Quantifying the Interference Caused by Subnormal Floating-Point Values

Isaac Dooley, Laxmikant Kale
Department of Computer Science
201 N Goodwin, Urbana, IL 61801
University of Illinois at Urbana-Champaign
{idooley2, kale}@uiuc.edu

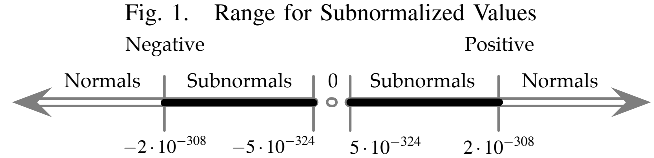
Abstract— **Ordinary floating point operations are slow when subnormal(also called denormalized or unnormalized) floating point values are used or produced. This paper describes how to quantify the extent of the slowdown caused by subnormal values by providing a simple micro-benchmark. Previous work has shown that the slowdown caused by subnormals is significant for some real world applications and that the occurrence of subnormals should not just be treated as an exceptional case[1]. This paper both proposes a micro-benchmark and then analyzes the results for running the proposed micro-benchmark in both C and Java on a variety of modern micro-architectures. The results presented show that all common modern micro-architectures are susceptible to this type of undesirable slowdown.**

I. Introduction

Operating system interference causes serious problems for application developers and system designers. Some seemingly innocuous minor problems can become amplified to overwhelming proportions on parallel machines. Dismal application performance is often the inciting force which leads to the discovery of new types of interference. Various groups have discovered a number of factors which limit the performance of large-scale parallel applications. These problems are significant because they affect some or all jobs run on large expensive high-performance computer systems. The problems so far discovered include OS threads waking randomly[2], [3], [4], improperly scheduling tasks on processors[5], [6], [7], [8], variance in system clocks[9], ignoring the potentially variable and non-deterministic cost of communication in SMPs or clusters[10], and the large time required for OS traps[11].

This paper describes a type of interference or irregularity in computational throughput which is of importance to both single-processor and multi-processor floating-point intensive applications. Simple tests which isolate and quantify the extent of interference-related problems are useful in predicting the optimal performance of an algorithm or application. Consequently, applications can be evaluated with respect to some theoretical model for performance that incorporates known factors such as noise and interference. This paper gives a simple micro-benchmark which can be used to demonstrate and quantify the impact of subnormal floating point values on a computation. The focus of this paper is both to provide a useful micro-benchmark and to evaluate or quantify the performance hit caused by subnormals on a wide range of modern processors.

Fig. 1. Range for Subnormalized Values



Subnormals are a class of floating point values defined by the IEEE 754 standard[12]. They are the range of floating point values closest to 0.0 as shown in Fig. 1. They have the smallest possible exponent, but the non-zero mantissa has at least one leading 0 bit. Since the mantissa contains leading zeroes, the value of a subnormal has fewer significant digits/bits than a normal, and thus a processor may be required to inform an application if a loss in accuracy occurs because a subnormal is produced by a floating point operation. It is widely believed that subnormals are not common in real applications and that if processors don't exhibit drastic performance differences between subnormals and normals, then subnormals can be ignored[13]. We agree with this general belief when considering only sequential applications where subnormals may occur infrequently but not for parallel applications where the slowdown can be amplified by a factor of p as shown in [1]. The belief that subnormals are not important is a guiding principle for processor designers who want their commodity chips to perform well for the common cases. Subnormals can cause serious performance issues for real world applications[1], and thus it is a worthwhile endeavor to quantify the extent of the performance impact caused by subnormals.

II. Subnormal Slowdown Micro-Benchmark

In order to quantify the comparitive effects of subnormals on a variety of processors, a simple freely-available portable micro-benchmark should be used. We wrote a simple micro-benchmark in C that quantifies the slowdown that occurs naturally through a gradual underflow to the subnormal range when averaging a series of 0's and 1's populated in an array. This section briefly describes the micro-benchmark, provides the code for the micro-benchmark, and describes in detail how the micro-benchmark works. Section III will give the results produced by the micro-benchmark on many types of processors. First we provide an algorithm for the micro-benchmark called *Subnormal Slowdown Micro-Benchmark*.

Algorithm *Subnormal Slowdown Micro-Benchmark*

1. $ITER \leftarrow 1000$
2. $SIZE \leftarrow 100000$
3. $tiny \leftarrow$ a small positive normalized value
4. $a \leftarrow \{1.0, 0.0, 0.0, \ldots\}$
5. **for** $i \leftarrow 1$ **to** $ITER$
6.   **for** $n \leftarrow 3$ **to** $SIZE$
7. $a[n] \leftarrow \frac{a[n]+a[n-1]+a[n-2]}{3}$
8. $a \leftarrow \{1.0, tiny, tiny, \ldots\}$
9. **for** $i \leftarrow 1$ **to** $ITER$
10.   **for** $n \leftarrow 3$ **to** $SIZE$
11. $a[n] \leftarrow \frac{a[n]+a[n-1]+a[n-2]}{3}$
12. $time_{slow} \leftarrow$ time for step 5
13. $time_{fast} \leftarrow$ time for step 9
14. $slowdown \leftarrow \frac{time_{slow}}{time_{fast}}$
15. print $slowdown$

The micro-benchmark we created simply creates an array of double precision values, initially setting all entries to 0.0 except for the first entry which is set to 1.0. Then a loop is repeated a fixed number of times. Each loop iteration scans through the array entries from the second 0.0 to the end, replacing each array element with the average of its current value and the the previous two array entries. The two previous array entries were just computed in the same loop iteration. The same process is then repeated with the array values initially being set to 1.0E-50 instead of 0.0. This causes all values in the array to stay inside the normalized range. The two sets of iterations are timed and a slowdown compares the runtime of the former to the later. The former will cause a large number of subnormal values to populate much of the array, while the latter contains only normalized values. A noticable difference in time does occur on all modern platforms that we tested.

The computed final value of the micro-benchmark is the ratio of the time taken for the loop with no subnormals in it to the time taken for the version of the loop with initial conditions for a that will cause subnormals to quickly fill most of the array. This ratio allows us to compare various platforms which have differing processor speeds, pipelining strategies, and mechanisms for handling subnorms. The slowdown ratio does not provide any hint to the absolute speeds of processors when they run the micro-benchmark, and such an analysis is beyond the intent and scope of this paper. We do provide the micro-benchmark code to satisfy the curiosity of the reader who can replicate any specific interesting results.

Both a C and Java version of the micro-bechmark are provided in the Appendix.

Now we describe the actual implementations of the benchmark and how they work on modern processors. The micro-benchmark creates an array named a. It applies an in-place Gauss-Seidel style averaging method which replaces each value with an average of three already computed values. When we run the micro-benchmark and print out the values in a that are produced, we see that the values in the array are converging to $\frac{1}{3}$, which is a steady-state value since $\frac{1}{3} = \left(\frac{1}{3}+\frac{1}{3}+\frac{1}{3}\right)\left(\frac{1}{3}\right)$. Fig. 2 displays the values in the first portion of the array a after exactly 1000 and 2000 iterations. After iteration n, approximately the first $3n$ values in the array a are equal to $\frac{1}{3}$. Of more importance for this paper, however, is the occurrence of subnormal values. When examining the numerical values in the array past the first $3n$ entries, we find an exponential dropoff which can easily be seen in Fig. 3. After some index all array entries become $4.94 \cdot 10^{-321}$ which is a subnormal value. Table I lists numerical values for a sample of points in the array after exactly 1000 iterations. After 1000 iterations with an array size of 100000, we see that 91% of the values in the array have underflowed into the subnormal range. After only one iteration, 97% of the entries in the array are already subnormals. Thus on average about 94% of the array values are subnormal during the first 1000 iterations. Although most of the values are subnormals, the micro-benchmark does not show the absolute worst case slowdowns which could be caused when all array entries are subnormals, but the micro-benchmark is simple and it comes close to the worst case scenario.

Fig. 2. Array values produced by the Subnormal Slowdown Micro-Benchmark. Shows convergence to $\frac{1}{3}$ as number of iterations increases from 1000 to 2000

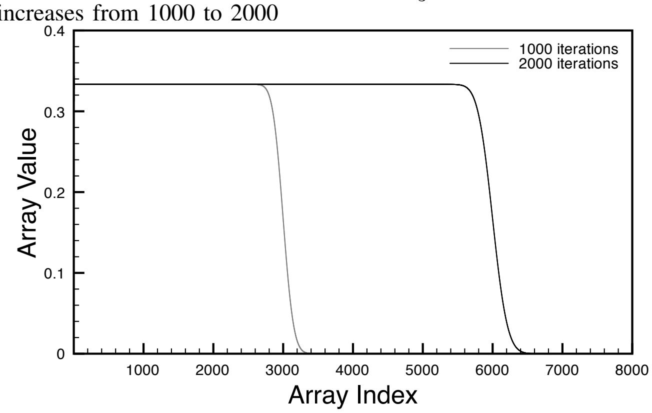
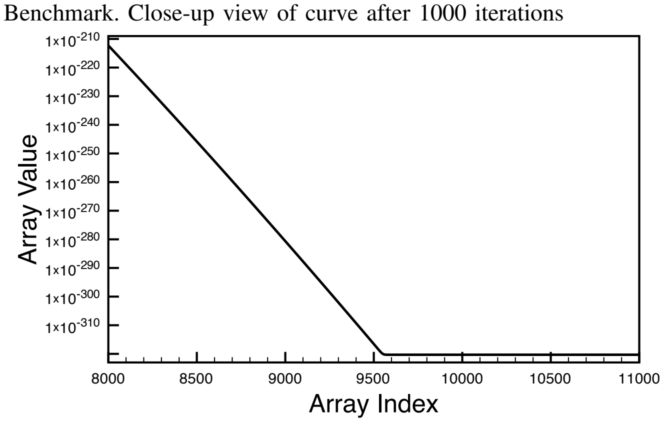


Fig. 3. Array values produced by the Subnormal Slowdown Micro-Benchmark. Close-up view of curve after 1000 iterations



Modern processors do not handle subnormals quickly due to design considerations which optimize the common paths but sacrifice the speed of what are considered exceptional cases by their designers. In the worst case the processors traps to the operating system kernel to handle a floating point operation in software, possibly via a user library which catches a SIGFPE exception. In the best case, an extra cycle may be required to process a subnormal value. The exact methods for handling the subnormals in the processor or in traps to the operating system is not important to an application developer. What is important to an application developer is the cost

TABLE I

ARRAY VALUES PRODUCED BY THE SUBNORMAL SLOWDOWN MICRO-BENCHMARK AFTER 1000 ITERATIONS. SUBNORMALS IN BOLD

Array Index	Slow Version	Fast Version
[0]	1	1
[500]	0.3333333333333	0.3333333333333
[1000]	0.3333333333333	0.3333333333333
[1500]	0.3333333333332	0.3333333333332
[2000]	0.3333333333332	0.3333333333332
[2500]	0.3333318162687	0.3333318162687
[3000]	0.1665328859509	0.1665328859509
[3500]	8.61E-06	8.61E-06
[4000]	1.17E-15	1.17E-15
[4500]	3.17E-30	3.17E-30
[5000]	1.16E-48	1.17E-48
[5500]	2.45E-70	1.00E-50
[6000]	9.52E-95	1.00E-50
[6500]	1.66E-121	1.00E-50
[7000]	2.66E-150	1.00E-50
[7500]	7.09E-181	1.00E-50
[8000]	5.05E-213	1.00E-50
[8500]	1.44E-246	1.00E-50
[9000]	2.31E-281	1.00E-50
[9500]	**2.77E-317**	1.00E-50
[10000]	**4.94E-321**	1.00E-50
[10500]	**4.94E-321**	1.00E-50
[11000]	**4.94E-321**	1.00E-50
[11500]	**4.94E-321**	1.00E-50
[...]	**...**	...
[99999]	**4.94E-321**	1.00E-50

of handling the subnormals. The micro-benchmark we propose provides a good way for a programmer to determine the cost of performing floating point operations on subnormals in any available computer system.

III. PERFORMANCE OF MODERN ARCHITECTURES

It is critical for application programmers and developers of numerical methods to know how modern computer platforms respond when subnormal values arise. This section provides a comparative survey of most modern micro-processor families. We include results from our micro-benchmark described in detail in section II when run on Intel, AMD, Alpha, and Power based systems. For the sake of completeness we ran the micro-benchmark compiled with each available compiler(including icc,xlc,gcc) on each platform with each of the following flags where available: -O, -O2, -O3, -ffast-math, -mieee, -mieee-with-inexact, -mfp-trap-mode=u, -mieee-conformant. We then provide the worst case slowdown for each platform. We did not consider all processor models even within a family because we have not had access to all possible types of machines. Table II lists in sorted order the results of the micro-benchmark on various systems.

TABLE II
WORST CASE SLOWDOWNS ON OUR SUBNORMAL SLOWDOWN MICRO-BENCHMARK ON COMMON MICRO-ARCHITECTURES

Designer	Processor	Slowdown
AMD	K6	1.4
IBM	PowerPC G4	1.4
IBM	PowerPC G4	1.6
IBM	BlueGene/L	1.9
IBM	Power4	2.1
IBM	PowerPC 970	2.4
AMD	AthlonXP	5.5
AMD	Athlon	6.0
AMD	AthlonXP	7.1
Intel	Pentium 3 Xeon	14.5
Intel	Pentium 3	15.8
Alpha	EV67	20.5
AMD	Athlon 64	21.4
AMD	Athlon 64 X2	23.3
AMD	Opteron64	23.8
Alpha	PCA56	31.9
Intel	Core Duo	44.2
Intel	Pentium 4	92.2
Alpha	EV6.8	95.1
Intel	P4 Xeon	97.9
Intel	Pentium 4	131.0
Intel	Itanium 2	183.2
Sun	UltraSPARC IV	520.0

Although we just listed in Table II the worst-case slowdowns over all compilers and compiler flags, our findings contradict the hypothesis that a particularly bad compiler is to blame for any slowdowns. Table III shows the full results for an Intel Xeon 3.2Ghz processor when using both gcc Intel's own icc and icpc. We see small but insignificant variations in the slowdowns, but all are between 82.2 and 97.9. On some platforms, if subnormals are disabled, i.e. flushed to zero(FTZ) by the processor, then the slowdowns can be greatly reduced, since subnormals never occur in the micro-benchmark. However, there are problems with flushing subnormals to zero: some processors do not support FTZ, FTZ will have its own performance cost, and numerical simulations may lose accuracy because a smaller range of floating point values can be represented exactly when FTZ is enabled[14].

The results show that in general, up until recently, the x86 processors by Intel and AMD had been progressively getting worse and worse. The Power family of processors all performed very well, and the modern Sun Ultra-SPARC IV platform performs poorly. The Sun machine does trap to the operating system upon every instruction that produces a subnormal. Thus the slowdown of 520 seems reasonable for modern architectures since a kernel trap will likely take a few hundred cycles, while a pipelined floating point operation on normalized values can take essentially a single cycle.

TABLE III
SUBNORMAL SLOWDOWN MICRO-BENCHMARK RESULTS FOR A 3.2GHZ INTEL XEON USING BOTH GCC AND ICC

Compiler Flags	icc	icpc	gcc
none	92.2	91.3	82.2
-O	89.0	97.9	90.0
-O2	91.9	91.1	90.9
-O3	86.6	87.3	89.0
-ffast-math	n/a	n/a	84.0

The good news, for programmers who require the accuracy provided by preserving subnormal values in their numerical methods, is that the trend for the past decade may be reversing. Modern Intel Core processors are significantly better than their older Pentium counterparts. The Core processors have significantly shorter pipelines than the Pentiums, and thus we suspect the generation of subnormals affect fewer in-flight instructions, which gives better performance. Unfortunately, subnormals are still 55 times slower than normals even on a recent Core Duo processor.

IV. SUBNORMALS IN JAVA VIRTUAL MACHINES

An increasing number of numerical codes are being developed in Java. It has been argued that Java can be a reasonable language for numerical computing if a few major performance bottlenecks are addressed[15]. Because people are using Java for numerical applications, we also quantify how Java programs are affected by subnormal values. Java runs all programs in a virtual machine, and it provides a strict standard for floating point arithmetic. A Java program executing similar instructions to a comparable C program could potentially take a performance hit due to the additional imposed restrictions of the Java floating point model. In order to quantify this impact in performance, we created a Java version of the same micro-benchmark described in section II.

The Java micro-benchmark code is almost unchanged from the C version. The timing routines and code structure obviously must be changed to fit the Java language, but the same algorithm and parameters are used. The micro-bechmark written in Java is provided in its entirety in the Appendix.

This Java micro-benchmark was run on some common platforms with the commonly available java compilers and virtual machines. The code was compiled with *javac* without any command line arguments. It was then run with the available command *java*. Our initial results show that the Java Virtual machine yields performance for the micro-benchmark which is similar to the worst case slowdown for the processor. An Intel Core Duo processor showed less than 1% difference between the slowdown exhibited in the Java and C versions of the micro-benchmark. The PowerPC processor yielded a slowdown 12% better under Java than the worst case for the C version. The Athlon 64 X2 processor yielded a slowdown 22% worse under Java than the worst case for the C version. Similarly with Java the Pentium 4 processor yielded a slowdown that was 21% better than the worst case C version. These results are shown in Table IV. So far it seems that Java's strict floating point requirements do not significantly affect the degree to which operations on subnormals are slower. Although the slowdown factors are similar between the Java and C versions of the micro-benchmark, the absolute times taken for the Java versions were all longer for both the slow and fast portions($time_{slow}$ and $time_{fast}$) of the micro-benchmark.

TABLE IV

WORST CASE SLOWDOWNS ON OUR JAVA VERSION OF THE SUBNORMAL SLOWDOWN MICRO-BENCHMARK

Processor	Worst-Case Slowdown C	Slowdown Java
PowerPC 970	**2.4**	2.1
Athlon XP	5.5	**8.1**
Athlon 64 X2	23.3	**28.5**
Core Duo	45.1	**45.2**
Pentium 4	**131.0**	103.5

In summary, the common Java compilers and virtual machines we used exhibited approximately the same behavior as the C version in regards to the slowdown caused by subnormal values.

V. FUTURE WORK

We do not yet know the extent of the slowdowns caused by subnormals in all application domains. Our previous paper showed the effects of subnormals on a parallel simulation of a 1D wave propagating through a finite 3D bar discretized as an unstructured tetrahedral mesh[1]. The wave propagating through the bar should ideally be a step function, but the chosen discretization cannot perfectly represent the wave. The wave therefore has similar characteristics to the micro-benchmark provided in this paper in that it has a region of subnormal values along the wavefront. This one application led the authors of [1] to discover that subnormals can seriously degrade the performance of a parallel program. We suspect this phenomenon occurs silently in a wide range of diverse applications from all types of domains including Computational Science and Artificial Intelligence. We have been seen the occurrence of subnormals in one machine learning algorithm, but have not rigorously examined this case. Additionally we expect some numerical relaxation techniques to produce many subnormals. As future work we hope to perform a wide survey of applications to determine the true real-world-extent of the performance issues quantified in this paper.

VI. SUMMARY

This paper describes a micro-benchmark that can demonstrate and quantify the performance impact caused by subnormal values in floating point computations. This paper provides code for the micro-benchmark in C and Java, and analyzes the performance of both versions on a wide range of computer systems. It was shown that subnormals significantly degrade performance when they occur frequently on any modern micro-architectures. Quantifying intereference causing problems is critical for predicting performance and achieving optimal performance of high performance applications on modern high performance systems.

APPENDIX

We provide both a C and Java version of the micro-benchmark proposed in this paper. First is the C version for the Micro-Benchmark:

```
/*
 File: subnormal-slowdown-bench.c
 Author: Isaac Dooley

 A program to test how subnormal or
 denormalized floating point values
 affect performance. Test your compiler
 and options with this program.

 This program will print the time of the
 "slower" version, namely the one which
 should contain denormalized values.
 It will also print the time taken by
 the "fast" version, which should
 contain no denormalized values.
 The slowdown is calculated as the
```

```
 ratio of these values.

 http://charm.cs.uiuc.edu/subnormal/
*/

#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>

#define SIZE 100000 /* array size */
#define ITER 1000   /* iterations */

/* Return the time in seconds */
double myTimer() {
  struct timeval tv;
  gettimeofday(&tv, 0);
  return (tv.tv_sec * 1.0) +
         (tv.tv_usec*0.000001);
}

int main(void)
{
 int i, j;
 double tstart, tslow, tfast;
 double *a=(double*)malloc(
           sizeof(double) * SIZE);

 /* Initialize with 0's */
 for (i = 1; i<SIZE; i++) a[i] = 0.0;
 a[0] = 1.0;
 tstart = myTimer();
 for (j=0; j<ITER; j++)
   for (i = 2; i<SIZE; i++)
 a[i] = (a[i] + a[i-1] + a[i-2])*
(1.0/3.0);
 tslow = myTimer()-tstart;

 /* Initialize with small */
 /* normalized values     */
 for (i = 1; i<SIZE; i++) a[i] = 1e-50;
 a[0] = 1.0;
 tstart = myTimer();
 for (j=0; j<ITER; j++)
   for (i = 2; i<SIZE; i++)
 a[i] = (a[i] + a[i-1] + a[i-2])*
(1.0/3.0);
 tfast = myTimer()-tstart;

 printf("time slow: %15.12f fast: "
"%15.12f\n",
tslow, tfast);
 printf("slowdown=%f\n",tslow/tfast );
 return 0;
}
```

The following is the Java code for the Micro-Benchmark, provided in its entirety.

```
/** Gause-Seidel Subnormal
  * Microbenchmark
  * Java Version
  */

class bench {
public static void main(String[] args) {
  System.out.println(
  "Running Gauss-Seidel Micro-bench");

  int SIZE = 100000;
  int ITER = 1000;
  int i, j;
  double tstart, tslow, tfast;
  double []a= new double[SIZE];

  /* Initialize with 0's */
  for (i = 1; i<SIZE; i++)
   { a[i] = 0.0; }
  a[0] = 1.0;
  tstart = myTimer();
  for (j=0; j<ITER; j++){
  for (i = 2; i<SIZE; i++){
     a[i] = (a[i] + a[i-1] + a[i-2])*
     (1.0/3.0); } }
  tslow = myTimer()-tstart;

  /* Initialize with normals */
  for (i = 1; i<SIZE; i++)
   { a[i] = 1.0E-50; }
  a[0] = 1.0;
  tstart = myTimer();
  for (j=0; j<ITER; j++){
  for (i = 2; i<SIZE; i++){
     a[i] = (a[i] + a[i-1] + a[i-2])*
     (1.0/3.0); } }
  tfast = myTimer()-tstart;

  System.out.printf(
   "time slow: %15.12f ", tslow);
  System.out.printf(
   "fast: %15.12f\n", tfast);
  System.out.printf(
```

```
    "slowdown=%f\n",tslow/tfast);
}

static double myTimer(){
  return System.currentTimeMillis()
    / 1000.0;
}
}
```

References

[1] O. Lawlor, H. Govind, I. Dooley, M. Breitenfeld, and L. Kale, "Performance degradation in the presence of subnormal floating-point values," in *Proceedings of the International Workshop on Operating System Interference in High Performance Applications*, September 2005.

[2] F. Petrini, D. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q," in *ACM/IEEE SC2003*, Phoenix, Arizona, Nov. 10–16, 2003.

[3] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, "NAMD: Biomolecular simulation on thousands of processors," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, MD, September 2002, pp. 1–18.

[4] R. G. Minnich, M. J. Sottile, S.-E. Choi, E. Hendriks, and J. McKie, "Right-weight kernels: an off-the-shelf alternative to custom light-weight kernels," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 2, pp. 22–28, 2006.

[5] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *3rd International Conference on Distributed Computing Systems*, 1982, pp. 22–30.

[6] D. G. Feitelson and L. Rudolph, "Parallel job scheduling: Issues and approaches," in *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, D. G. Feitelson and L. Rudolph, Eds., vol. 949. Springer, 1995, pp. 1–18. [Online]. Available: citeseer.ist.psu.edu/feitelson95parallel.html

[7] A. Gupta, A. Tucker, and S. Urushibara, "The impact of operating system scheduling policies and synchronization methods of performance of parallel applications," in *SIGMETRICS '91: Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 1991, pp. 120–132.

[8] A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective distributed scheduling of parallel workloads," in *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 1996, pp. 25–36.

[9] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, os clock ticks, and fine-grained parallel applications," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2005, pp. 303–312.

[10] R. Mraz, "Reducing the variance of point to point transfers in the ibm 9076 parallel computer," in *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 1994, pp. 620–629.

[11] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska, "The interaction of architecture and operating system design," in *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM Press, 1991, pp. 108–120.

[12] IEEE Task P754, *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*, IEEE, New York, 1985.

[13] W. Kahan, "Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic," 1996. [Online]. Available: http://http.cs.berkeley.edu/ wkahan/ieee754status/ieee754.ps

[14] J. Demmel, "Underflow and the reliability of numerical software," *SIAM J. Sci. Stat. Comput.*, vol. 5, no. 4, pp. 887–919, 1984.

[15] J. E. Moreira, S. P. Midkiff, and M. Gupta, "From flop to megaflops: Java for technical computing," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 265–295, 2000.