

Datamation: a quarter of a century and four orders of magnitude later

Paolo Bertasi, Michele Bonazza, Marco Bressan, Enoch Peserico
Department of Information Engineering
University of Padova, Italy
{bertasi,bonazzam,bressanm,enoch}@dei.unipd.it

Abstract—The combination of the high-performance *psort* sorting library and of a carefully tuned desktop-class cluster allowed us to improve the previous record on the Datamation sort benchmark by over an order of magnitude, sorting a million 100 byte records *from disk to disk* in a few dozen milliseconds. Of the many implementation and configuration choices we faced, the most crucial were judicious data placement and access patterns on disk, adoption of UDP sockets instead of MPI, careful pruning of virtually all system daemons, and rejection of “on demand” frequency scaling.

I. INTRODUCTION

This work describes the design and evaluation of a fast, distributed sorting system. Through a careful combination of a small, desktop-class cluster and of a high-performance sorting library it can run the Datamation benchmark (sorting one million 100 byte records *from disk to disk*) in less than 40 milliseconds. This is more than an order of magnitude faster than the previous Datamation record (440 milliseconds [1]), and more than four orders of magnitude faster than what was achieved when the benchmark was introduced, a quarter of a century ago (980 milliseconds [2]).

This section briefly describes the Datamation sort benchmark, and why it remains of interest even today. Section II provides the details of our system and of the experimental setup. Section III examines in depth our performance data, and in particular the effects of different hardware and software configuration choices. Finally, Section IV analyses the significance of our results, before concluding with the bibliography.

A. Datamation sort

The Datamation sort benchmark was one of the three benchmarks introduced in [3] as a measure of a system’s transaction processing power (together with the *Filescan* benchmark, and with the *DebitCredit* benchmark that eventually evolved into the well-known TPC series [4]). It involved sorting one million 100 byte records according to the first 10 bytes *from disk to disk* in as little time as possible, and provided a simple, well-rounded evaluation of most parts of a computing system – processor(s), memory, disks and, for computing clusters, interconnection network.

In 1987, Datamation sort could be run in 980 seconds; but within a decade this had dropped to a handful of seconds. [5] pointed out that startup delays amounted to an ever increasing fraction of the running time of the benchmark; which thus

provided, with each passing year, a progressively less accurate reflection of a system’s *throughput*. [5] proposed two new throughput benchmarks, PennySort and MinuteSort, designed to resist the obsolescence engendered by the ever-increasing performance of computing systems; since then, MinuteSort and PennySort have spawned a large number of heirs [2].

B. A metric for responsiveness

Even though it is no longer really indicative of a system’s throughput, the Datamation sort benchmark remains of great interest even today. Instead of throughput, it now measures a system’s responsiveness and delay in accessing the mass storage layers of its memory hierarchy. This is a crucial performance aspect for a large and growing number of application domains: from web services [6] and cloud computing, to high performance computing such as that of Weather Research and Forecast [7], and from Massive Multiplayer Online Games [8], to distributed large dataset processing [9].

As of today, the two main delay-to-mass-storage bottlenecks in high-performance systems – particularly the most cost-effective ones, which are based on desktop-class computing clusters [10], [11] – tend to be the disks and the network [6], [12], [13]. For the former, careful data placement and access patterns is the key to high performance [14] (this is less true for solid state drives, but their steep price/byte makes it unlikely that they will displace standard platter-based disks as the main mass-storage option for several years). For the network, as the bandwidth/dollar ratio keeps increasing at an exponential pace that often surpasses that of Moore’s Law, the fundamental challenge has become bypassing the limits of protocols (such as TCP) originally designed for long-distance communication [13].

II. SETUP

This section describes our cluster Eridanus, detailing its hardware and software components, and the experimental setup to evaluate its performance. Our goal is to tune Eridanus as carefully as possible, and to measure the impact of a large set of parameters – including hardware, OS, and software parameters – on the overall performance. Subsection II-A describes the system configuration, while Subsection II-B describes the experimental setup.

A. System setup

“Eridanus” is a 16-node cluster, where each node is organized as follows (see Table I). The CPU is a quad-core i7 950 Nehalem running at 3.07GHz, theoretically delivering 48.96Gflops (see [15]). Each CPU embeds four dedicated L1 and L2 caches and one shared L3 cache. The motherboard is an ASUS P6T SE hosting 12GB of DDR3 RAM running at 1600MHz with 9CL-9RCD-9RP-24RAS timings. Banks are accessed in triple channel configuration, for a theoretical maximum transfer rate of 12800MB/s – in practice, this rate is approached only for sequential accesses, while for random accesses it dramatically decreases to around 3500MB/s (see Figure 1).

Each of the 6 SATA II channels is connected to a Samsung Spinpoint F3 HD103SJ, a 1TB 7200RPM hard disk drive with a 32MB cache whose declared average latency is 4.17ms. Judicious placement of, and access to, data on the disks is crucial for performance: data close to the disk’s outer rim can be accessed at a rate almost twice that of data close to the disk’s spindle, and to amortize the rotational and head-movement delays one has to transfer at least 1MB/disk (see Figure 2). We measured a transfer rate of approximately 145MB/s when reading from the portion of the disk nearest to the outer rim; unfortunately, the aggregated transfer rate is limited by the maximum bandwidth (about 680 MB/s) of the motherboard’s ICH10R chipset.

Eridanus nodes are interconnected by two networks. The first is a 1G Ethernet used for remote administration and monitoring tasks. This network relies on a Netgear JGS524 switch providing up to 48Gbps of switching capacity, a forward rate of 1.48M packets per second and a jumbo frame latency of $20\mu s$. The second is a dedicated 10G Ethernet on a Fujitsu-Siemens XG2600 switch, interfaced via a Myricom 10G-PCIE-8B-S NIC and theoretically providing a switching capacity of up to 520Gbps and a latency of up to 300ns. A *netperf* [16] test experimentally confirms a throughput of more than 9.8Gbps for both TCP and UDP-based point-to-point communications.

Each node runs a Debian GNU/Linux OS supported by the 2.6.32 kernel, and hosts a software RAID0 (striping) built over its 6 hard disks. We adopted SGI’s XFS [17] as filesystem, due to its superior performance and limited resource usage (see [18] for a more detailed assessment), leaving to the filesystem tools the detection of the underlying RAID configuration.

Table I
HARDWARE COMPONENT OF THE ERIDANUS CLUSTER.

CPU (per node)	Intel i7 Nehalem @ 3.07GHz
RAM (per node)	12GiB Kingston DDR3 @ 1600 MHz
Motherboard (per node)	ASUS P6T SE
Hard disks (per node)	6 Samsung HD103SJ Spinpoint F3
10G-E NIC (per node)	Myricom 10G-PCIE-8B-S
GPU (per node)	Nvidia GTX 470
10G-E switch	Fujitsu-Siemens XG2600
1G-E switch	Netgear JGS524

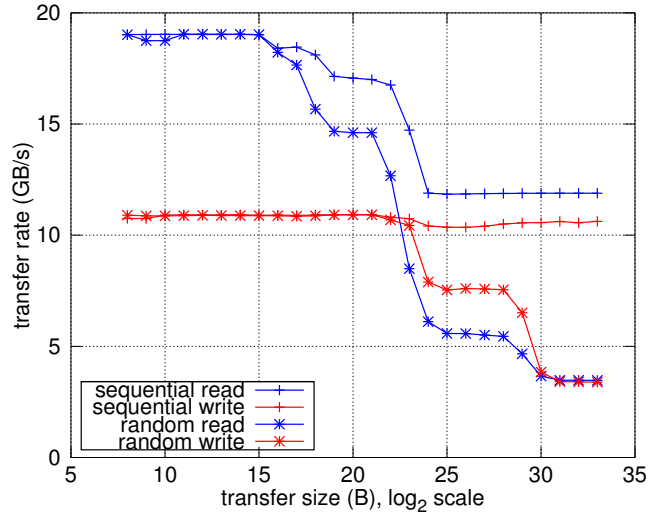


Figure 1. Read and write transfer rates of the main memory of a single Eridanus node, for sequential and random access patterns, as a function of the transfer size.

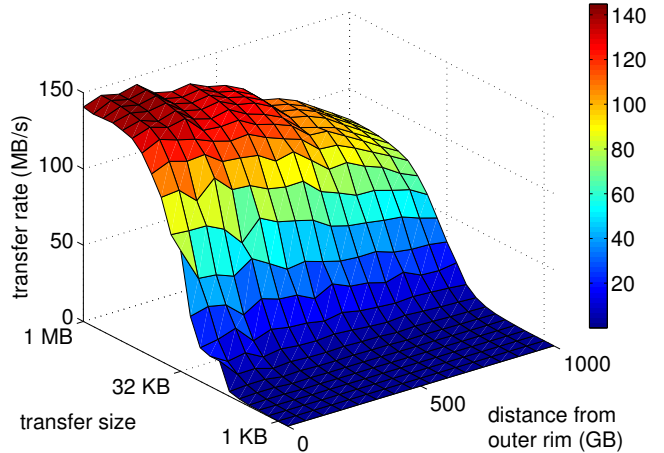


Figure 2. Read transfer rate of a Samsung Spinpoint F3 HD103SJ as a function of the transfer size and of the distance from the disk’s outer rim.

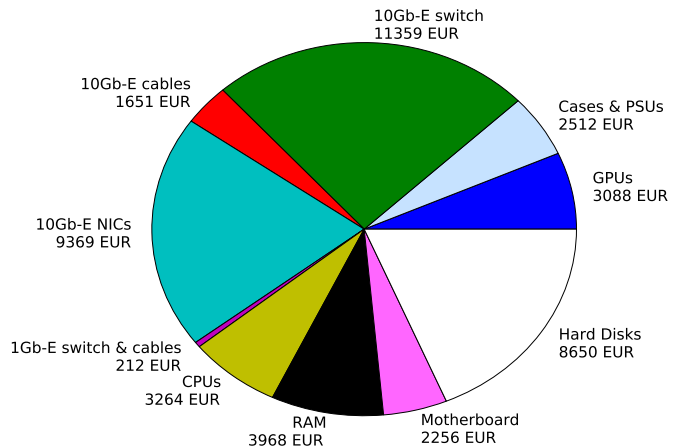


Figure 3. Price breakdown of the Eridanus cluster

B. Experimental setup

Following the Datamation benchmark specifications, we sorted 1 million 100-byte records (according to their 10-byte prefix) from disk to disk. The input, generated by the *gensort* [2] tool, was evenly distributed between the nodes with $10^6/16 = 62500$ records per node. To sort the data we employed an experimental distributed version of *psort* [18], which is currently the fastest stable external sorting software for PC-class machines according to the PennySort benchmark [2].

One can partition the execution of *psort* in three phases: a local bucketing phase, a global distribution phase, and a local sorting phase. In the local bucketing phase, each node reads its input records from the disks and distributes them in 16 different buckets so that each bucket contains approximately the same number of records. In the global distribution phase, each node sends all records of bucket i to node i , for $i = 0, \dots, 15$. Finally, in the local sorting phase, each node sorts the $\approx 6M$ of records received during the distribution phase and writes them back to disk.

The distribution phase deserves a brief note. *psort* performs every communication via UDP sockets, using the standard POSIX routines `sendto()` and `recv()`. Due to the minimal overhead of the UDP protocol, this solution guarantees a very low latency – but at the cost of manually managing the potential loss of datagrams due to congestion. To this purpose, *psort* implements a collision/congestion avoidance mechanisms that divides the distribution phase into several stages, globally synchronized by short UDP messages, guaranteeing that communications always involve only disjoint pairs of nodes. This avoids collisions and congestions and fully exploits our full-duplex links. This solution requires a little bit more effort than using the well-known MPI libraries, but gives substantially better performance (an MPI version of *psort* that we tested gave poor results, see Section III).

We considered two different I/O scenarios. In the “platter” scenario, I/O accesses are performed to the disks *platters* and not only to their on-board caches, and thus the execution time involves moving the disk’s head to the correct track, and waiting for the disk platter to spin the appropriate angle. This is what happens when different runs are required to process different portions of the (potentially large) input file and thus cannot be cached – the most common real-world scenario. Thus, before each single execution of *psort*, we read at least 32MB (the size of a single disk’s cache) from each disk, invalidating the contents its cache, and set the write-through flag of the device via `hdparm`.

In the “cache” scenario, we instead allow the disk to cache the input and output data by avoiding any I/O operation between consecutive executions and without setting any flag. This not only significantly lowered the average execution time, but also drastically reduced the variance which arises from the presence of a large disk population [19], making the behaviour of the rest of the system clearer. This scenario is less realistic, but it strictly matches the original Datamation specification (from disk to disk, rather than from platter to platter); also, it

provides an estimate of the time savings that could be achieved using solid state drives.

We ran *psort* by broadcasting its command line to all nodes, which were running an ad-hoc UDP-based remote minimal shell that spawns processes via `fork()` and `exec()`. The running time of *psort* was measured inside the shell, thus taking into account the process creation/destruction times – which appeared to be around 1ms. For each configuration (see below) we ran 200 separate executions, given the extremely low times (and thus the relatively high jitter) involved. We then measured the minimum and average execution time and its standard deviation.

In two steps, we explored a large portion of the parameter space to find the configuration which minimizes the average execution time of *psort* in the “cache” scenario. First, we manually fixed the values that were evidently optimal – e.g., we used 10G Ethernet instead of 1G Ethernet, and set the maximum CPU frequency – postponing a “sanity check” of their optimality. Then, we extensively (and automatically) tested all remaining parameter combinations which were not evidently suboptimal (such as a heavily downclocked RAM or a RAID stripe size that forces the input to be distributed between only 3 disks). We found a minimum average execution time of less than 39.8ms given by the “reference configuration” summarized in Table II, which for each parameter lists its “reference value” and the set of all tested values (hw/OS defaults in italics).

Table II
REFERENCE SYSTEM CONFIGURATION GIVING THE MINIMAL AVERAGE
psort EXECUTION TIME IN THE “CACHE” SCENARIO.

parameter	optimal value	tested values (<i>default</i>)
CPU clock (GHz)	3.07	<i>on-demand</i> , 1.6, 2, 2.66, 3.07
RAM frequency (MHz)	1600	<i>1066</i> , 1600
RAIDed disks	6	5, 6
RAID stripe size (KiB)	128	32, 128, <i>512</i> , 1024, 2048
HDD read-lookahead	enabled	disabled, <i>enabled</i>
FS read-ahead (sectors)	2048	0, 256, 2048
FS positioning	outer rim	outer rim, inner rim
network type	10Gb-E	1Gb-E, 10Gb-E
switch fwd. mode	cut-through	<i>cut-through</i> , store-and-forward
MTU (byte), 10Gb-E	9000	1500, 8160, <i>9000</i>
MTU (byte), 1Gb-E	—	<i>1500</i> , 7200
NIC coalescence (μ s)	0	0, 75, 150
NIC txqueuelen (byte)	100	100, <i>1000</i> , 10000
MSI	enabled	disabled, <i>enabled</i>

Starting from the reference configuration, we explored the nearby parameter space changing one parameter at a time (i.e. keeping all the others to their reference value) to its hw/OS default value or to the nearest different value(s). We measured the running time of *psort* for each of these alternative configurations, obtaining a measure of the impact of “misconfiguring” each parameter. The next section presents and discusses these results.

III. RESULTS

This section discusses the impact of each system parameter on the overall system performance.

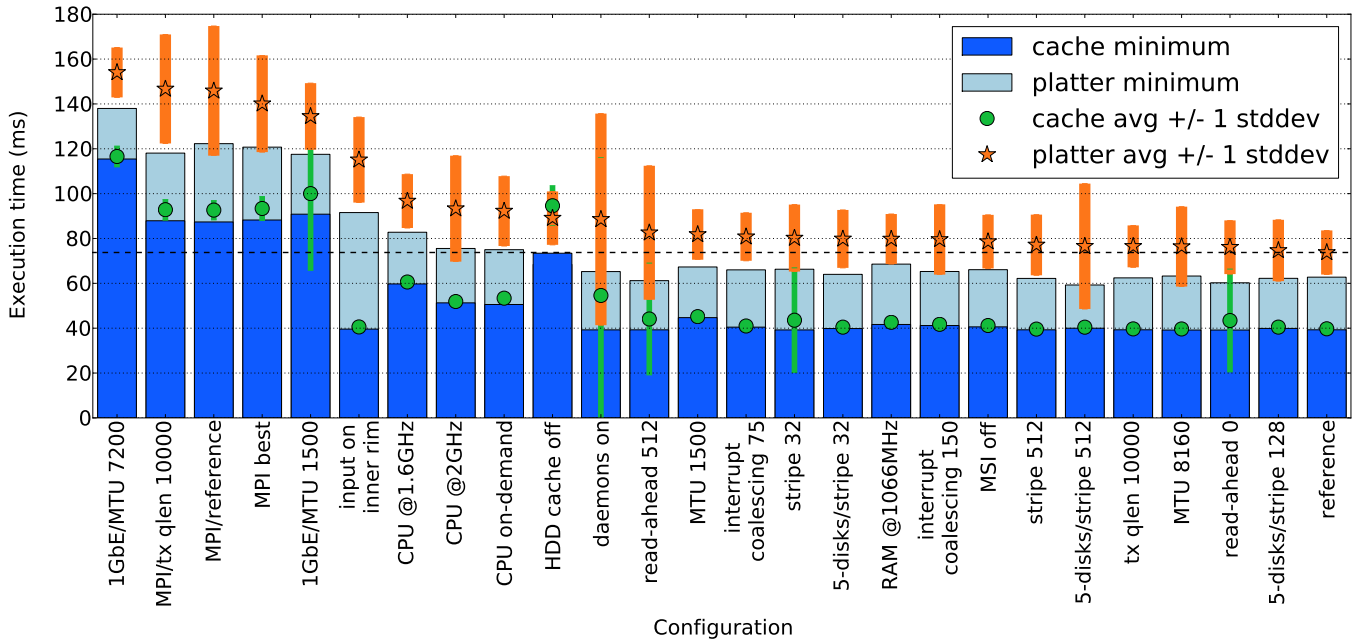


Figure 4. Minimum execution time (out of 200 runs) of *psort*, in the *cache* (dark blue) and *platter* (light blue) scenarios, as a function of the system configuration. The center and extent of the thinner, red and green bars represent instead the average and standard deviation of the execution times.

Figure 4 shows the minimum (solid colour) and average execution time of *psort*, in the cache and platter scenarios, for the reference configuration and 25 different alternative configurations, sorted in decreasing order of the platter average execution time. With an average execution time of less than 39.8ms in the cache scenario and less than 73.8ms in the platter scenario, the reference configuration outperforms every alternative configuration, tying with 3 others (the difference is less than 0.5%) in the cache scenario. Figure 4 clearly shows that some parameters have a far greater impact on performance than others. In order of decreasing importance:

- **Disks.** As can be expected, head movement and rotational delay impose a heavy performance penalty. In every configuration (with the only exception of HDD cache off, when the two scenarios become virtually identical) the platter scenario imposes an overhead of at least 30ms, in one case even exceeding 50ms. Furthermore, it introduces a significant “noise”, increasing the low standard deviation of the cache scenario (0.4ms in the reference configuration) to more than 9ms in all the cases.
- **Network infrastructure.** Switching from the 10G, 9000B MTU Ethernet to the 1G, 7200B MTU Ethernet imposes to both scenarios an overhead of more than 75ms. Perhaps surprisingly, this overhead decreases when a smaller MTU of 1500B is used with 1G Ethernet, but still stands over 60ms.
- **Communication protocols.** We tested an MPICH [20] version of *psort* which performs communication using the MPI routines `MPI_Irecv()` and `MPI_Send()`. We chose to test MPI not only for its widespread use in high-performance computing [21]–[23] and cluster

benchmarking [11], [24]–[26] but also to evaluate its quality as a tool to measure (the influence of parameter tuning on) a system’s responsiveness. It turns out that MPI-based *psort* is significantly slower than standard *psort* (+50ms in the cache scenario and +70ms in the platter scenario). Increasing the `txqueuelen` parameter of the Linux kernel to 10kB, which could in theory improve TCP’s efficiency, did not yield better results. And better configurations for MPI yielded only a marginal improvement – the best MPI configuration we found still incurs a 70ms platter overhead and a 45ms cache overhead.

- **Data placement.** As stated in Subsection II-B, careful data positioning on the disk is crucial to performance: with the input file close to the disk’s spindle, rather than on the outer rim, *psort* needs at least 40 more milliseconds (in the platter scenario – obviously the cache scenario is unaffected).
- **CPU frequency.** Decreasing the CPU frequency to 2GHz and 1.6GHz increases the average execution time by, respectively, ≈ 10 ms and ≈ 20 ms. Interestingly, the “on demand” frequency setting, that should automatically adjust CPU frequency to the minimum level necessary to avoid performance losses, also incurs a 10ms increase – probably because of the low execution times.
- **System daemons.** By default, Debian GNU/Linux comes with a series of daemons (crond, atd, rsyslog, acpid, etc.) that periodically wake up and consume resources – such as CPU or RAM cycles. Besides increasing the average execution time by ≈ 15 ms in both scenarios, these daemons drastically increase its standard deviation to more than 60ms in the cache scenario and 45ms in the

platter scenario.

- **Filesystem read-ahead.** Decreasing the filesystem read-ahead to 512 sectors from the reference 2048 imposes a small but observable overhead of 8ms in the platter scenario and 4ms in the cache scenario; but more surprisingly it takes the standard deviations to, respectively, 30ms and 25ms.
- **Other.** The remaining alternative configurations resulted in smaller differences. These configurations involved incrementing the interrupt coalescing, reducing the RAID stripe size, removing the slowest disk of each node, downclocking the main memory, disabling the message signalled interrupts of the NIC, increasing the transmission queue length of the kernel and disabling the filesystem read-ahead.

IV. CONCLUSIONS

The combination of the high-performance *psort* sorting library, and of a carefully tuned desktop-class cluster, allowed us to improve the previous record on the Datamation sort benchmark by over an order of magnitude, sorting a million 100 byte records *from disk to disk* in less than 40 milliseconds (less than 60 from platter to platter). This suggests that cheap mass storage is fast enough to host data-intensive applications whose response time should be dominated by either Internet delays or by human reaction delays – at least as long as such mass storage is managed carefully, relying heavily on the outer rim of disks and transferring data only in multi-Megabyte chunks.

The process of fine-tuning our cluster for the task also taught us a number of important lessons. First, MPICH is slow even when well configured (and configuring it well is by no means easy). Although it is designed for high performance computing – so that one may naively expect it to incur no greater delays than protocols designed for long distance networks – it is much slower than intelligently used UDP sockets. It also introduces more jitter, raising doubts about the suitability of MPI-based applications as benchmarks for high-performance systems.

Second, daemons on modern OSes are legion; and they can introduce both significant delays, and incredibly high levels of jitter whose source is relatively hard to track down. This puts both the developer and the end-user of a low latency application in a quandary: should one turn off virtually all daemons in the system (including very useful ones such as `atd` or `cron`) and live a life of ascetic but reliable high-performance – or keep at least the most useful daemons, and face significant and unpredictable performance fluctuations?

Third, one should beware of “on demand / automated / self-optimizing” CPU frequency scaling systems promising to pair energy savings during low utilization periods with peak performance at critical points: the response time of such systems may just be too slow for low latency applications, causing serious performance penalties.

V. ACKNOWLEDGEMENTS

This work was possible thanks to a generous equipment donation (all the disks in our cluster) by Samsung. It was also supported in part by Univ. Padova under Strategic Project AACSE, by MIUR under Project AlgoDEEP, and by PAT, FBK, and INFN under Project Aurora-Science.

REFERENCES

- [1] F. I. Popovici, J. Bent, B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Datamation 2001: A sorting odyssey,” University of Wisconsin, Tech. Rep., 2001.
- [2] Sort benchmark home page. [Online]. Available: <http://sortbenchmark.org>
- [3] D. Bitton *et al.*, “A measure of transaction processing power,” *Datamation*, vol. 31, pp. 112–118, 1985.
- [4] TPC benchmarks. [Online]. Available: <http://www.tpc.org/information/benchmarks.asp>
- [5] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet, “Alphasort: a cache-sensitive parallel external sort,” *The VLDB Journal*, vol. 4, pp. 603–628, 1995.
- [6] E. Nahum, T. Barzilai, and D. Kandlur, “Performance issues in WWW servers,” *IEEE/ACM Trans. Netw.*, vol. 10, pp. 2–11, 1999.
- [7] J. Michalakes, J. Hacker, R. Loft, M. O. McCracken, A. Snively, N. J. Wright, T. Spelce, B. Gorda, and R. Walkup, “WRF nature run,” *J. Phys.: Conference Series*, vol. 125, no. 1, pp. 12–22, 2008.
- [8] L. Pantel and L. C. Wolf, “On the impact of delay on real-time multiplayer games,” in *Proc. NOSSDAV*, 2002, pp. 23–29.
- [9] J. Lin, S. Konda, and S. Mahindrakar, “Low-latency, high-throughput access to static global resources within the Hadoop framework,” University of Maryland, Tech. Rep., 2009.
- [10] L. A. Barroso, J. Dean, and U. Hözlze, “Web search for a planet: The Google cluster architecture,” *IEEE Micro*, vol. 23, pp. 22–28, 2003.
- [11] Z. Hill and M. Humphrey, “A quantitative analysis of high performance computing with Amazon’s EC2 infrastructure: The death of the local cluster?” in *Proc. IEEE/ACM GRID*, 2009, pp. 26–33.
- [12] M. R. Swanson and L. B. Stoller, “Low latency workstation cluster communications using sender-based protocols,” University of Utah, Tech. Rep., 1996.
- [13] S. R. Donaldson, J. M. D. Hill, and D. B. Skillicorn, “Performance results for a reliable low-latency cluster communication protocol,” in *Proc. IPPS/SPDP Workshops*, 1999, pp. 1097–1114.
- [14] D. Kunkle, “Roomy: a system for space limited computations,” in *Proc. PASC0*, 2010, pp. 22–25.
- [15] Intel microprocessor export compliance metrics. [Online]. Available: <http://www.intel.com/support/processors/sb/cs-023143.htm>
- [16] Netperf. [Online]. Available: <http://www.netperf.org/netperf/>
- [17] J. Mostek, B. Earl, S. Levine, S. Lord, R. Cattelan, K. McDonnell, T. Kline, B. Gaffey, and R. Ananthanarayanan, “Porting the SGI XFS file system to Linux,” in *Proc. USENIX Freenix*, 2000, pp. 65–76.
- [18] P. Bertasi, M. Bressan, and E. Peserico, “psort, yet another fast stable sorting software,” in *Proc. SEA*, 2009, pp. 76–88.
- [19] E. Pinheiro, W.-D. Weber, and L. A. Barroso, “Failure trends in a large disk drive population,” in *Proc. USENIX FAST*, 2007, pp. 17–29.
- [20] E. Lusk, N. Doss, and A. Skjellum, “A high-performance, portable implementation of the MPI message passing interface standard,” *Parallel Computing*, vol. 22, pp. 789–828, 1996.
- [21] NAS parallel benchmarks. [Online]. Available: <http://www.nas.nasa.gov/Resources/Software/npb.html>
- [22] HPC Challenge. [Online]. Available: <http://icl.cs.utk.edu/hpc>
- [23] Intel MPI Benchmarks. [Online]. Available: <http://software.intel.com/en-us/articles/intel-mpi-benchmarks/>
- [24] A. Bukhamsin, M. Sindi, and J. Al-Jallal, “Using the Intel MPI Benchmarks (IMB) to evaluate MPI implementations on an Infiniband Nehalem Linux cluster,” in *Proc. SpringSim*, 2010, pp. 240:1–240:4.
- [25] L. Chai, Q. Gao, and D. K. Panda, “Understanding the impact of multi-core architecture in cluster computing: a case study with Intel dual-core system,” in *Proc. CCGRID*, 2007, pp. 471–478.
- [26] N. Eicker and T. Lippert, “Low-level benchmarking of a new cluster architecture,” in *Advances in Parallel Computing*, vol. 15, 2008, pp. 381–388.