Lessons learned in benchmarking — Floating point benchmarks: can you trust them?

John Markus Bjørndalen, Otto Anshus
Department of Computer Science, University of Tromsø, Norway
johnm@cs.uit.no, otto@cs.uit.no

Abstract

Benchmarks are important tools for understanding the implication of design choices for systems, and for studying increasingly complex hardware architectures and software systems.

One of the assumptions for benchmarking within systems research seems to be that the execution time of floating point operations do not change much with different input values. We report on a problem where a textbook benchmark showed significant variation in execution time depending on the input values, and how a small fraction of *denormalized* floating point values (a representation automatically used by the CPU to represent values close to zero) in the benchmark could lead to the wrong conclusions about the relative efficiency of PowerPC and Intel P4 machines. Furthermore, a parallel version of the same benchmark is demonstrated to incorrectly indicate scalability problems in the application or communication subsystem.

There is significant overhead in handling these exceptions on-chip on modern Intel hardware, even if the program can continue uninterrupted. We have observed that the execution time of benchmarks can increase by up to two orders of magnitude. In one benchmark, 3.88% denormalized numbers in a matrix slowed down the benchmark by a factor 3.83.

We suggest some remedies and guidelines for avoiding the problem.

1 Introduction

Benchmarks are important tools for studying systems. Understanding the characteristics of a benchmark is important to avoid drawing the wrong conclusions. We report on a problem where the execution time of a textbook benchmark showed significant variations depending on the computed floating point values and how these problems occurred as a warning to practitioners of benchmarking. We also show how the performance characteristics of the benchmark could be misinterpreted, wrongly indicating a scalability problem in the application or communication subsystem, and how a sequential benchmark could lead to the wrong conclusions about the relative performance of a PowerPC and an Intel P4 computer.

Programmers and users tend to ignore that floating point numbers are only approximations of values and how these numbers are represented in hardware. *Denormalized* numbers is a floating point representation that computers today use

automatically for small numbers close to 0 when the result of an operation is too small to represent using the normal representation. This paper shows that computations involving denormalized numbers has a significant performance impact, even when the CPU is configured to mask traps designed to warn the programmer of the denormal numbers. With as little as 3.88 percent of the floating point computations involving denormalized values, the entire computation takes 3.83 times longer to run.

The problem was discovered when the execution time of each process in a cluster benchmark was studied and plotted over time. A strange pattern of varying computation time appeared, which at first made us suspect that there was a bug in the benchmark code.

This paper continues as follows: section 2 describes normalized and denormalized representation of floating point, section 3 describes our benchmark and section 4 our experiment result, section 5 discusses the results, lessons learned and provides some guidelines. Finally we conclude in section 6.

2 Floating point representation, Normalized and Denormalized Form

The IEEE 754 standard for floating point[1] specify the representation and operations for single precision (32-bit) and double precision (64-bit) floating point values. This provides a finite amount of space to represent real numbers that mathematically have infinite precision. Thus, computers only provide an approximation of floating point numbers, and a limited range of values that can be represented. For single precision floating point, the range is about $1.18 \cdot 10^{-38}$ to $3.40 \cdot 10^{38}$ positive and negative.

Limited precision and values that are only approximations create a number of problems, thus there are many guidelines for proper use of floating point numbers. Examples are that floating point numbers should be sorted before summing, and that floating point values should in general not be used as counters in loops.

One of the goals for the designers of the IEEE floating point standard was to balance the resolution and magnitude of numbers that could be represented. In addition, a problem at the time was handling applications that *underflowed*, which is where the result of a floating point operation is too close to zero to represent using the normal representation. In the 70s, it was common for such results to silently be converted to 0, making it difficult for the programmer to detect and handle underflow[2].

As a result, a provision for *gradual underflow* was added to the standard, which adds a second representation for small floating numbers close to zero. Whenever the result of a computation is too small to represent using normalized form, the processor automatically converts the number to a *denormal* form and signals the user by setting a status register flag and optionally trapping to a floating point exception handler. This allows the programmer to catch the underflow, and use the denormalized values as results for further computations that ideally would bring the values back into normalized floating point ranges.

The processor automatically switches between the two representations, so computer users and programmers rarely see any side effects of this, and operations for normal values are also valid for any combination of normal and denormal values. Many programmers may be unaware that their programs are using denormalized values.

Goldberg gives a detailed description of denormalized numbers and many other floating point issues in [3].

Instructions involving denormalized numbers trigger Floating Point Exceptions. Intel processors handle underflows using on-chip logic such that reasonable results are

produced and the execution may proceed without interruption [4]. The programmer can unmask a control flag in the CPU to enable a software exception handler to be called.

The Intel documentation [5], and documents such as [3], warn about significant overhead when handling exceptions in software, but programmers may not expect the overhead to be very high when the floating point exceptions are handled in hardware and essentially ignored by the programmer.

3 Experiments

Jacobi Iteration benchmark

We first found the problem described in this paper in the implementation of a well known algorithm for solving partial differential equations, Successive Over-relaxation (SOR) using a Red-Black scheme. To simplify the analysis in this paper, we use the Jacobi Iteration method, which has similar behavior to the SOR benchmark.

```
for (itt = 0; itt < ITERS; itt++) {
    prev = arr1;
    comp = arr2;
    if (itt % 2 == 1) {
        prev = arr2;
        comp = arr1;
    }
    for (i =  1; i < N - 1; i++)
        for (j = 1; j < N - 1; j++)
            comp[i][j] = 0.25 * (prev[i-1][j  ] + prev[i+1][j    ] +
                                 prev[i  ][j-1] + prev[i  ][j + 1]);
    }
}
```

Figure 1: *Source code for the Jacobi benchmark, tracing time stamps removed.*

An example problem solved by Jacobi is a square sheet of metal represented as a matrix of points, where the temperature of the points on the edges of the sheet are known. Figure 2 shows an illustration of the problem. The temperature of the internal gray nodes are computed from the known values of the boundary (black) points.

The Jacobi iteration method gradually refines an approximation by iterating over the array a number of times, computing each internal point by averaging the neighbors (see figure 3). Two arrays are used: one is the source array, and the other is the target array where the results are stored. When the next iteration starts, the roles of the arrays are swapped (the target is now the source and vice versa).

A solution is found when the system stabilizes. The algorithm terminates when the difference between the computed values in two iterations of the algorithm is considered small enough to represent a solution.

The source code is shown in figure 1. The implementation is a simplified version of Jacobi, running for a fixed number of iterations instead of comparing the newly computed array with the previous array to determine whether the system has stabilized. This simplifies benchmarking and analysis.

Methodology

The experiments were run on the following machines:

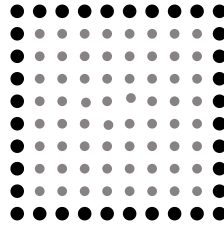

Figure 2: *Heat distribution problem with Jacobi. In a square sheet of metal, the temperature along the edges is known (black points), and the temperature for internal points (gray points) are computed based on the known values for the edges.*

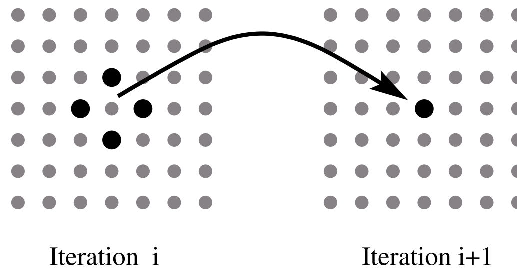

Figure 3: *Jacobi iterative method: in each iteration, the cells in the new matrix are computed as the average of the cells left, right, above, and below from the previous iteration (colored black in the figure).*

- Dell Precision Workstation 370, 3.2 GHz Intel Pentium 4 (Prescott) EMT64, 2GB RAM, running Linux kernel 2.4.21.
- A cluster of 40 Dell Precision Workstation 370, configured as above, interconnected using gigabit Ethernet over a 48-port HP Procurve 2848 switch.
- Apple PowerMac G5, Dual 2.5 GHz PowerPC G5, 4GB DDR SDRAM, 512KB L2 cache per CPU, 1.25GHz bus speed.

Verification of the results on the Dell Precision machine has been run on other Intel-based computers showing similar overheads: a Dell Precision Workstation 360 (3.0 GHz Intel Pentium 4 Northwood, 2GB RAM), a Dell Inspiron 9200 (notebook with 2GB RAM, 2.0 GHz Pentium-M running Linux in VMware) and a IBM Thinkpad T40p (notebook with 1GB RAM, 1.6GHz Intel Pentium-M running Linux in VMware).

The benchmark was compiled with GCC version 3.3.5 with the flags "-Wall -O3" on the Intel architecture. On the PowerMac, GCC 4.0.0 was used with the same flags. Intels C-compiler, version 8.0, was also used with the same flags for the comparisons shown in table 1.

The execution time is measured by reading the wall-clock time before and after the call to the Jacobi solver (using *gettimeofday*), and does not include startup overhead such as application loading, memory allocation and array initialization. All benchmarks are executed with floating point exceptions masked, so no software handlers are triggered.

The benchmarks were run with two sets of input values:

- One that produce denormal numbers: the border points are set to 10.000, and the interior points are set to 0.

- One that does not produce denormal numbers: the border points are set to 10.000 and the interior points are set to 1.

In all cases, we run Jacobi for 1500 iterations with a 750x750 matrix. In addition, the experiment in Table 1 use a matrix where all border and interior points are initialized to a denormal floating point value: 10^{-39}.

Validating the results

To ensure that we didn't report on operating system overhead for handling floating point exceptions, the benchmark was modified to check that FP exceptions were masked, and that the corresponding status registers were updated during the benchmark. The Intel Architecture Software Developer's Manual [4] documents the instructions and registers necessary to do this in chapter 7.

Using these instructions, we also enabled traps from the CPU in one experiment, verifying that we caught the floating point exceptions in a signal handler. These experiments, the documentation, inspection of the registers and additional experiments using VMware (VMware introduces extra overhead in trapping to the operating system) convinced us that the overhead is introduced only by the CPU, and not by software overhead as, for example, with the Intel Itaniums [5].

4 Results

Impact of denormal numbers on Intel Prescott

The experiment was run on a Dell Precision WS 370. The benchmark was instrumented using the Pentium time stamp counters to measure the execution time of each iteration of Jacobi (0-1500), and of the execution time of each row in each iteration.

In a separate run, the contents of the matrices were dumped to disk for later analysis. These dumped values were then used to generate maps of where the denormalized numbers were in the matrices (shown in figure 5) and to count the number of denormal results per iteration, which is shown in figure 6.

Experiments without denormalized numbers

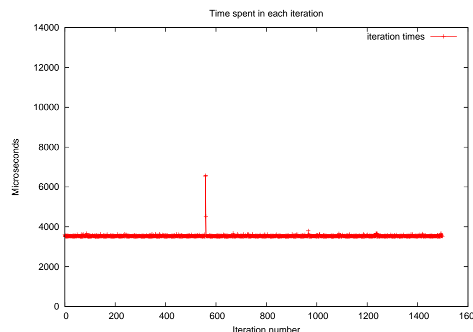

Figure 4: *Computation time for each iteration of Jacobi when no denormalized numbers occur.*

Figure 4 shows the execution time of each iteration of Jacobi when no denormalized numbers occur in the computation. Apart from a spike, the execution time is very stable, as one would expect since the branches in the algorithm are not influenced by the computed values.

Experiments with denormalized numbers

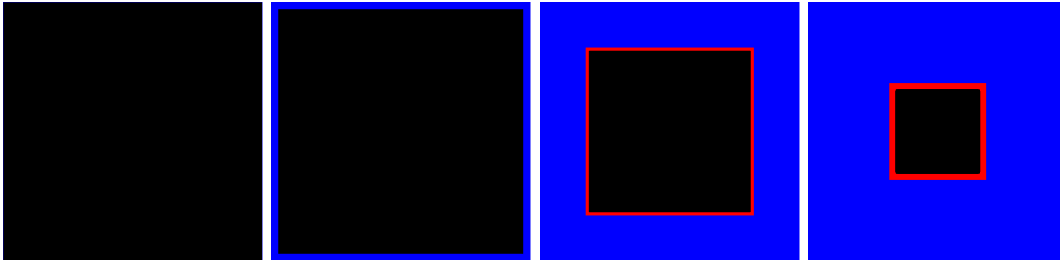

Figure 5: *Graphical representation of the result matrices for four iterations of the JACOBI benchmark (iterations 0, 20, 200 and 600). Black represents 0-values, blue represents non-zero values, and red represents nonzero values in denormal form.*

Figure 5 shows snapshots of the matrix for four iterations of the algorithm. In the beginning there is only a narrow band of non-zero (blue) values. For each iteration, non-zero values gradually spread further inwards in the matrix, but they do so by being averaged with the zero-values in the array. A wavefront is created, with small values pushing inwards towards the center of the matrix, followed by gradually larger values.

After a number of iterations, the values that make up the edge of the wavefront are too small to represent using normalized floating point values, and the CPU automatically switches over to denormalized form (shown as red in the snapshots). The band of denormalized numbers grow in width and sweep towards the center until they eventually disappear.

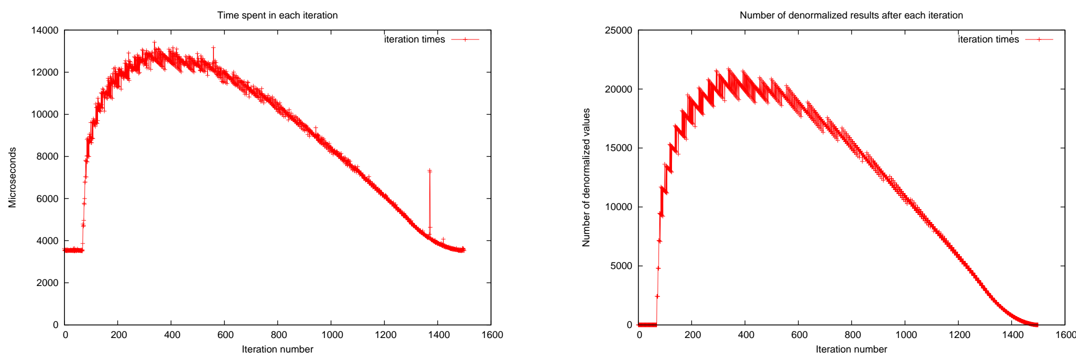

Figure 6: *Left: execution time of each of 1500 iterations. Right: number of floating point values in denormalized form stored in the result matrix on each iteration.*

Figure 6 shows the execution time per iteration of Jacobi when denormalized numbers occur in the computation (left). The figure also shows the number of denormalized numbers in the result array for each iteration (right).

The number of denormalized numbers correspond directly with the execution time per iteration. When denormalized numbers occur, the computation time rapidly increase, and after a while gradually decrease while the number of denormalized numbers decrease. After a while, the last denormalized number disappears and the computation time is back to normal. The jagged pattern on the graph showing the number of denormalized numbers is also reflected in the jagged pattern of the graph showing the execution time.

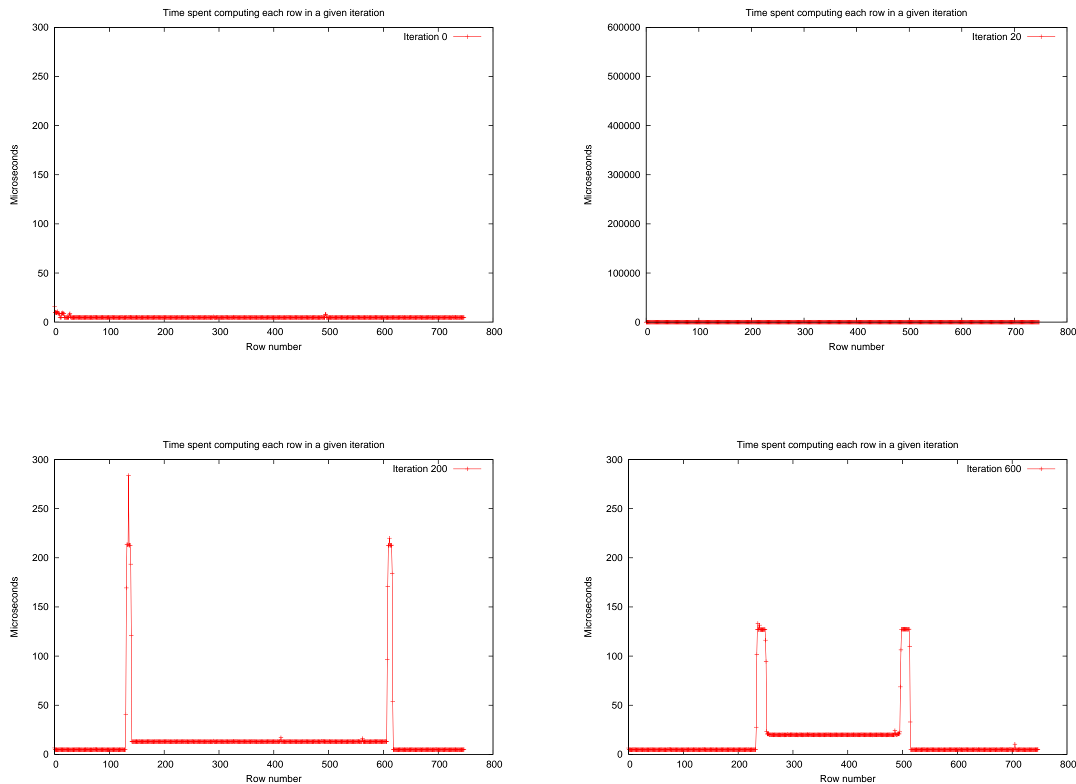

Figure 7: *Computation time for each row in the matrix for four iterations of the algorithm (iterations 0, 20, 200 and 600). The effect of denormalized numbers is clearly visible as rows with denormalized numbers have a higher execution time than rows without denormalized numbers. The computation time depends on the number of denormalized numbers (see lower left, which corresponds to the lower left picture in graph 5).*

Figure 7 shows the computation time of each row of the matrix for the four iterations that are shown in figure 5. The computation time for each row is clearly correlated to the number of denormalized values in that row.

Impact on execution time from denormalized numbers

In every iteration, we compute new values for the $748 \cdot 748 = 559504$ interior points of the matrix. The maximum number of denormalized numbers stored in an iteration was 21716 at iteration 338. That's about 3.88% of the internal numbers computed for that iteration. This is also the iteration that takes the longest to compute at 13427 microseconds. Iterations that have no denormalized numbers typically take around 3509 microseconds, which means that 3.88% denormalized numbers slows down the entire iteration by a factor 3.83.

Machine + BM	min execution time	max execution time
Intel, normal	5.33	5.34
Intel, denormal	13.44	13.47
Intel, all denormal	371.73	373.05
PowerPC, normal	5.04	5.05
PowerPC, denormal	5.40	5.41
PowerPC, all denormal	18.42	18.43
Intel, ICC, normal	4.68	4.70
Intel, ICC, denormal	12.86	12.86
Intel, ICC, all denormal	368.30	368.75

Table 1: *Minimum and maximum execution times of Jacobi on an Intel Prescott EMT 64 3.2GHz and on a PowerPC (Macintosh dual G5) for parameters that result in no denormalized values (normal), up to 4% denormalized numbers (denormal) and where all values computed are in denormalized form (all denormal). Also shown are the execution times using the Intel C Compiler version 8.0.*

The total number of computed cells for all 1500 iterations is $748 \cdot 748 \cdot 1500 = 839256000$. The total number of denormalized cells is 18090464.

PowerPC vs. Intel

On the PowerPC architecture, applications need to trap to the operating system to read the time stamp counter. To avoid extra kernel overhead, the tracing calls for each row were removed, and only the execution time of the entire Jacobi algorithm was measured.

The contents of the arrays were also dumped to disk in a separate run and checked to verify that denormalized numbers occurred.

The minimum and maximum execution times of 5 runs of each benchmark is reported in table 1. The numbers, and some other informal benchmarks on the Intel computer indicate that denormalized numbers introduce an overhead of up to two orders of magnitude if exceptions are masked. If exceptions are unmasked, we have to pay an additional trap to the operating system kernel and potentially a call to a user level signal handler before continuing.

The Intel processor has a higher impact from handling denormalized numbers than the PowerPC machine, with a factor 70 between “normal” and “all denormal” compared to a factor 3.65 on the PowerPC machine.

This means that denormalized numbers occurring in a benchmark may skew the results of a benchmark moved between architectures. As an example, consider the results of the “PowerPC denormal” (5.40) and “Intel denormal” (13.44), which indicate that the PowerPC architecture is more efficient for the Jacobi algorithm. With a different data set, however, the difference is much smaller, as can be seen when the benchmarks only contain normalized numbers (5.04 vs. 5.33).

Parallel Jacobi on a cluster

Figure 8 shows the results of running parallelized implementation of Jacobi using LAM-MPI 7.1.1 [6] using 1 to 8 nodes of the cluster. The implementation divides the matrix into bands that are $1/N$ rows thick and each process exchange the edges of its band with the neighbor processes bands.

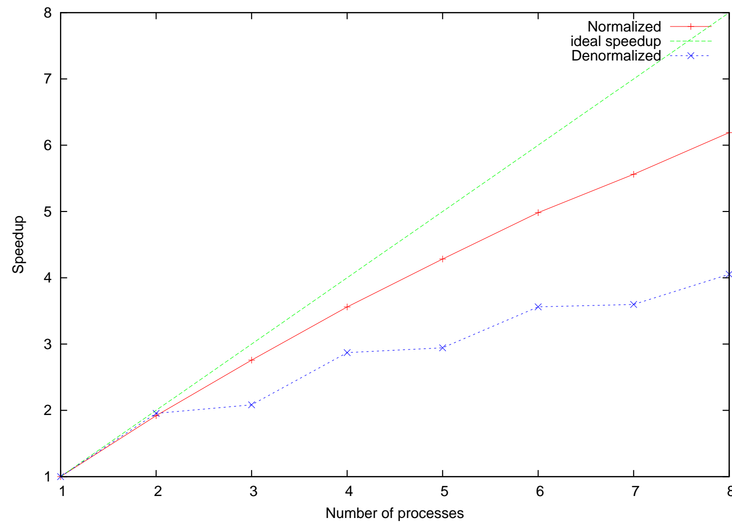

Figure 8: *Speedup of Jacobi on a cluster of 8 nodes, using 1-8 nodes.*

The graph shows that the dataset with denormalized numbers influence the scalability of the application, resulting in a speedup for 8 processes of 4.05 compared to a speedup of 6.19 when the computation has no denormalized numbers.

The step like shape of the "Denormalized" graph is a result of the denormalized band of numbers that move through the rows (as shown in figure 7), influencing at least one of the processes in every iteration. Since the processes are globally synchronized, one process experiencing a tight band of denormalized numbers will slow down the other processes.

This experiment shows that performance anomalies of floating point operations may cause an unaware programmer to attribute too much performance overhead to the parallel algorithm or communication subsystem, as the main change between the sequential and parallel version is the parallelized algorithm and the added communication.

5 Discussion

Normally, users don't care about the computed results in benchmarks as the important part is to have a representative instruction stream, memory access and communication pattern. The assumption is that the execution time of floating point values do not change significantly when using different input values.

Textbooks don't warn against using zero as an initial value for the equation solving methods. One book (Andrews [7] page 535) even suggest that the interior values should be initialized to zero: *"The boundaries of both matrices are initialized to the appropriate boundary conditions, and the interior points are initialized to some starting value such as zero"*. A factor contributing to using the value zero is that students are taught to use zero values for variables to help catching pointer errors, unless they have a good reason for initializing otherwise.

Implication for benchmarks and automatic application tuning

Benchmarks may be vulnerable when varying execution time of floating point operations can mask other effects. Worse even, wrong results from a benchmark may make later analysis based on the data invalid.

An example of automatic application or library tuning is ATLAS [8], where performance measurements are used to automatically tune compilation of the math kernel to a target architecture. If measurements are thrown off, the tuning heuristics may select the wrong optimizations. We have not tested ATLAS for any problems.

Remedies

Denormal values are well-known for people concerned about the accuracy of applications. Some remedies may help or delay the problem: changing from single precision to double precision may delay the point at where denormal values are introduced in our Jacobi benchmark, though we have observed that the band of denormal values will be wider when denormal numbers *do* occur.

Other remedies such as enabling flush-to-zero (converting denormal numbers to zero) are architecture or compiler-specific, and may influence the results of the application.

For this particular benchmark, using nonzero values as initial values for the internal points can remove the problem. The application would also benefit from using multi-grid methods, which could reduce the problem of denormalized numbers considerably.

For benchmarks, checking the status flags for floating point exceptions is one method of detecting that the problem occurred. If the flag is set, the benchmark results should be considered suspect and investigated, or a different dataset should be used.

The problem is that checking for floating point exceptions is a suggestion on the line of checking return values from function calls. Programmers, however, tend to forget checking the return values of functions. We recently heard of an example where a high performance computing application running on a cluster experienced a node failure. The application didn't detect that one of the nodes had died, and the computation continued. Fortunately for the users, the results from the computation were suspicious enough that the users found out about the problem.

An alternative is enabling the software handlers for floating point exceptions, allowing the benchmark to halt if exceptions occur. This is acceptable for some benchmarks and application tuners, but may not be acceptable for all applications. It also requires that benchmark designers are aware of the problem. On the other hand, not handling floating point exceptions may result in an application that is not behaving as designed.

Lessons learned

The most important lesson is: when there is something you can't explain, this is usually a good reason to investigate further, even if you can avoid the problem by choosing different input values. This study is a result of a benchmark problem that a student couldn't resolve in a benchmark that was virtually copied from a textbook. The student solved the problem by choosing a different input data set without understanding the cause of the problem. However, unless you understand the causes, you can not guarantee that the problem will not come back, so the solution was only temporary.

If performance is important, profile with multiple granularities and make sure you understand the performance data. The problem is finding the right granularity for a given problem. Too much overhead from instrumentation may mask the problem investigated and create datasets that are too large for the programmer to cope with.

We wrote a number of special purpose tools to examine and visualize the problem to understand its nature, including tools that create animations of the distribution of denormalized numbers, execution times of individual rows and the magnitude of the values in the arrays. The tools and graphs provide a large set of data and visualizations (both static and animated), and we are currently experimenting with visualizations of the measurement data on a display wall.

Guidelines for design and implementation

To reduce the chances of problems such as the ones identified in this paper, programmers can:

- Turn on floating point exceptions and allow the program to halt and report the problem if these occur. Floating point exceptions should normally not occur in a correctly behaving program, so terminating a benchmark and selecting a different data set or modifying the algorithm is preferable to running a benchmark with performance that varies depending on the input. Care must be taken to mask exception types that should be expected (such as the *precision* exception).
- Do not ignore potential inaccurate or illegal floating point operations in benchmarks. Hardware architects optimize for the common case, and handling exceptions should normally not be the common case.
- Programmers should read and understand the performance guides and software architecture guides for the architectures they program for.
- Make sure you understand the performance behavior of the sequential program before benchmarking parallelized applications. Puzzling behavior of a trivial benchmark may be rooted in the architecture, and not necessarily in the algorithms used.
- Profile at multiple granularities.

Developers should also follow standard development practices for applications, such as turning on the warnings that the compiler can provide. This also goes for developing applications. One environmental simulation application we have worked with accessed arrays outside the legal boundaries in several locations, and a version of the application also has division by zero errors in an equation solver that may or may not have caused problems. Modern compilers can help you catch many of these errors and problems, sometimes with some runtime overhead.

6 Conclusions

As architectures are getting increasingly complex[1] and the cost of using different components in the systems change, benchmarks may be difficult to create and interpret correctly. Optimizing for the common case can introduce unoptimized cases that, even if they occur infrequently, cost enough to significantly impact applications, a point well made by Intel's chief IA32 architect [9].

---

[1] Examples are increasing difference in latency between different levels of caches and memory, and the introduction of Simultaneous Multithreading and Single-Chip Multiprocessors

Benchmarks are often simplified versions of existing applications where the results of the computations are not used since the goal of a benchmark is to measure a systems behavior under a specific use. In these cases a programmer may choose to ignore some of the guidelines for programming floating point applications since the accuracy of unused results are not important, and since the guidelines generally stress the problems around correctness and accuracy of floating point values and operations.

We have shown that one concern, namely floating point exceptions, can easily occur in a benchmark and cause significant overhead even when the exceptions are masked off and handled internally in the CPU. Textbooks and well-known guidelines for programmers contribute to setting the trap for programmers.

We have also shown how a benchmark influenced by floating point exceptions in only a small fraction of the calculations may lead to the wrong conclusions about the relative performance of two architectures, and how a benchmark may wrongly blame the parallel algorithm or communication library for a performance problem on the cluster when the problem is in the applications use of floating point numbers.

In reflection of some of the observed behavior of our simple benchmark, we suggest some guidelines for programmers designing and implementing benchmarks.

References

[1] IEEE 754: Standard for Binary Floating-Point Arithmetic. `http://grouper.ieee.org/groups/754/`.

[2] Charles Severance. IEEE 754: An Interview with William Kahan. *IEEE Computer*, pages 114–115, March 1998. A longer version is available as *An Interview with the Old Man of Floating Point* `www.cs.berkeley.edu/~wkahan/ieee754status/754story.html`.

[3] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[4] Intel. *Intel Architecture Software Developer's Manual – Volume 1: Basic Architecture*. Intel, Order Number 243190, 1999.

[5] Intel. *Intel C++ Compiler and Intel Fortran Compiler 8.X Performance Guide, version 1.0.3*. Intel Corporation, 2004.

[6] LAM-MPI homepage. http://www.lam-mpi.org/.

[7] Gregory R. Andrews. *Foundations of multithreaded, parallel, and distributed programming*. Addison Wesley, 2000.

[8] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[9] Bob Colwell. What's the Worst That Can Happen? *IEEE Computer*, pages 12–15, May 2005.