmovge, srl, bic, sll, sra, xor, and, bis |
| Byte manipulation | insll, inslh, mskll, mskhl, insqh, zap, insql, mskwl, mskqh, mskbl, insbl, extwh, insbl, extwh, mskql, extql, inswl, extqh, extwl, extll, extlh, zapnot, extbl |
| Other | addt, ldt, stt, mulq, callsys, cpys, trapb, rdteb, mb |

During the scaled transactions, each instruction group showed a decrease in the number of instructions executed, ranging from negligible to as much as 54 percent. In addition, the number of byte manipulation and logical shift instructions decreased, because the method of loading or storing bytes and words on the original Alpha Architecture made heavy use of these types of instructions.

In our last examination, we looked at the instruction variation between a scaled and a cached transaction. The major difference between the two transactions is the additional I/O required by the scaled version of the benchmark. Table 14 gives the results. The Original version of the SQL Server database executed an extra 4,596 instructions during the cached transaction as compared to the scaled transaction. For the Byte/Word version, only an additional 1,334 instructions were executed.

## Conclusions

The introduction of the new single byte and word manipulation instructions in the Alpha Architecture improved the performance of the Microsoft SQL Server database. We observed a decrease in the number of instructions executed per transaction, the elimination of some instructions in the workload, a redistribution of the instruction mix, and an increase in relative performance. The results are in line with expectations when the addition of the new instructions was proposed.

We limited our investigation to a single commercial workload and operating system. Testing a workload with more I/O, such as the TPC-C benchmark, would

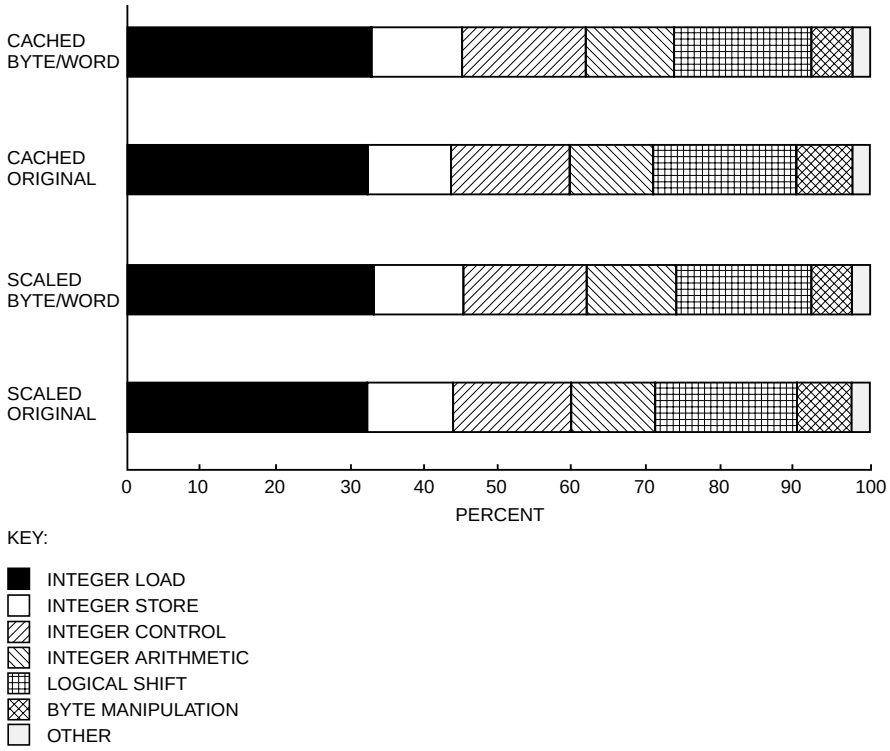


**Figure 4**
Instruction Group Distribution

**Table 14**
Instruction Variations (Scaled Minus Cached Transactions)

| Instruction | Original | Byte/Word | Instruction | Original | Byte/Word | Instruction | Original | Byte/Word |
|---|---|---|---|---|---|---|---|---|
| stw | 0 | −2 | cmplt | −4 | +51 | subl | −69 | −66 |
| ldwu | 0 | −26 | rdteb | −17 | −7 | br | −30 | +20 |
| ldbu | 0 | +117 | extwl | −6 | −5 | sra | −77 | −34 |
| cmovlbc | −1 | +1 | stq_u | +1 | −1 | bsr | −45 | −26 |
| callsys | −3 | +2 | blt | −33 | −1 | s4addl | −16 | +4 |
| s8subq | −4 | −3 | bic | −12 | −1 | ret | −50 | −25 |
| zap | −8 | −6 | extll | −5 | +44 | zapnot | −15 | +1 |
| umulh | −3 | −3 | extlh | −5 | +44 | addq | −115 | −54 |
| mull | −14 | −14 | bge | −15 | +12 | subq | −165 | −94 |
| ornot | −6 | −6 | mb | −78 | −40 | ldah | −109 | −55 |
| cmpeq | −11 | −8 | sll | −50 | −21 | extbl | +19 | +14 |
| blbs | −6 | −6 | cmovge | −65 | −51 | xor | −148 | −48 |
| s8addl | −2 | −4 | blbc | −11 | −11 | and | −41 | +21 |
| mskwl | −1 | 0 | bgt | −39 | −26 | bne | −209 | −64 |
| jsr | −8 | +3 | ldl_l | −30 | −15 | addl | −163 | +1 |
| cpys | −17 | 0 | stl_c | −30 | −15 | ldq_u | +15 | +113 |
| mskqh | −3 | +4 | extql | −3 | +2 | stl | −335 | −249 |
| cmovne | +13 | +24 | cmple | −1 | +16 | lda | −207 | −91 |
| cmoveq | −1 | +17 | inswl | −9 | −4 | stq | −314 | −125 |
| extwh | −1 | 0 | srl | −24 | −15 | ldq | −333 | −160 |
| trapb | +7 | −4 | extqh | −7 | −3 | beq | −252 | +11 |
| mskql | −3 | +1 | cmpule | −18 | −10 | bis | −394 | −39 |
| jmp | +1 | +15 | cmpult | −13 | +54 | ldl | −998 | −488 |
| | | | | | | **Totals** | **−4596** | **−1334** |

produce a different set of results and would merit investigation. The use of another database, such as the Oracle RDBMS, which makes greater use of byte operations, would possibly result in an even greater performance impact. Lastly, rebuilding the entire operating system to use the new instructions would make an interesting and worthwhile study.

## Acknowledgments

As with any project, many people were instrumental in this effort. Wim Colgate, Miche Baker-Harvey, and Steve Jenness gave us numerous insights into the Windows NT operating system. Tom Van Baak provided several analysis and tracing/simulation tools for the Windows NT environment. Rich Grove provided access to early builds of the GEM compiler back end that contained byte and word support. Stan Gazaway built the SQL Server application with the modifications. Vehbi Tasar provided encouragement and sanity checking. John Shakshober lent insight into the world of TPC. Peter Bannon provided the early prototype machine. Contributors from Microsoft Corporation included Todd Ragland, who helped rebuild the SQL Server; Rick Vicik, who provided detailed insights into the operation of the SQL Server; and Damien Lindauer, who helped set up and run the TPC benchmark. Finally, we thank Dick Sites for encouraging us to undertake this effort.

## References and Notes

1. Z. Cvetanovic and D. Bhandarkar, "Characterization of Alpha AXP Performance Using TP and SPEC Workloads," *21st Annual International Symposium on Computer Architecture,* Chicago (1994).

2. W. Kohler et al., "Performance Evaluation of Transaction Processing," *Digital Technical Journal,* vol. 3, no. 1 (Winter 1991): 45–57.

3. S. Leutenegger and D. Dias, "A Modeling Study of the TPC-C Benchmark," *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data,* SIGMOD Record 22 (2), (June 1993).

4. R. Sites and E. Perl, *PatchWrks—A Dynamic Execution Tracing Tool* (Palo Alto, Calif.: Digital Equipment Corporation, Systems Research Center, 1995).

5. W. Kohler, A. Shah, and F. Raab, *Overview of TPC Benchmark C: The Order-Entry Benchmark* (San Jose, Calif.: Transaction Processing Performance Council Technical Report, 1991).

6. R. Sites, "Alpha AXP Architecture," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue 1992): 19–34.

7. *Alpha AXP Systems Handbook* (Maynard, Mass.: Digital Equipment Corporation, 1993).

8. *DECchip 21064A-233, -275 Alpha AXP Microprocessor Data Sheet* (Maynard, Mass.: Digital Equipment Corporation, 1994).

9. *Alpha 21164 Microprocessor Hardware Reference Manual* (Maynard, Mass.: Digital Equipment Corporation, 1994).

10. R. Sites and R. Witek, *Alpha AXP Architecture Reference Manual,* 2d ed. (Newton, Mass.: Digital Press, 1995).

11. G. Kane, *MIPS R2000 RISC Architecture* (Englewood Cliffs, N.J.: Prentice Hall, 1987).

12. J. Hennessy, N. Jouppi, F. Baskett, and J. Gill, *MIPS: A VLSI Processor Architecture* (Stanford, Calif.: Computer Systems Laboratory, Stanford University, Technical Report No. 223, 1981).

13. J. Hennessy, N. Jouppi, F. Baskett, T. Gross, J. Gill, and S. Przybylski, *Hardware/Software Tradeoffs for Increased Performance* (Stanford, Calif.: Computer Systems Laboratory, Stanford University, Technical Report No. 228, 1983).

14. The original MIPS Architecture at Stanford University did not contain single byte manipulation instructions; this decision was reversed for the first commercially produced MIPS R2000 processor. The Intel x86 Architecture has always included these instructions.

15. C. Cole and L. Crudele, personal correspondence, December 1996.

16. Microsoft Corporation developed the ARC firmware for the MIPS platform. During the early days of the port of Windows NT to Alpha, DIGITAL's engineers ported the ARC firmware to the Alpha platform.

17. The Alpha instruction type mix included PALcode calls, barriers, and other implementation-specific PALcode instructions.

## Biographies

**David P. Hunter**
David Hunter is the engineering manager of the DIGITAL Software Partners Engineering Advanced Development Group, where he has been involved in performance investigations of databases and their interactions with UNIX and Windows NT. Prior to this work, he held positions in the Alpha Migration Organization, the ISV Porting Group, and the Government Group's Technical Program Management Office. He joined DIGITAL in the Laboratory Data Products Group in 1983, where he developed the VAXlab User Management System. He was the project leader of the advanced development project, ITS, an executive information system, for which he designed hardware and software components. David has two patent applications pending in the area of software engineering. He holds a degree in electrical and computer engineering from Northeastern University.

**Eric B. Betts**
Eric Betts is a principal software engineer in the DIGITAL Software Partners Engineering Group, where he has been involved with performance engineering, project management, and benchmarking for the Microsoft SQL Server and Windows NT products. Previously with the Federal Government Region, Eric was a member of the technical support group and a technical lead on several government programs. Before joining DIGITAL in 1990, he worked in many different software development areas at Martin Marietta and the Defense Information Systems Agency. Eric received a B.S. in computer science from North Carolina Central University.