Trusting floating point benchmarks – are your benchmarks really data independent?

John Markus Bjørndalen and Otto J. Anshus

(johnm, otto)@cs.uit.no
Department of Computer Science, University of Tromsø, Norway

Abstract. Benchmarks are important tools for studying increasingly complex hardware architectures and software systems.
Two seemingly common assumptions are that the execution time of floating point operations do not change much with different input values, and that the execution time of a benchmark does not vary much if the input and computed values do not influence any branches. These assumption do not always hold.
There is significant overhead in handling *denormalized* floating point values (a representation automatically used by the CPU to represent values close to zero) on-chip on modern Intel hardware, even if the program can continue uninterrupted. We have observed that even a small fraction of denormal numbers in a textbook benchmark significantly increases the execution time of the benchmark, leading to the wrong conclusions about the relative efficiency of different hardware architectures and about scalability problems of a cluster benchmark.

1 Introduction

Denormalized numbers is a floating point representation that computers use automatically for numbers close to 0 when the result of an operation *underflows* and is too small to represent using the normal representation. Instructions involving denormalized numbers trigger Floating Point Exceptions, which are handled on-chip on modern processors, producing reasonable results and allowing the execution to proceed without interruption [1]. The programmer can unmask a control flag in the CPU to enable a software exception handler to be called.

Intel documentation [2], 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.

2 Experiments

We first found the problem in an implementation of a well known algorithm for solving partial differential equations, Successive Over-relaxation (SOR) using a Red-Black scheme. To simplify the experiments, we use the Jacobi Iteration method, which has similar behavior to the SOR benchmark. We provide more details about the experiments and how textbooks and general programmer guidelines trap the programmers in [4].

The experiments were run on the following machines:

– **Intel**: Dell Precision Workstation 370, 3.2 GHz Intel Pentium 4 (Prescott) EMT64, 2GB RAM, running Rocks Linux 3.3.0 with Linux kernel 2.4.21 in 64-bit mode.
– **cluster**: A cluster of 40 Dell Precision Workstation 370s, configured as above, interconnected using gigabit Ethernet over a 48-port HP Procurve 2848 switch.
– **PowerPC**: Apple PowerMac G5, Dual 2.5 GHz PowerPC G5, 4GB DDR SDRAM, 512KB L2 cache per CPU, 1.25GHz bus speed.
– **AMD**: AMD Athlon 64 X2 4400+ 2.2GHz, 2GB DDR SDRAM, running Ubuntu Linux 5.10 with kernel 2.6.12-10-k7-smp.

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.

Jacobi executed for 1500 iterations with three different 750x750 matrices as datasets: one that produces about 4% denormal numbers (*denormal*), one that does not produce denormal numbers (*normal*) and one where all input and computed values are in denormal form (*all denormal*). All benchmarks are executed with floating point exceptions masked, so no software handlers are triggered.

3 Results

3.1 Impact of denormal numbers on Intel Prescott

The benchmark was instrumented using the Pentium time stamp counter to measure the execution time of each iteration of Jacobi (1-1500), 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 count the number of denormal results per iteration, shown in figure 1 (right), and to visualize where denormalized numbers occured in the computations.

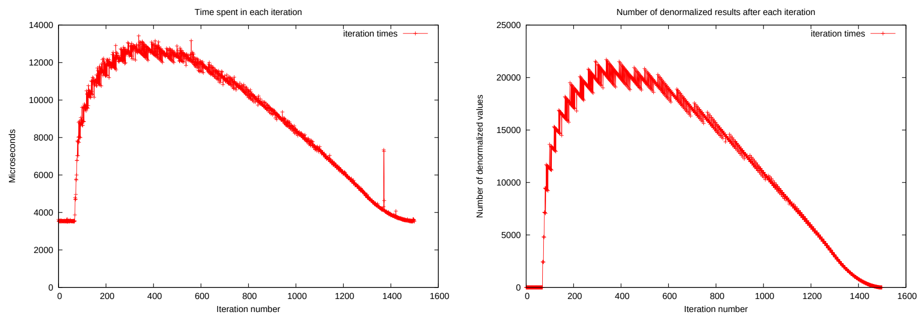

Fig. 1: Left: execution time of each of 1500 iterations. Right: number of floating point values in denormalized form stored in the result matrix in each iteration. In experiments with no denormalied numbers, the graphs show straight horizontal lines.

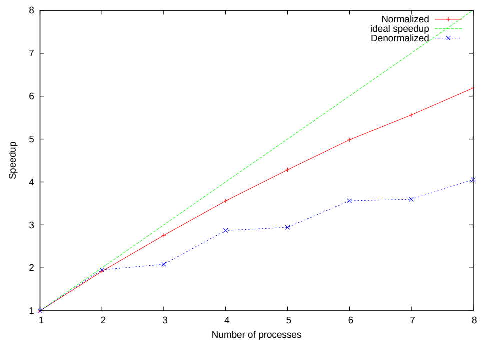

Fig. 2: Speedup of Jacobi on a cluster using 1-8 nodes.

Machine + Dataset	min	max
Intel, *normal*	5.33	5.34
Intel, *denormal*	13.44	13.47
Intel, *all denormal*	371.73	373.05
AMD, *normal*	6.65	6.71
AMD, *denormal*	8.46	8.61
AMD, *all denormal*	88.38	88.99
PPC, *normal*	5.04	5.05
PPC, *denormal*	5.40	5.41
PPC, *all denormal*	18.42	18.43

Table 1: Minimum and maximum execution times of 5 runs of Jacobi.

Figure 1 shows that the number of denormalized numbers per iteration (right) correspond directly with the execution time per iteration (left). When denormalized numbers occur, the computation time rapidly increases, and when the number of denormalized numbers decrease, the computation time decreases correspondingly. 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.

3.2 Impact on comparisons of PowerPC, AMD and Intel P4 Prescott

Table 1 shows that the Intel processor has higher overheads when 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 may influence comparisons of 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 the dataset with normalized numbers, the difference is much smaller (5.04 vs. 5.33).

This problem is more significant for the comparison of AMD vs. Intel, where for the *denormal* case, the AMD processor is faster than the Intel processor, while for the *normal* case, the Intel processor is faster.

This indicates that when moving benchmarks across architectures, corner cases that previously did not influence the benchmarks significantly may start to have an impact, and the benchmarks may not work as originally planned even if the computed results are identical.

3.3 Parallel Jacobi on a cluster

Figure 2 shows the results of running Jacobi using LAM-MPI 7.1.1 [5] with 1 to 8 nodes of the cluster. The implementation divides the matrix into bands that are $1/N$ rows thick and each process exchanges the edges of its band with the neighbor processes.

The graph shows that the dataset with denormalized numbers significantly 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 stairs in the *denormal* graph is a result of a denormalized band of numbers that move through the rows, influencing at least one of the processes in every iteration. Since the processes are globally synchronized, one process experiencing a band of denormalized numbers will slow down the other processes.

4 Conclusions

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 [6].

We have 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.

Effects such as these could potentially cause heuristics that automatically tune libraries for a given architecture, such as ATLAS [7], to select the wrong optimizations.

To determine the scope of the problem, we have started running experiments with known benchmarks to determine whether they are influenced by denormal numbers or similar corner cases. Preliminary experiments with High Performance Linpack (HPL) [8] indicates that it computes denormalized numbers in one section of the benchmark, but we do not yet know whether the benchmark is influenced by this.

Determining the extent of the problem for a number of benchmarks would be simplified with a benchmark validation and profiling tool which could determine whether known hardware corner cases are triggered, and to what extent these cases are triggered. One approach we are investigating is to build such a tool based on the Bochs PC emulator.

References

1. Intel. *Intel Architecture Software Developer's Manual – Volume 1: Basic Architecture*. Intel, Order Number 243190, 1999.
2. Intel. *Intel C++ Compiler and Intel Fortran Compiler 8.X Performance Guide, version 1.0.3*. Intel Corporation, 2004.
3. David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.
4. John Markus Bjørndalen and Otto Anshus. Lessons learning in benchmarking – Floating point benchmarks: can you trust them? November 2005.
5. LAM-MPI homepage. http://www.lam-mpi.org/.
6. Bob Colwell. What's the Worst That Can Happen? *IEEE Computer*, pages 12–15, May 2005.
7. 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.
8. The LINPACK Benchmark, http://www.netlib.org/linpack/.