Optimized Sparse Solver in MSC/NASTRAN for Compaq Alpha-Based Architecture

Pari Rajaram
Susanne M. Balle
Chuck Newman
Compaq Computer Corporation

ABSTRACT

Numerical simulations of stress, vibration, and heat transfer for the analysis and optimization of structures and mechanical components are now mainstream engineering tools in the Aerospace and Automotive industries. As models become larger and more sophisticated, prediction accuracy increases, but computers need to work faster to keep turnaround time and throughput acceptable. In this paper we present substantially enhanced MSC/NASTRAN performance results using optimized sparse solvers. The combination of these new solvers and Compaq's latest 64-bit Alpha systems brings true supercomputer power to the user, and offers the capacity to process a large number of analyses and the capability to solve large simulations fast and affordably, with direct benefit in time-to-market and product quality. The technologies used for CPU, system, and solver performance optimization, and the improvements achieved, are discussed and quantitative comparisons are made.

1. Introduction

Application such as MSC/NASTRAN which perform numerical simulations of stress, vibration, and heat transfer for analysis and optimization of structures and mechanical components are now the mainstream engineering tools in the Aerospace and Automotive industries. These simulations are generated by floating point intensive applications, and customers are building more complex and finer models with more degrees of freedom. These demands have forced the high performance computing systems vendors to constantly improve their products to satisfy this need.

There are many components of computer systems that affect the performance of high performance (HPC) applications. First, there is the system hardware which includes the microprocessor, memory subsystem, system bus and the I/O subsytem. Second are the compilers for generating a very efficient instruction stream on the architecture. Finally, there is a well optimized algorithm implementation which plays a very important role in performance enhancement on that architecture.

In this paper, we will describe the advanced and new features of the new 21264 microprocessor architecture. The CPU 21264 is a super–scaler microprocessor with out-of order and speculative execution. Out-of–order execution implies that instructions can execute in an order that is different from the order that the instructions are fetched. Instructions which cannot execute are deferred while subsequent instructions proceed. Speculative execution means the CPU executes instructions which may not be on the final execution path. This is particularly useful, for instance, when the 21264 predicts branch directions and speculatively executes down the predicted path. These two features allow the 21264 CPU to keep busy, resulting in better MSC/NASTRAN performance than possible with earlier Alpha chips at the same clock rate.

The 21264 also offers very high bandwidth for data access. For example, the 21264 based Compaq DS20 system includes a high-bandwidth system interface that channels up to 4+ Gbytes per second of board cache data and 2.6 Gbytes per second of main memory data into the processor, feeding its demanding CPU core.

In the next section of this paper we will talk about the loop optimization techniques used on some of the MSC/NASTRAN routines to achieve exceptional performance on Compaq's Alpha computers. Loop optimization techniques were employed to increase data reuse in cache by reusing the data which had been fetched from memory, and to increase data reuse in registers by reusing the data which had been fetched from cache. This improves cache hit ratio and delivers good performance.

Finally, we will compare the performance of v70.5 MSC/NASTRAN on a 21164 based system against a similar 21264 system. We will also compare the performance of MSC/NASTRAN v70.5 against the optimized version of MSC/NASTRAN on Alpha platform with standard MacNeal-Schwendler Corp. MSC/NASTRAN benchmarks.

2. 21264 Alpha Microprocessor

The 21264 is an advanced RISC microprocessor enabling both higher clock rate and lower cycles per instruction (*cpi*) through aggressive out-of-order execution, improved branch prediction schemes and higher memory bandwidth. The core of the 21264 is a highly out-of-order processor with a peak execution rate of six instructions per cycle and a sustainable rate of four per cycle [1]. Up to 80 instructions can be in process at once, more than in any other microprocessor. Registers are renamed on the fly, with 41 extra integer registers (80 total) and 41 extra floating-point registers available [1]. The two main features in 21264 which helps MSC/NASTRAN run faster are discussed in detail in section 2.1 and 2.2.

2.1 Out-of-Order and Speculative Execution

The out-of-order issue logic receives four fetched instructions every cycle, rename/remaps the registers to avoid unnecessary register dependencies, and queues the instructions until operands and/or functional units become available. It dynamically issues up to a maximum of six instruction every cycle – four integer and two FP instructions. These instructions get issued as soon as the functional units and data become available. There are 32 integer and 32 floating point registers available to the software (compiler and assembler) and 41 integer and 41 floating point registers available to hold speculative results prior to instruction commitment/retirement in a large 80 instruction in-flight window. The out-of-order unit coupled with register renaming and speculative execution (based on advanced branch prediction) schedules independent instructions from the large 80 instruction in-flight window to the integer and floating point execution unit. This implies that up to 80 instructions can be in partial state of completion at any time, keeping the CPU busy. Instructions are scheduled as soon as possible, maximizing instruction parallelism and minimizing the latency. [1]

This feature in Alpha 21264 microprocessor is illustrated with a simple example.

```
    load r9 <- temp
L1: load r1 <- a(i)
    load r2 <- b(i)
    add  r3 <- r1+r2
    mult r4 <- r9*r3
    store r4 -> c(i)
    branch to L1 on true
L2: load r1 <- d(i)
    load r2 <- b(i)
    add  r3 <- r1+r2
    mult r4 <- r9*r3
    store r4 -> e(i)
    branch to L2 on true
L3: jsr end_of_subroutine_
```

Let us assume that one iteration of the loop L1 and L2 take 10 cycles each. Let the loop L1 be executed four times by the algorithm and jumps to loop L2 which gets executed only once. Assume that compiler has used software pipelining and schedules the entire code in 35 cycles (25 cycles for L1 and 10 for L2). Also assume that *a(1)* and *b(3)* gets a read cache miss.

Now in the inorder execution, the first iteration of add will stall the entire execution until *a(1)* is fetched into the register r1. Once the add operation completes the subsequent mult operation will proceed as it is dependent on add instruction. Eventually the third iteration of add will again stall the entire execution due to the cache miss of *b(3).* Let each cache miss cost 8 cycles, then the total time for the execution of L1+L2 is of 25 + 8+8 +10 cycles.

In the out-of-order execution, when there is a cache miss of *a(1),* the second iteration of L1 will execute before the first iteration of L1 completes and similarly with the third and fourth iteration of L1 and hence we may hide the cache miss latency of *a(1)* and *b(3)* and may execute well below 25+10 cycles.

The combination of out-of-order, speculative execution based on advance branch prediction and register renaming in the 21264, the loop L2 gets executed in the unused cycles of L1 loop and the results of it are held in the internal registers of the chip. When the branch condition become true, the instruction of L2 are no longer executed, but the internal registers are remapped to the software registers, thus executing the entire code in much lesser than 35 cycles.

2.3 Memory System

The memory system of the 21264 offers a very high bandwidth. It is comprised of one or two caches as well as the main memory. The first level cache is a large on-chip 64 KB two-way set-associative instruction cache and a 64 KB two-way set-associative data cache. This L1 cache receives up to two memory operations (loads or stores) from the integer execution unit every cycle and hence delivers up to 16 bytes per cycle. Load to use latency L1 data cache is 3 cycles for integer data and 4 cycles for floating point data. Since the L1 caches are pipelined, they offer 8+ GB/sec sustained bandwidth with 16 bytes/cycle for a 500 MHz processor.

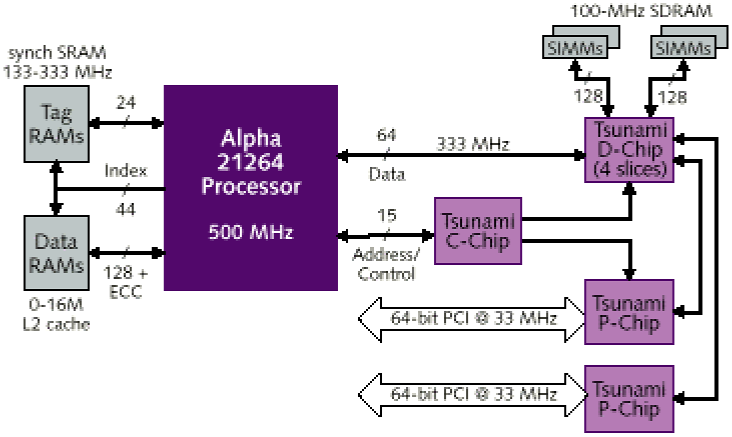


Fig 2. Memory subsystem – External cache and memory interface to 21264 [2]

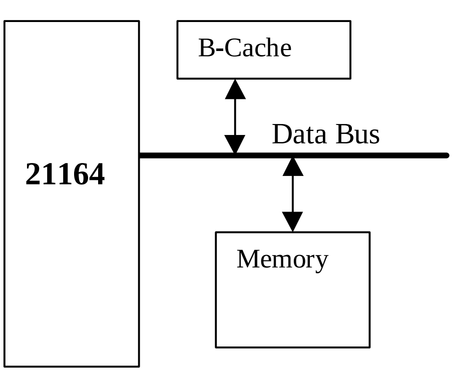


Fig. 3. Simplified diagram of 21164's external-
Cache and memory interface

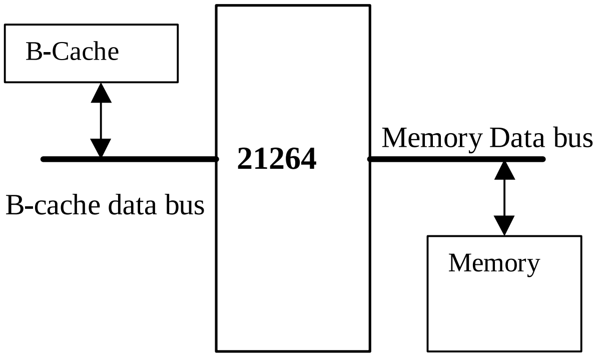
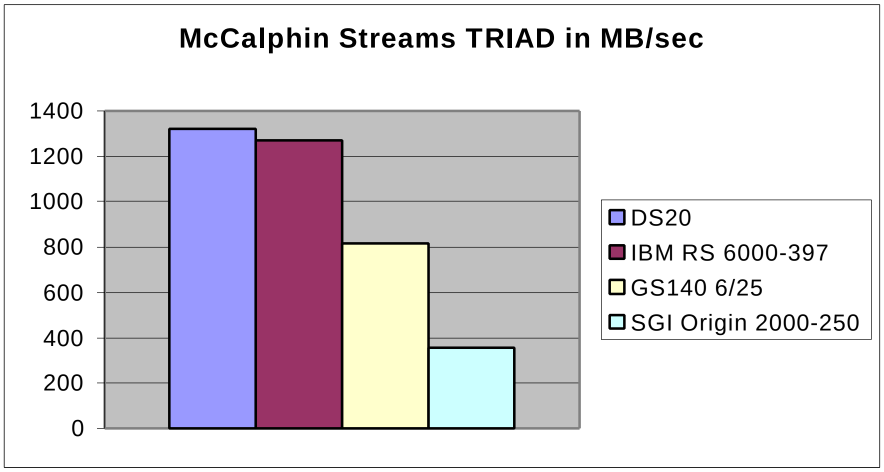


Fig 4. Simplified diagram of Fig 2.

The L1 cache is optionally backed up by a large off-chip cache of up to 16 MB, known as the L2 cache. This cache is direct mapped and shared by both instructions and data. Unlike its predecessor (21164), which has a single 128-bit bus to the external cache and main memory as shown in Fig 3., the new chip 21264 includes a dedicated 128-bit L2 cache bus and a separate 64-bit system bus, as Figure 2 shows, which operates at speeds up to 333 MHz. The load to use latency of L2 cache is 12 cycles for a typical system and can provide a sustained bandwidth of 4+ GB/sec. The 21264 memory system supports up to 8 64-bit read misses and 7 64-bit write back off-chip memory references. This implies a high degree of parallelism between the memory system and CPU. The CPU continues to execute other independent instructions even though there are many (up to 16) cache misses. This translates into higher performance.

The 21264 provides cache prefetch instructions that allow the compiler to take advantage of the high-bandwidth features of memory system. These prefetching techniques are very useful for applications such as MSC/NASTRAN that have loops that reference very large arrays. Compaq's GEM optimizing compilers can predict these memory references in loops very intelligently and prefetch the data by issuing load instructions far in advance of when the data is needed.

The result of this high bandwidth memory feature can be seen by comparison of the McCalpin stream benchmark results of various vendors.



3. Optimization of Sparse Algorithms

Finite Element Analysis (FEA) applications such as MSC/NASTRAN are computational intensive problems with calculations involving floating point data. The primary means to get more performance from a RISC system is to keep the CPU busy with useful floating point operations like multiplication and addition. But most of the time the numerical algorithms are memory intensive in nature and operate on large arrays that have a high ratio of memory references to floating point operations. Since the processor speed is much larger than the memory sub-system speed, the instructions will often stall waiting for their operands to be loaded into registers from the memory sub-system. Hence many processor cycles are wasted waiting for the data. Therefore it is very important to reduce the number of memory accesses by reusing the data once they are brought into cache and registers. There are three optimization techniques we have implemented in many MSC/NASTRAN kernels with the goal of reusing the data once they are brought into registers and cache.

3.1 Unrolling

Loop unrolling is a very common technique to increase the register reuse by unrolling the outer loops of a nested loop. This technique is only useful if the algorithm references the same data multiple times in the execution of the loops. This technique is best illustrated with matrix multiplication.

Original Matrix Multiplication:

```
   C  MATRIX-MATRIX MULTIPLICATION
      DO I=1,N
       DO J=1,N
        TEMP=C(J,I)
        DO K=1,N
         TEMP= TEMP+A(J,K)*B(K,I)
        END DO
        C(J,I)=TEMP
       END DO
      END DO
```

Loop Unrolling employed to the above matrix multiplication algorithm:

```
C UNROLLING I and J loops by 2
 DO I=1,N,2
  DO J=1, N,2
    TEMP1 = C(J,I)
    TEMP2 = C(J+1,I)
    TEMP3 = C(J,I+1)
    TEMP4 = C(J+1,I+1)
    DO K=1,N
     TEMP1 = TEMP1  + A(J,K)  * B(K,I)
     TEMP2 = TEMP2  + A(J+1,K)* B(K,I)
     TEMP3 = TEMP3  + A(J,K)  * B(K,I+1)
     TEMP4 = TEMP4  + A(J+1,K)* B(K,I+1)
     END DO
     C(J,I)     = TEMP1
     C(J+1,I)   = TEMP2
     C(J,I+1)   = TEMP3
     C(J+1,I+1) = TEMP4
   END DO
  END DO
```

In the first case, the total number of memory references are $2N^3 + 2N^2$. N^3 memory references are due to loading elements of the A array and N^3 are due to loads of the B array. $2N^2$ memory references are from loads and stores of the C array.

After loop unrolling, the total number of memory references are $N^3 + 2N^2$. This is because every element of *A* and *B* array are fetched once but used twice in the unrolled loop and hence their memory reference are reduced by a factor of 2. In the inner loop of the unrolled case there are 4 memory references and 8 floating point operations, whereas in the first case there are 2 memory references and 2 floating point operations. The ratio of floating point operations (useful work) to memory references is 2.0 in the unrolled case. In the original case this ratio is 1.0. Consequently, the CPU does more useful work and spends less time waiting for data. The total number of memory references can be further reduced by more aggressive unrolling, but if pushed too far will require more registers than are available. The result will be degradation of performance. The optimal unrolling on Alpha based systems is to unroll each of the two outer most loops by 4. For reference, see [3].

3.2 Blocking

While loop unrolling increases data-reuse in registers, blocking helps increase data-reuse in cache. Instead of operating on entire rows or columns of an array (which may be too large to fit in cache), blocked algorithms operate on submatrices, or blocks, so that data loaded into the faster levels of memory hierarchy (cache) are reused. This technique reduces cache misses and therefore reduces the number of times the CPU has to wait for data from memory.

```
C  BLOCKING THE LOOPS I, J and K by BLK
   DO II=1,N,BLK
     DO JJ=1,N,BLK
      DO KK=1,N,BLK
       DO I=II,min(II+BLK-1,N)
        DO J=JJ, min(JJ+BLK-1,N)
         TEMP=C(J,I)
         DO K=KK, min(KK+BLK-1,N)
          TEMP= TEMP+A(J,K)*B(K,I)
         END DO
         C(J,I)=TEMP
        END DO
       END DO
      END DO
     END DO
    END DO
```

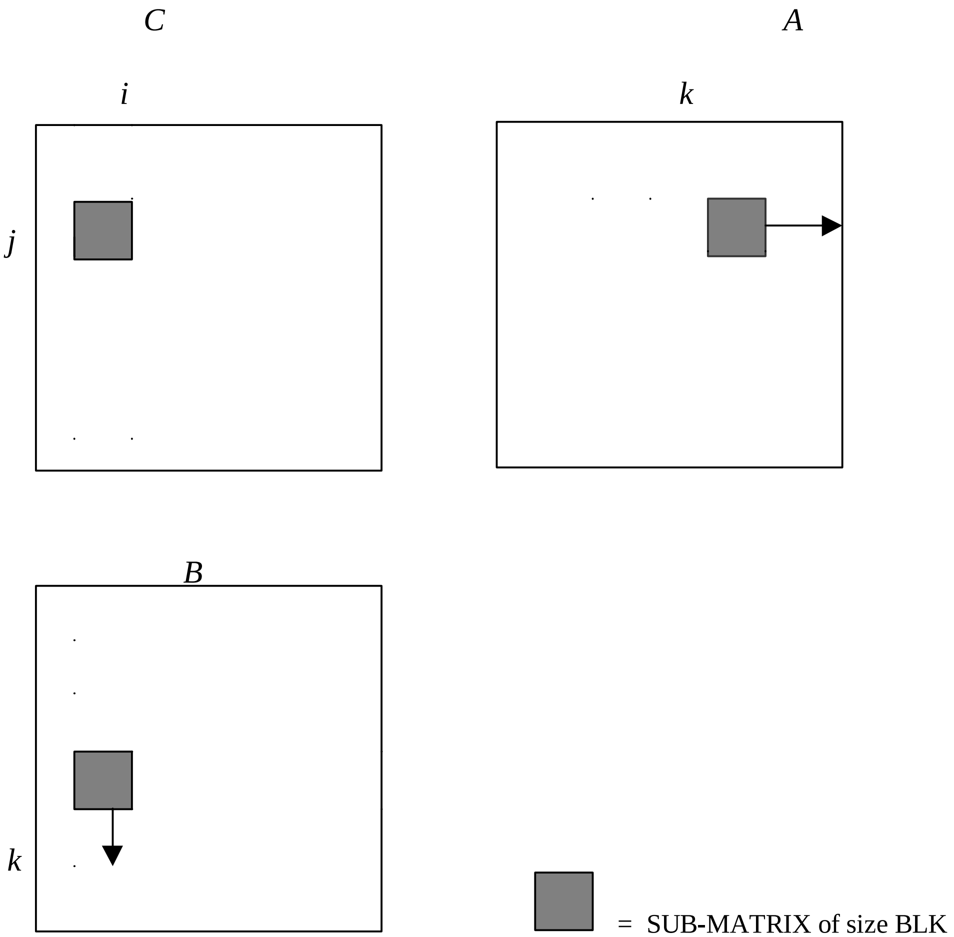


Fig:4. Pattern of data access using blocking

The total number of data references in this case is slightly more than before, but most of these references can be resolved from cache rather than memory.

Fig. 4 illustrates the data access of the blocked algorithm. From figure 4 we see that the block of *C* gets reused as consecutive blocks from *A* and *B* are processed. In the inner most loop of the blocked algorithm, every element of *A* and *B* array in the block gets reused *BLK-1* times. Only the first load of *C* and the last store of *C* need to go to the memory; all other references are resolved in cache.

One would expect that performance will increase with increases in block size of the sub-matrix being reused, but it is only true up to a point. The problem with blocking has to do with how data are placed in cache. Data are loaded into and stored from cache in groups called cache lines. On Alpha systems each cache line holds 4 or 8 floating point data, depending on the cache and the system.

The memory address of each datum determines where the datum will live in cache. When a cache line containing the required datum is loaded into cache at its destined location, it will displace the cache line currently at that location. If two data are both needed in sequential iterations and both need to reside at the same cache line then these data will repeatedly displace each other. This phenomenon is known as cache thrashing.

The concept of "set associativity" helps by providing multiple possible locations in cache for each datum, but it does not completely eliminate the problem unless the cache is fully associative, meaning that any cache line can be placed at any location in the cache. For reference, see [3].

3.3 Copy Optimization

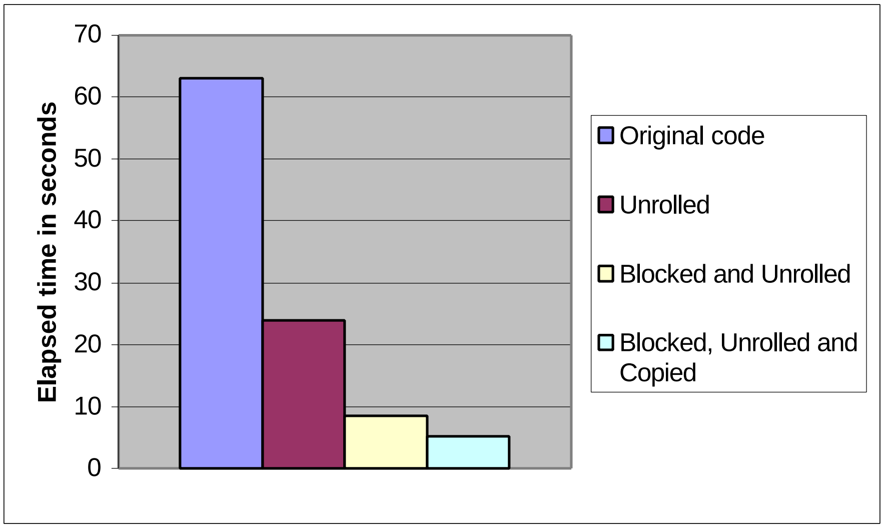
As the blocking factor increases, the amount of cache thrashing increases exponentially. An analogy would be leaves falling in your yard in autumn. Imagine a one acre yard with a tree with one acre worth of leaves about to fall. The yard represents the cache and each leaf represents a cache line that needs to exist in cache. While it is unlikely that the first two leaves to fall will land on top of each other, it is certain that before the last leaf falls there will be a lot of overlapping leaves.

Fortunately, there are techniques that can reduce cache thrashing, but they are not without cost.

One technique is to decrease the size of the block (or blocking factor) such that only a fraction of the cache is used, reducing the likelihood that cache lines will overlap. Since the value of blocking goes up approximately linearly with the size of the blocks, this technique will significantly limit the efficacy of blocking

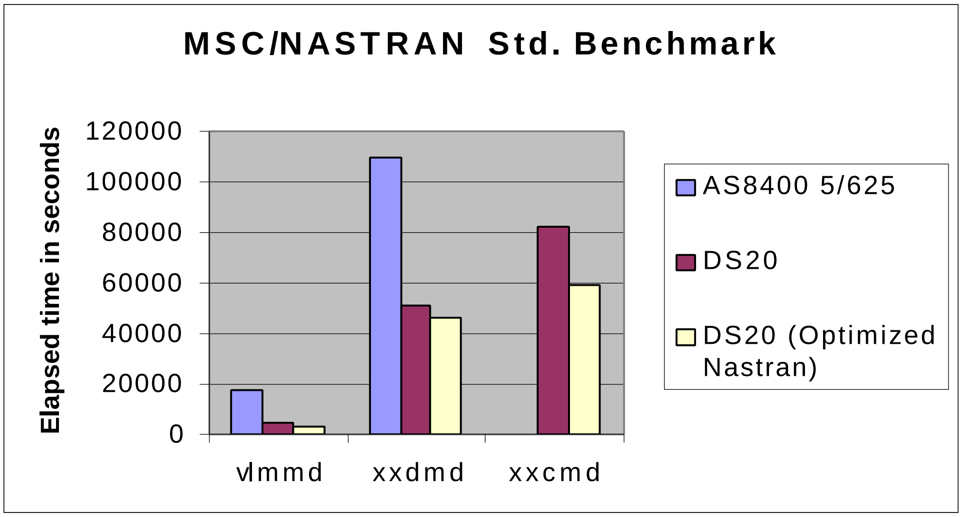
Another technique is to copy the blocks into a section of memory whose size is equal to the size of the cache, perform the calculations on the data, then copy the modified data back to the original matrices. The data will be laid out in the temporary memory such that no thrashing will occur, allowing maximum block size. For reference, see [3].

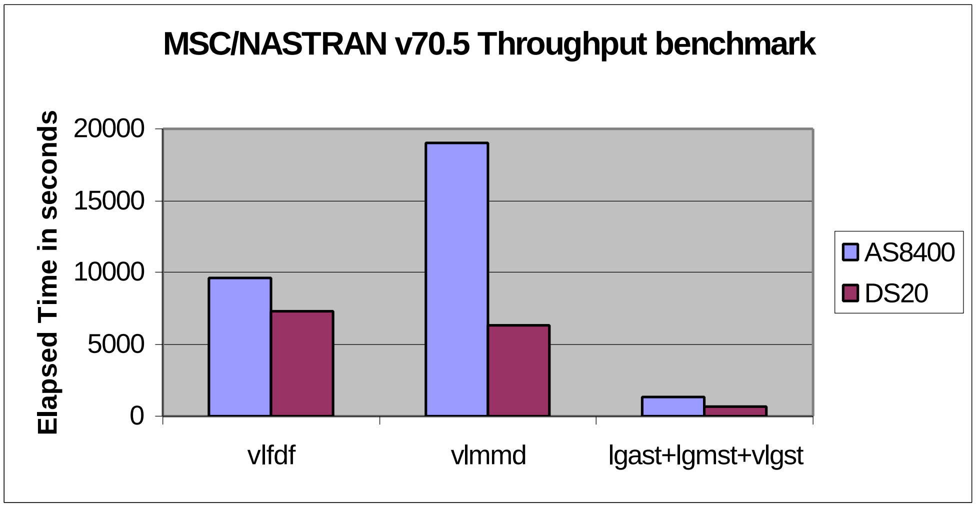
The performance improvement for blocking, unrolling and copying of a small test matrix is shown below.



4. MSC/NASTRAN Standard Benchmark Results on Alpha systems

The results of three MSC standard benchmarks with MSC/NASTRAN v70.5 and the latest optimized MSC/NASTRAN on two Alpha servers are compared below. AS8400 is a 21164 microprocessor based server. DS20 is the 21264 microprocessor based server. The results show us that 21264 microprocessor based system DS20 is more than twice faster than 21164 microprocessor based system AS8400. The optimization of many MSC/NASTRAN kernels with blocking, unrolling and copying helped us gain more performance over MSC/NASTRAN v70.5. The standard benchmarks vlmmd, xxdmd and xxcmd were run in single cpu. Vlfdf and vlmmd were run simultaneously while lgast, lgmst and vlgst were run sequentially for the throughput benchmark.





5. Conclusion

The 21264 microprocessor offers twice the MSC/NASTRAN performance over 21164 microprocessor based system. It has a high clock rate and advanced features like out-of-order and speculative execution. It includes a high bandwidth memory system that offer 2.6 GB/sec peak bandwidth. Memory references are reduced and cache hit ratio is increased by optimizing the loops of MSC/NASTRAN sparse kernels. The combination of these new optimized sparse solvers and Compaq's 21264 Alpha microprocessor based systems brings true supercomputer power to the MSC/NASTRAN user at an affordable cost.

Bibliography

1. R.E. Kessler, E.J. McLellan and D.A. Webb. "The Alpha 21264 Microprocessor Architecture".ICCD'98, October, 1998

2. Linley Gwennap. "Digital 21264 Sets New Standard". Microprocessor Report, Vol 10, Issue 14., 1996

3. Kevin Dowd and Charles Severance. "High Performance Computing". O'Reilly publication.