TITLE: Beyond the CM-5: A Case Study in Performance Analysis for the CM-5, T3D, and High Performance RISC Workstations

AUTHOR(S): David M. Beazley
                  Peter S. Lomdah

SUBMITTED TO: Submitted to IEEE Conference "Supercomputing '95", December 1995, San Diego, CA

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

By acceptance of this article, the publisher recognized that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution or to allow others to do so for U.S. Government purposes.

The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Abstract

We present a comprehensive performance evaluation of our molecular dynamics code SPaSM on the CM-5 in order to devise optimization strategies for the CM-5, T3D, and RISC workstations. In this analysis, we focus on the effective use of the SPARC microprocessor by performing measurements of instruction set utilization, cache effects, memory access patterns, and pipeline stall cycles. We then show that we can account for more than 99% of observed execution time of our program. Optimization strategies are devised and we show that our highly optimized ANSI C program running only on the SPARC microprocessor of the CM-5 is only twice as slow as our Gordon-Bell prize winning code that utilized the CM-5 vector units. On the CM-5E, we show that this optimized code runs faster than the vector unit version. We then apply these techniques to the Cray T3D and measure resulting speedups. Finally, we show that simple optimization strategies are effective on a wide variety of high performance RISC workstations.

1 Introduction

It has often been remarked by history professors that “the only way to know where you’re going is to know where you’ve been.” In this paper, we will apply this simple principle to our molecular dynamics code developed for the CM-5 by performing a detailed study of processor utilization. Having spent over 2 years working on the CM-5, it is now time to start thinking about other architectures. In order to get high performance on new systems, we feel that we must step back and make a thorough assessment of how we used the CM-5 in the first place. By performing this study, we hope to discover performance problems so that we can avoid the other historical observation that “history tends to repeat itself.”

Our performance study will take an unconventional approach. Rather than focusing on Mflops, communications times, scaling, and other popular measurements, we will explore a very simple question: How well did our code utilize the processing power of the CM-5 processing nodes? To answer this question, we will attempt to quantify code behavior in a way similar to that found in Hennessy and Patterson [6]. We will focus on instruction set utilization, stall behavior, and cache effects in addition to scientific and algorithmic considerations. From this information we will suggest optimization strategies and measure resulting speedups. Finally, we will test these optimizations on the Cray T3D and high performance HP, SGI and IBM workstations. As RISC
2  Historical background

Over the past few years, we have developed a short-range molecular dynamics code SPaSM (Scalable Parallel Short-range Molecular dynamics) that we have been using to study properties of fracture mechanics in materials [1, 8]. The original code was written entirely in ANSI C with explicit message passing using the CMMD 1.0 message passing library on the CM-5. Early measurements indicated that we were getting 2 Mflops/node compared to the 128 Mflops peak performance of the vector units so much of our effort was spent programming the VUs [1]. Since the message-passing model maps very cleanly onto our application, we decided to access the VUs by rewriting a few critical routines in CDPEAC assembler language [7, 21]. As a result, our code was selected as one of the winners in the 1993 IEEE Gordon Bell Prize competition for sustaining 25-50 Gflops on the 1024 processor CM-5 at Los Alamos [7].

3  Vector Unit utilization

It is safe to say that getting high performance on the CM-5 is believed to revolve entirely around the vector units (VUs) [10, 11]. Our use of the vector units is isolated to a critical routine in our code that calculates the pair-interactions between two lists of particles. Written in CDPEAC assembler language, this routine processes 32 different pairs of particles simultaneously[7]. Unfortunately, due to the difficulty of vectorizing an inherently unstructured calculation, we have found that only 50% of the floating point operations performed in this routine are needed in our simulation. Thus, despite our ability to run the VUs quickly, we experience a significant performance penalty in the form of wasted floating point operations. Despite this inefficiency, we achieved performance ranging from 5-10 times faster than our original C code so our efforts seemed worth the cost.

4  Performance of the SPARC

In many respects, the performance of the VUs is an uninteresting problem since it only relates to the CM-5 and not to RISC architectures in general. A more interesting question is to find out how well our original C code used the SPARC processor. To the best of our knowledge, the performance of the SPARC has been completely ignored in CM-5 performance studies. As we will show, the performance of the SPARC may not be as bad as one might first imagine.

4.1  CM-5 processing nodes

Each processing node on the CM-5 features a 33 Mhz SPARC microprocessor manufactured by Cypress Semiconductor (CY7C601) [10]. This processor features a simple pipelined architecture, a 64 Kbyte direct-mapped combined data/instruction cache, and a floating point coprocessor capable of performing double precision operations at approximately 5.1 Mflops [9].
4.2 Analysis strategy

We start by taking our original C code developed before starting our work on the CM-5 vector units. Our goal will be to quantify all aspects of our code's execution on a single node. This includes dynamic measurements of instructions executed, memory access behavior, cache effects, and pipeline stalls. We will analyze the assembler source code generated by the C compiler for the most critical subroutines of our code. A small test problem will be used to make dynamic measurements. Cache effects and memory access patterns will be studied using a simulator that we have developed. All code has been compiled using the gcc 2.5.8 C compiler with an optimization level of -02. The CM-5 was running CMMD 3.3-Beta1 and CMOST 7.4.0 for all of the timings reported here.

A test problem involving 256000 atoms will be run on a 32 processor CM-5. The atoms are arranged in a uniform cubic lattice with density 1.0. Atoms interact according to a Lennard-Jones potential truncated at a cutoff of 2.5\sigma. This particular arrangement gives each node approximately 8000 atoms. While this simulation is relatively small compared to our normal production simulations, it makes performance analysis reasonable while still giving each node a sufficient amount of work.

4.3 Make the common case fast

By timing the original C code, we find that approximately 88% of the CPU time is spent calculating forces between atoms. The remaining 12% is spent message passing and performing other bookkeeping. Thus the greatest performance gains will be achieved by focusing entirely on the force calculation. In previous work, we have discussed an optimization strategy for communications so we refer the reader to [2].

4.4 Instruction set measurements

First, the force calculation was analyzed with a profiler to find out how often various code segments were executed. Then, using the assembler output of the compiler, dynamic instruction set measurements can be constructed. Table 1 shows the breakdown of different types of instructions executed by our code. Memory and control operations are further broken down into different types. We notice that we execute more memory operations than any other type of instruction. This is particularly disturbing considering the widening gap between processor and memory performance in modern systems. We also note that gcc has not inlined a critical subroutine as shown by the sizable number of CALL and RET operations.

4.5 Memory access and cache behavior

To study memory access patterns, we developed a simulator of the 64 Kbyte direct-mapped unified instruction/data virtual cache. The cache is organized into 2048 lines of 32 bytes each and is operated in write-through mode with no write allocate [9]. A 64 entry table lookaside buffer (TLB) is used for virtual address decoding, but we ignore this in our analysis. Our simulator provides three very simple operations: an instruction fetch, a data load, and a store operation. The simulator keeps track of the data stored in each cache line and records the number of hits and misses. While the simulator is simplistic, our goal is to produce a rough estimate of how well we use the cache in a controlled environment.
### Table 1. Dynamic instruction set usage

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction Count</th>
<th>% of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Operations</td>
<td>18279360</td>
<td>17.6%</td>
</tr>
<tr>
<td>Floating Point</td>
<td>27453440</td>
<td>26.4%</td>
</tr>
<tr>
<td>Memory Operations</td>
<td>47864320</td>
<td>46.1%</td>
</tr>
<tr>
<td>LD (load word)</td>
<td>1951232</td>
<td>1.9%</td>
</tr>
<tr>
<td>LDD (load double)</td>
<td>31361024</td>
<td>30.2%</td>
</tr>
<tr>
<td>STD (store double)</td>
<td>14552064</td>
<td>14.0%</td>
</tr>
<tr>
<td>Control</td>
<td>7025920</td>
<td>6.8%</td>
</tr>
<tr>
<td>BR (conditional branch)</td>
<td>3784960</td>
<td>3.6%</td>
</tr>
<tr>
<td>RET (return from subroutine)</td>
<td>1624064</td>
<td>1.6%</td>
</tr>
<tr>
<td>CALL (subroutine call)</td>
<td>1616896</td>
<td>1.6%</td>
</tr>
<tr>
<td>NOP (no operation)</td>
<td>3233792</td>
<td>3.1%</td>
</tr>
<tr>
<td>Total</td>
<td>103856832</td>
<td>100%</td>
</tr>
</tbody>
</table>

As input to the simulator, we use the assembler code generated by the compiler to determine the proper sequence of instruction fetch and data access operations. Special simulator functions are then added to the code in order to trace the control-flow of the original program. For the test problem, the simulated cache performance for a single time step is shown in Table 2. Remarkably, we find that we have an extremely high cache hit rate. To better understand why this is the case, all addresses at which a cache miss occurred were plotted in Figure 1. In the figure, we see a highly regular access pattern as our calculation steps through all of the data in memory.

<table>
<thead>
<tr>
<th>Type of access</th>
<th>Hits</th>
<th>%</th>
<th>Miss</th>
<th>%</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>103809282</td>
<td>99.999%</td>
<td>47550</td>
<td>0.001%</td>
<td>103856832</td>
</tr>
<tr>
<td>Load Data</td>
<td>33145187</td>
<td>99.5%</td>
<td>167069</td>
<td>0.5%</td>
<td>33312256</td>
</tr>
<tr>
<td>Store Data</td>
<td>14538469</td>
<td>99.9%</td>
<td>13595</td>
<td>0.1%</td>
<td>14552064</td>
</tr>
<tr>
<td>Total</td>
<td>151492938</td>
<td>99.8%</td>
<td>228214</td>
<td>0.2%</td>
<td>151721152</td>
</tr>
</tbody>
</table>

#### 5 Comparison with observed timings

With a clear picture of instruction set usage and cache behavior, we can now make a prediction of the execution time and compare it with the actually observed execution time. A timing measurement was made on the test code and we find that a total of 10.46 seconds are required with message passing and bookkeeping accounting for approximately 1.21 seconds. This leaves 9.25 seconds of CPU time spent in the force calculation.

Now, consider the execution profile in the previous section. From the SPARC Users Guide, we find that all integer operations require 1 clock cycle[9]. Due to the unified instruction/data cache, memory operations require multiple cycles with LD requiring 2, LDD requiring 3, and STD requiring 4 cycles [9]. Floating point operations are also multicycle, but addition/subtraction, multiplication/division, and integer operations can be overlapped as long as there are no structural
or data hazards. The floating point unit is not pipelined and delivers a peak performance of approximately 5.1 Mflops. Stall behavior was determined by writing another simulator that tracks the state of each functional unit (integer, floating point add, floating point multiply, etc...) and keeps track of any data-dependencies along with the timing requirements for various operations. From this we find that approximately 90.6 million stall cycles are introduced. Finally, from the SPARC users guide we find that all cache misses require a minimum of 3 clock cycles to be resolved [9]. Using our simulated cache results, approximately 684000 cache miss cycles will also occur. Putting all of this data together, we get the predicted execution time shown in Table 3. All times are calculated by dividing the number of cycles by 33 million (corresponding to 33 Mhz).

<table>
<thead>
<tr>
<th>Number of cycles</th>
<th>Time (sec)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Instructions</td>
<td>28539072</td>
<td>0.865</td>
</tr>
<tr>
<td>Memory Operations</td>
<td>156193792</td>
<td>4.733</td>
</tr>
<tr>
<td>Floating Point</td>
<td>27453440</td>
<td>0.832</td>
</tr>
<tr>
<td>Hazards (pipeline stalls)</td>
<td>90600000</td>
<td>2.745</td>
</tr>
<tr>
<td>Cache Miss</td>
<td>684642</td>
<td>0.021</td>
</tr>
<tr>
<td>Total</td>
<td>303483682</td>
<td>9.196</td>
</tr>
</tbody>
</table>

Table 3. Predicted execution time from known instruction analysis.

In the table, we have been able to account for virtually all of the CPU time observed in an actual simulation. We see that memory operations are taking more than half of the clock cycles and that a significant number of stall cycles occur due to data and structural hazards. It should also be noted that most floating point operations can be overlapped with other useful instructions so the 9.0% figure above only represents the time required to start a floating point operation. A large number of floating point operations are overlapped with integer and memory operations.
pt1 = pl->ptr;
for (i = 0; i < pl->n; i++, pt1++) {
    pt2 = p2->ptr;
    for (j = 0; j < p2->n; j++, pt2++) {
        force(&pt1->r, &pt2->r, &f);
        pt1->f.x += f.x;
        pt1->f.y += f.y;
        pt1->f.z += f.z;
        pt2->f.x -= f.x;
        pt2->f.y -= f.y;
        pt2->f.z -= f.z;
    }
}

register double rx,ry,rz,fx, fy, fz, f;
register int n1, n2;
n1 = pl->n;
n2 = p2->n;
pt1 = pl->ptr;
for (i = 0; i < n1; i++, pt1++) {
    rx = pt1->r.x;
    ry = pt1->r.y;
    rz = pt1->r.z;
    fx = fy = fz = 0;
    pt2 = p2->ptr;
    for (j = 0; j < n2; j++, pt2++) {
        if (force(rx, ry, rz, &pt2->r, &f) {
            fx += f.x;
            pt2->f.x -= f.x;
            fy += f.y;
            pt2->f.y -= f.y;
            fz += f.z;
            pt2->f.z -= f.z;
        }
    }
    pt1->f.x += fx;
    pt1->f.y += fy;
    pt1->f.z += fz;
}

Code 1. Original force calculation

Code 2. Modified force calculation

6 Optimization strategies

As applications programmers, there is a tendency to ignore the underlying hardware of a machine, but by looking at a SPARC architecture manual and a copy of our code, we have learned more about it’s real behavior than we would have been able to obtain using any performance analysis tool we are aware of. Since memory accesses seem to be a major performance problem, we begin by suggesting memory optimization strategies.

6.1 Good programming, bad performance

The first performance problem we discovered was the result of a programming oversight and is shown in Code 1. This code calculates all of the pair interactions between two lists of particles. A user definable function force() is used to compute the pair interaction. This design is modular and makes it easy to change the physics of a particular problem. However, close inspection of the assembler output reveals that the 6 statements after the force() call actually require 19 memory operations and 6 floating point operations. We also find that in approximately 80% of the iterations, the value of f is 0 so these operations are completely useless.

This problem can easily be solved by simply putting a return code on the force function that indicates whether f is 0 or not. By checking the return code, we can eliminate nearly half of the memory operations and a significant number of floating point operations. Timings indicated that a 58% speedup could be obtained using this one simple modification.
6.2 Inlining

Since gcc did not inline the force calculation, we can fix this problem by inlining the force calculation ourselves. This eliminates nearly half of the control and NOP operations in Table 3. The moral of the story—compilers don’t always do everything you might imagine.

6.3 Pointers and dynamic memory management

Since we make extensive use of dynamically allocated arrays and data structures our code relies heavily on pointer manipulation. For most C compilers, every value referenced by pointer will always result in a memory access (due to the potential of aliasing problems). Since this creates memory performance problems, we can avoid the problem by copying data to local variables and using them in critical routines. An example of this is shown in Code 2 although for clarify, the inlined force function is not shown.

This method works particularly well with nested loops as shown above, since values reused in the inner loop will now be placed in registers. By expliciting using local variables in this manner, C compilers are able to effectively use registers with dynamically allocated data structures. Instruction set measurements reveal that this modification results in an elimination of 5.5 million memory operations from the test problem.

7 Performance Improvements

Making the above changes to the code, we can now measure speedups on the test problem. A timing shows that the modifications reduce the execution time of the test problem from 10.4 seconds to 4.92 seconds, a speedup of 111%. Dynamic instruction set measurements now reveal the data in Table 4. We notice that we have achieved a huge reduction in memory operations from nearly 48 million before to just 8 million now. The number of control operations has been cut in half due to inlining however it’s interesting to notice that the number of conditional branch operations has remained essentially the same. Floating point operations now dominate as we would expect for a scientific code. In fact, using the floating point operation count above, we get a calculation rate of 4.0 Mflops which is roughly 78% of the peak floating point performance of the SPARC.
To get a better idea of the improvement we show timings for a slightly larger problem more representative of the simulations we usually perform. In addition we show the performance of the vector units and code performance on the CM-5E which uses a 40 Mhz SuperSPARC on each of the nodes. For the CM-5E, code was compiled using the Sun acc compiler version 2.01 with the options -dalign -fast -native -04. The results are shown in Table 5.

Here we get a slightly better speedup of 119%. Remarkably, our performance on the SPARC is only 2.2 times slower than that of the VUs. This is truly an amazing result considering that the peak performance of the VUs is 25 times faster. The performance on the CM-5E is even more interesting. First, we immediately see the advantage of running on a more advanced superscalar architecture as the unmodified C code is running 275% faster on the CM-5E even though the clock rate is only 21% faster than before. Much of this speedup is a direct result of separate instruction and data caches and a pipelined floating point unit capable of 40 Mflops peak performance. However, our optimizations provide an additional 88% speedup and the startling result that the optimized C code running on the SuperSPARC alone is faster than using the vector units on either machine. This certainly raises the issue of whether it was a good idea to use the VUs in the first place. It’s also clear that the inefficiencies introduced in using the VUs is a potentially serious performance problem. In many of our recent production simulations on the CM-5 we have found that the performance of the optimized C code surpasses that of the VUs. Clearly, this challenges the notion that the only way to get performance on the CM-5 is by using the VUs.

### Table 5. Optimized performance on the CM-5 and CM-5E

<table>
<thead>
<tr>
<th>System</th>
<th>Particles</th>
<th>Unmodified</th>
<th>Optimized</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM-5 (33 Mhz SPARC)</td>
<td>1024000</td>
<td>42.63</td>
<td>19.54</td>
<td>119%</td>
</tr>
<tr>
<td>CM-5 (Vector Units)</td>
<td>1024000</td>
<td>-</td>
<td>8.87</td>
<td>- %</td>
</tr>
<tr>
<td>CM-5E (40 Mhz SuperSPARC)</td>
<td>1024000</td>
<td>11.37</td>
<td>6.05</td>
<td>88%</td>
</tr>
<tr>
<td>CM-5E (Vector Units)</td>
<td>1024000</td>
<td>-</td>
<td>6.11</td>
<td>-</td>
</tr>
</tbody>
</table>

8 The Cray T3D

By studying our CM-5 performance, it was our immediate goal to get good code performance on the Cray T3D which uses a 150 Mhz DEC Alpha on each of its processing nodes [12, 13]. The details of porting SPaSM to the T3D have been discussed in detail elsewhere, but currently we have a single version of source code written in ANSI C that runs on both platforms[3]. As result, comparing code compiled on both machines is relatively straightforward. For message passing, we use the Cray shared memory library which provides somewhat better performance than PVM. All code was compiled using the Cray standard C compiler version 4.0.3.2 using an optimization level of -02.

First, we present dynamic instruction set measurements for both the original and optimized versions of code as shown in Table 6. The optimized CM-5 counts are repeated for comparison. We notice that the Cray C compiler effectively inlined the force calculation in the unoptimized code resulting in a fewer number of instructions. More importantly, we see that the modifications for the CM-5 have resulted in a significant drop in the total number of instructions on the T3D. Also notice that the total number of floating point, memory, and control instructions executed is almost identical to the CM-5.
By performing a timing of a 1024000 particle simulation on a 32 node T3D we get a drop in CPU time from 8.57 seconds to 3.70 seconds, a 132% speedup. Our optimizations developed for the CM-5 appear to be quite effective on the T3D.

9 High performance RISC workstations

To see how our optimizations might perform on other systems, a test problem involving 32000 atoms was run on a variety of RISC workstations shown in Table 7. Scaled up to 32 processors, this would correspond to a simulation involving 1024000 atoms so it provides some basis for comparison with our T3D results. The processors selected are also used in parallel supercomputing systems so this provides an indication of how optimizations developed for the CM-5 might help on other parallel machines. In the table, we see that our hand optimizations have resulted in substantial speedups on all systems tested.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Compiler</th>
<th>Unmodified</th>
<th>Optimized</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP-735 (99 Mhz HP-PA 7100a)</td>
<td>cc -0 -Aa</td>
<td>4.68</td>
<td>1.89</td>
<td>148%</td>
</tr>
<tr>
<td>IBM 590 (66 Mhz Rios 2)</td>
<td>xlc -0</td>
<td>4.98</td>
<td>2.58</td>
<td>92%</td>
</tr>
<tr>
<td>SGI Onyx (150 Mhz R4400)</td>
<td>cc -o2 -mips2</td>
<td>6.40</td>
<td>3.21</td>
<td>99%</td>
</tr>
<tr>
<td>SGI Onyx (75 Mhz R8000)</td>
<td>cc -64 -02</td>
<td>7.73</td>
<td>3.47</td>
<td>123%</td>
</tr>
</tbody>
</table>

Table 7. Time for simulating 32000 atoms (cutoff = 2.5σ)

10 Branch and loop optimizations

Unfortunately, the DEC Alpha and other superscalar processors present other types of problems not present on the CM-5. The first of these problems is the handling of conditional branches. In our code, the inner loop of the calculation involves a conditional branch that is taken approximately 80% of the time. This certainly presents problems with certain branch prediction schemes if mispredicted. A second common problem is keeping a superscalar processor busy enough to get high performance. Frequent branch operations certainly don’t help this situation.

To solve some of these problems, we can apply the standard tactic of loop unrolling to our force calculation [5, 6]. However, instead of simply expanding the force calculation into multiple copies, we expand the calculation so that we place as many floating point operations as possible
before the first conditional branch instruction. If we unroll the inner loop by four this results in a code with 33 floating point operations followed by four conditional branches. This does not solve the branch prediction problem, but the long sequence of floating point operations allows for better instruction scheduling and performance. Table 8 shows the effect of loop unrolling on the CM-5, T3D, and workstations.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Problem Size</th>
<th>Optimized</th>
<th>Unrolled</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM-5 (32 nodes)</td>
<td>1024000</td>
<td>19.54</td>
<td>23.53</td>
<td>-17%</td>
</tr>
<tr>
<td>CM-5E (32 nodes)</td>
<td>1024000</td>
<td>6.05</td>
<td>6.19</td>
<td>-2.3%</td>
</tr>
<tr>
<td>T3D (32 nodes)</td>
<td>1024000</td>
<td>3.70</td>
<td>3.18</td>
<td>16%</td>
</tr>
<tr>
<td>HP-735</td>
<td>32000</td>
<td>1.89</td>
<td>1.62</td>
<td>17%</td>
</tr>
<tr>
<td>IBM 590</td>
<td>32000</td>
<td>2.58</td>
<td>1.95</td>
<td>32%</td>
</tr>
<tr>
<td>SGI Onyx (R4400)</td>
<td>32000</td>
<td>3.21</td>
<td>3.78</td>
<td>-15%</td>
</tr>
<tr>
<td>SGI Onyx (R8000)</td>
<td>32000</td>
<td>3.47</td>
<td>2.01</td>
<td>73%</td>
</tr>
</tbody>
</table>

Table 8. The effect of loop unrolling

Here, we see moderate to large speedups on all of the 64 bit architectures, but worse performance on all of the older 32 bit processors. The performance degradation is directly due to the limited number of 64-bit floating point registers (only 16 are available on the SPARC and MIPS R4400). However, for the T3D and HP we get moderate speedups of approximately 16%. On the IBM we get a larger speedup of approximately 30%. Loop unrolling is a clear winner on the R8000 where we have observed speedups over 70%. For larger problems, we have found that loop unrolling pays off even more with performance gains ranging from 40 to over 100%.

11 Fallacies and Pitfalls

Keeping with the style set by [6], we now hope to shed light on several issues that we encountered during our code analysis and optimization.

11.1 Fallacy: Reducing the number of memory operations

Much of our optimization work has focused on reducing the number of memory accesses performed by our code. However, it is possible to carry this to extremes. For example, if we use an outer-loop unrolling scheme instead of inner loop unrolling on the T3D, we notice that we can reduce the number of memory operations by nearly 40% and get a 9% reduction in the total number of instructions performed using inner loop unrolling. This is shown in Table 9. However, timing the code we find that this modification actually results in execution that is approximately 1.5% slower (in some cases, we measured as much as an 8% performance penalty).

11.2 Fallacy: Reducing the number of floating point operations

A second modification has been suggested in the molecular dynamics literature [4]. By adding a filtering mechanism to the code, it is possible to eliminate 13% of the floating point operations in addition to a significant number of memory operations as shown in Table 9. However, when timed, this modification runs 43.7% slower than the code with inner loop unrolling even though it executes fewer floating point operations and fewer memory operations.
### Table 9. Effect of different optimizations on T3D.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Original Count</th>
<th>Inner Loop Count</th>
<th>Outer Loop Count</th>
<th>Reduced FLOP Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>3806976</td>
<td>2045360</td>
<td>2101760</td>
<td>3806976</td>
</tr>
<tr>
<td>FP</td>
<td>20346064</td>
<td>20368424</td>
<td>2038064</td>
<td>17720832</td>
</tr>
<tr>
<td>Memory</td>
<td>7978368</td>
<td>9541332</td>
<td>6839296</td>
<td>6453376</td>
</tr>
<tr>
<td>Control</td>
<td>3389120</td>
<td>2497522</td>
<td>2618880</td>
<td>5097920</td>
</tr>
<tr>
<td>Total</td>
<td>35518528</td>
<td>34452638</td>
<td>31928000</td>
<td>33079104</td>
</tr>
<tr>
<td>Time (1024000 atoms)</td>
<td>3.70</td>
<td>3.18</td>
<td>3.23</td>
<td>4.57</td>
</tr>
</tbody>
</table>

11.3 It’s all about branches anyways...

Our attempts to further reduce memory and floating point operations on the T3D has resulted in code that runs slower even though these code versions execute fewer overall instructions. If we look more carefully at Table 9, we see that our attempts to reduce memory and floating point operations have actually increased the number of control instructions. In fact, if we graph the execution time versus the relative number of control operations, we get the near linear relationship as shown in Figure 2.

![Figure 2. Performance vs. Number of Control Instructions on T3D](image)

12 Conclusions

Recently, there has been a considerable amount of discussion about the difficulty of getting high performance on modern high performance computing systems and the need for new tools and better compilers. Unfortunately, we feel that many of these efforts have resulted in tools that hide the hardware from the user to such an extent that detailed performance analysis is virtually impossible. Rather than relying on these tools, we have found that the best tool for optimizing our code has been a few books on modern computer architecture.
We have demonstrated that it is possible to analyze code execution at a fine level of detail while formulating optimization strategies. Furthermore, we have shown that by making a few simple modifications to our code, we have been able to achieve nearly a 120% speedup on the CM-5, a 168% speedup on the T3D, and a huge speedups on a wide range of RISC workstations. In fact, on the CM-5E, we were able to beat the performance of our code that uses the vector units. Even more important, our modifications have been made to a single version of the source code that runs on all of the platforms used in this paper. Thus, by optimizing for the CM-5, we automatically optimized for a wide range of other machines. This clearly shows that modern architectures are not as radically different as many people claim and that by writing simple and understandable code, it is possible to get very predictable code behavior.

While our primary goal has been to improve the performance of our molecular dynamics code, we hope that we have provided insight into how a particular application performs on a variety of existing machines. As computational scientists, we have felt that the commonly published benchmarks bear little relationship to the performance really seen on any given machine. Rather than citing page after page of Mflop ratings, we hope that we have provided the reader with a unique view of what might be going inside the machine for their application. As for the tool developers, we can only wish that they would give applications programmers more credit and develop more tools for analyzing the truly interesting features of code execution.

Acknowledgements

We would like to acknowledge many people who have contributed to the success of this project. Adam Greenberg, Mark Bromley, Mike Drumheller, Denny Dahl, and Burl Hall of Thinking Machines Corporation provided valuable assistance with understanding the CM-5 architecture and making performance measurements on the CM-5E. Wayne Vierra of Cray Research has provided assistantance with the T3D. We would also like to acknowledge Al Davis and Chris Johnson of the University of Utah, Bruce Wampler of the University of New Mexico, and our collaborators in MD research Niels Grønbech-Jensen, Pablo Tamayo, Timothy Germann, and Brad Holian for their valuable input. Lastly, we acknowledge the generous support of the Advanced Computing Laboratory and the continued assistance of David Rich, Jerry Delap, and Mike Krogh. This work was performed under the auspices of the U.S. Department of Energy.

References


