Measuring NUMA effects with the STREAM benchmark

Lars Bergstrom
University of Chicago
larsberg@cs.uchicago.edu

Abstract

Modern high-end machines feature multiple processor packages, each of which contains multiple independent cores and integrated memory controllers connected directly to dedicated physical RAM. These packages are then connected to one another via an interconnect, creating a system with a heterogeneous memory hierarchy. This design breaks the common illusion for programmers of a uniform address space where any access to global memory takes the same amount of time, ignoring cache issues. Access to memory that is directly connected to the physical processor that the code is running on is not only lower latency (due to a shorter distance) but also provides different bandwidth.

The impact of this heterogeneous memory architecture is not easily understood from vendor benchmarks. Even where these measurements are available, they provide only best-case memory throughput and rely on hand-tuned generation of the perfect assembly instructions to maximize throughput. This work presents concrete measurements of NUMA effects on both a 48-core AMD Opteron machine and a 32-core Intel Xeon machine, obtained using modifications to the well-known STREAM benchmark compiled with the GCC compiler with full optimizations enabled. This version of the benchmark has already been used across industry and academic settings to measure both existing and prototype hardware.

Categories and Subject Descriptors B.3.0 [*Hardware*]: MEMORY STRUCTURES—General; B.8.0 [*Hardware*]: Performance and Reliability—General

General Terms Measurement, Performance

Keywords Memory, NUMA

[Copyright notice will appear here once 'preprint' option is removed.]

1. Introduction

While modern laptops and desktop machines still have a single processor attached to memory, higher-end machines contain either two or four processor packages. These processor packages are each connected directly to a bank of memory chips. Separately, these processor packages have an interconnect between one another. This non-uniform memory architecture (NUMA) design is already the norm in servers, and further increases the importance of writing code with good memory locality.

Typical programming models provide a uniform view of memory. That is, values in memory are not treated differently depending on which physical piece of memory hardware they reside. This model presents two problems. First, physical memory that is closer to the processor provides less latency — ignoring cache effects, less time is required from the time that the fetch is issued until the data arrives at the processor. Second, requests that are further away and use the interconnect between the processors can much more easily fill up the limited bandwidth, resulting in dramtic slowdowns, much in the manner that highways have large bandwidth but slow to a crawl under heavy traffic. With the upcoming Intel Many Integrated Core Architecure [Inta], which provides a large number of x86 cores located on an add-on board, this non-uniformity of latency and bandwidth to access memory is only going to increase.

Further, programmers currently only have spec sheets and manufacturer-provided benchmarks to understand the maximum achievable throughput and minimum latencies. Worse, there is no straightforward way to measure either the latency penalty due to accessing memory far away or the congestion penalty due to saturating the interconnect between processors.

This work makes the following contributions:

1. We describe the architecture of and concretely measure the bandwidth and latency due to the memory topology in both a 48-core AMD Opteron server and a 32-core Intel Xeon server. These results show that NUMA is a feature that programmers need to worry about.
2. We explain the changes we made to the classic STREAM benchmark, both to validate our work and to show how to tune your own code to ensure good memory local-

Component	Hierarchy	# Total
Processor	4 per machine	4
Node	2 per processor	8
Core	6 per node	48

Table 1. Processor topology of the AMD machine.

	Bandwidth (GB/s)
Local Memory	21.3
Node in same package	19.2
Node on another package	6.4

Table 2. Theoretical bandwidth available between a single node (6 cores) and the rest of an AMD Opteron 4P system.

ity, which is the key to achieving good performance on NUMA architectures.

3. We have made our changes to the benchmark available. These changes are already in use in both industrial and academic settings to evaluate hardware.

2. Hardware

In this work, we evaluate two large, server-class machines.

2.1 AMD Hardware

Our AMD benchmark machine is a Dell PowerEdge R815 server, outfitted with 48 cores and 128 GB physical memory. This machine runs x86_64 Ubuntu Linux 10.04.2 LTS, kernel version 2.6.32-27. The 48 cores are provided by four AMD Opteron 6172 "Magny Cours" processors [Car10, CKD$^+$10], each of which fits into a single G34 socket. Each processor contains two nodes, and each node has six cores. The 128 GB physical memory is provided by thirty-two 4 GB dual ranked RDIMMs, evenly distributed among four sets of eight sockets, with one set for each processor. As shown in Figure 1, these nodes, processors, and RAM chips form a hierarchy with significant differences in available memory bandwidth and number of hops required, depending upon the source processor core and the target physical memory location. Each 6 core node (die) has a dual-channel double data rate 3 (DDR3) memory configuration running at 1333 MHz from its private memory controller to its own memory bank. There are two of these nodes in each processor package. This processor topology is also laid out in Table 1.

Bandwidth between each of the nodes and I/O devices is provided by four 16-bit HyperTransport 3 (HT3) ports, which can each be separated into two 8-bit HT3 links. Each 8-bit HT3 link has 6.4 GB/s of bandwidth. The two nodes within a package are configured with a full 16-bit link and an extra 8-bit link connecting them. Three 8-bit links connect each node to the other three packages in this four package configuration. The remaining 16-bit link is used for I/O. Table 2 shows the bandwidth available between the different elements in the hierarchy.

Each core operates at 2.1 GHz and has 64 KB each of instruction and data L1 cache and 512 KB of L2 cache. Each node has 6 MB of L3 cache physically present, but, by default, 1 MB is reserved to speed up cross-node cache probes.

Component	Hierarchy	# Total
Processor	4 per machine	4
Node	1 per processor	4
Core	8 per node	32

Table 3. Processor topology of the Intel machine.

	Bandwidth (GB/s)
Local Memory	17.1
Other Node	25.6

Table 4. Theoretical bandwidth available between a single node (8 cores) and the rest of an Intel Xeon system.

2.2 Intel Hardware

The Intel benchmark machine is a QSSC-S4R server with 32 cores and 256 GB physical memory. This machine runs x86_64 RedHat Enterprise Linux, kernel version 2.6.18-194.11.4.el5. The 32 cores are provided by four Intel Xeon X7560 processors [Intb, QSS]. Each processor contains 8 cores, which can be but are not configured to run with 2 simultaneous multithreads (SMT). This topology is laid out in Table 3. As shown in Figure 2, these nodes, processors, and RAM chips form a hierarchy, but this hierarchy is more uniform than that of the AMD machine.

Each of the nodes is connected to two memory risers, each of which has a dual-channel DDR3 1066 MHz connection. The 4 nodes are fully connected by full-width Intel QuickPath Interconnect (QPI) links. Table 4 shows the bandwidth available between the different elements in the hierarchy.

Each core operates at 2.266 GHz and 32 KB each of instruction and data L1 cache and 256 KB of L2 cache. Each node has 24 MB of L3 cache physically present but, by default, 3 MB is reserved to speed up both cross-node and cross-core caching.

3. STREAM benchmark

The C language STREAM benchmark [McC07] consists of the four operations listed in Table 5. These synthetic memory bandwidth tests were originally selected to measure throughput rates for a set of common operations that had significantly different performance characteristics on vector machines of the time. On modern hardware, each of these

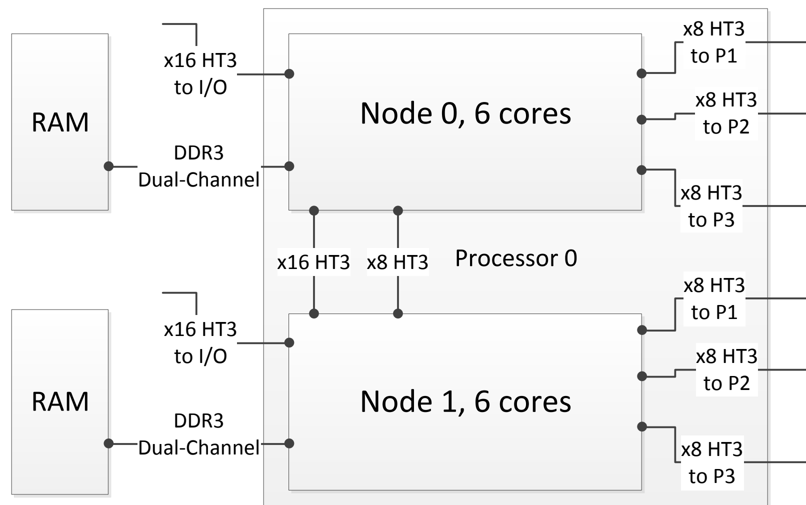


Figure 1. Interconnects for one processor in a quad AMD Opteron machine.

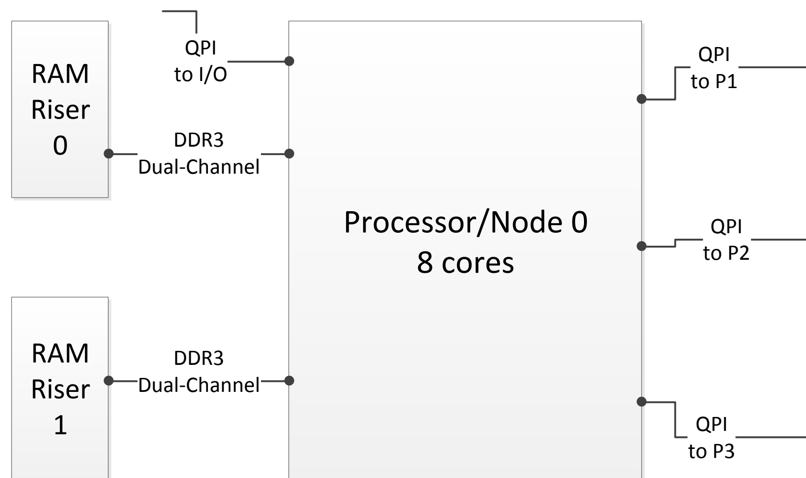


Figure 2. Interconnects for one processor in a quad Intel Xeon machine.

Name	Code
COPY	`a[i] = b[i];`
SCALE	`a[i] = s*b[i];`
SUM	`a[i] = b[i]+c[i];`
TRIAD	`a[i] = b[i]+s*c[i];`

Table 5. Basic operations in the STREAM benchmark. The variables `a`, `b`, and `c` are vectors of floating-point numbers. The variable `s` is a scalar floating-point value.

tests achieve similar bandwidth, as memory is the primary constraint, not floating-point execution.

3.1 Modifications

The modified STREAM benchmark to measure NUMA effects is available at: `https://github.com/larsbergstrom/NUMA-STREAM`

3.2 NUMA-aware access

The existing STREAM benchmark does not support NUMA awareness for either the location of the running code or the location of the allocated memory. We modified the STREAM benchmark to measure the achievable memory bandwidth for these operations across several allocation and access configurations. Our modifications use OpenMP and

libnuma to control the number and placement of each piece of running code and corresponding memory [CDK+01, Kle04].

The baseline STREAM benchmark allocated and then initialized a large, static vector of **`double`** values.[1] The benchark then initializes the values for these vectors. On Linux, Windows, and OSX, the default behavior of the operating system is to allocate physical memory pages from the memory bank attached to the processor where the code was running when it first accessed the page. To exploit this behavior, we use the `numa_run_on_node` function to pin threads to physical nodes.[2] In the default case, we perform all accesses on the same node. For the non-NUMA configurations, accesses are performed on a different node. On Intel machines, any reasonable permutation is sufficient to ensure use of the interconnect (e.g. `bad_node=(node+1)%total_nodes;`). On the AMD hardware, however, adjacent nodes in the same package have sigificantly better access than nodes in different packages. The default source code's behavior does not take this into account, though the reported benchmark numbers used a hand-tuned assignment to ensure the threads were accessing memory on another package.[3]

3.3 Strided access

While the STREAM benchmark's suggested array sizes for each processor are larger than the L3 cache, the tests do not take into account cache block sizes. We extended the tests with support for strided accesses to provide a measure of RAM bandwidth in the case of frequent cache misses. This strided access support also allows us to directly measure the latency of memory access. The only change required to perform this was ensuring that subsequent iterations of the loops use an index far enough away from the previous one to ensure that it was not directly pulled into the cache by a previous read or write.

3.4 Compilation

On both machines, the GCC version is 4.4.3. The benchmark is compiled with optimization level -O3.

4. Bandwidth evaluation

Figure 3 plots the bandwidth, in MB/s, versus the number of threads. Larger bandwidth is better. The results are for the **`COPY`** test, but all of the tests were within a small factor. Four variants of the STREAM benchmarks were used:

1. *Unstrided* accesses memory attached to its own node and uses the baseline STREAM strategy of sequential access through the array.
2. *Unstrided+non-NUMA* accesses the array sequentially, but is guaranteed to access that memory on another package.
3. *Strided* also accesses local memory, but ensures that each access is to a new cache block.
4. *Strided+non-NUMA* strides accesses, and also references memory from another package.

NUMA aware versions ensure that accessed memory is allocated on the same node as the thread of execution and that the thread is pinned to the node, using the libnuma library [Kle04]. To do this, the modified benchmark pins the thread to a particular node and then uses the libnuma allocation API to guarantee that the memory is allocated on the same node. The non-NUMA aware versions also pin each thread to a particular node, but then explicitly allocate memory using libnuma from an entirely separate package (not just a separate node on the same package). When there are less threads than cores, we pin threads to new nodes rather than densely packing a single node.

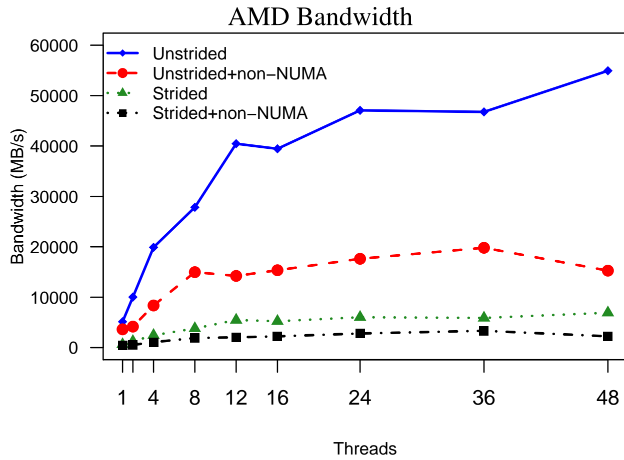
It should not be surprising that the unstrided variants exhibit roughly eight times the bandwidth of their strided versions, as cache blocks on these machines are 64 bytes and the **`double`** values accessed are each 8 bytes. In the NUMA aware cases, scaling continues almost linearly until eight threads and increases until the maximum number of available cores on both machines. On AMD hardware, non-NUMA aware code pays a significant penalty and begins to lose bandwidth where NUMA aware code does not at 48 cores. On the Intel hardware the gap between NUMA and non-NUMA aware code is very small even when the number of threads is the same as the number of cores. But, the Intel hardware does not offer as much peak usable bandwidth for NUMA-aware code, peaking near 40,000 MB/s whereas we achieve nearly 55,000 MB/s on the AMD hardware.

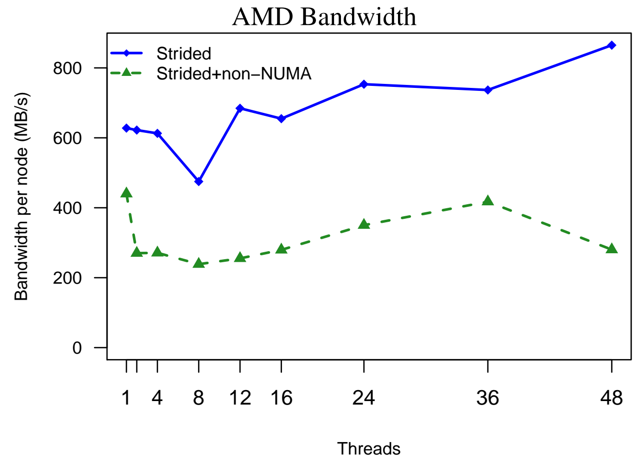
Figure 4 plots the bandwidth against threads again, but this time divided by the number of active nodes to provide a usage data relative to the theoretical interconnect bandwidth detailed in Table 2 for the AMD machine and Table 4 for the Intel machine. Our benchmarks allocate threads sparsely on the nodes. Therefore, when there are less than 8 threads on the AMD machine, that is also the number of active nodes. On the Intel machine with 4 nodes, when there are less than 4 threads, that is the number of active nodes. These graphs show that on both machines there is a significant gap between the theoretical bandwidth and that achieved by the strided **`COPY`** stream benchmark. It is also clear that there is a significant non-NUMA awareness penalty on the AMD machine but that penalty is less on the Intel machine. However, the Intel machine begins to reach saturation at 32 cores,

[1] There is no difference in bandwidth when using **`long`** values.

[2] We use this behavior rather than per-core pinning because variabililty between runs increases dramatically in the per-core configuration with all cores used, due to interference from the operating system processes.

[3] The most straightforward way to determine a good malicious node assignment is by using the `numactl` program. Its `--hardware` flag lists distances between nodes.





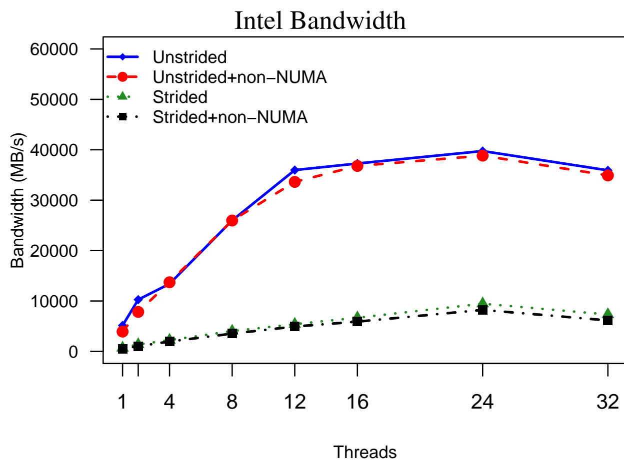


Figure 3. Bandwidth vs. number of threads on the STREAM benchmark, comparing strided and NUMA configurations. Larger bandwidth is better.

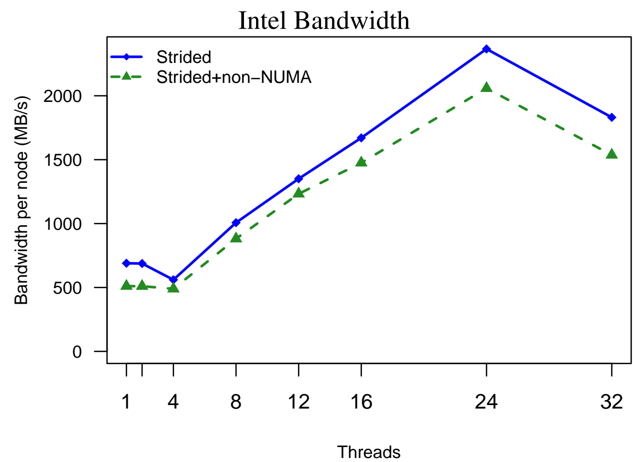


Figure 4. Bandwidth per node vs. number of threads on the STREAM benchmark, comparing strided configurations. Larger bandwidth is better.

whereas the NUMA aware AMD machine continues to increase per-node bandwidth up to 48 cores.

5. Latency evaluation

Figure 5 plots the latency times, in nanoseconds, versus the number of threads. Smaller latency times are better. To measure the latency times, we only consider the strided STREAM benchmarks to ensure that we are measuring only the time to access RAM, and not the time to access cache. As was the case with bandwidth, the AMD machine's NUMA aware tests maintain good values up to large numbers of processors. The non-NUMA aware AMD benchmark begins to exhibit high latencies at moderate numbers of threads. On the Intel machine, latency numbers remain low until more than 24 cores are in use, and then the latencies grow similarly for both NUMA aware and non-NUMA aware code.

On the AMD machine, these benchmarks and evaluations clearly indicate that at high numbers of threads of executions, poor choices of memory location or code execution can have significant negative impact on the latency of memory access and memory bandwidth. For the Intel machine, memory performance uniformly increases until high numbers of threads, at which point is uniformly decreases, seeing little effect from NUMA awareness. But, none of these effects show up in practice until more than 24 threads are in use.

6. Conclusion

We have described and measured the memory topology of two different high-end machines using Intel and AMD processors. These measurements demonstrate that NUMA effects exist. Therefore, memory-intensive workloads that will run on high-end multiprocessor machines should strongly consider engineering their allocations and accesses to achieve good locality and cache use.

Further, we have shown that the NUMA penalty is significantly lower on Intel systems due to the larger cross-processor bandwidth provided by QPI. But, the AMD sys-

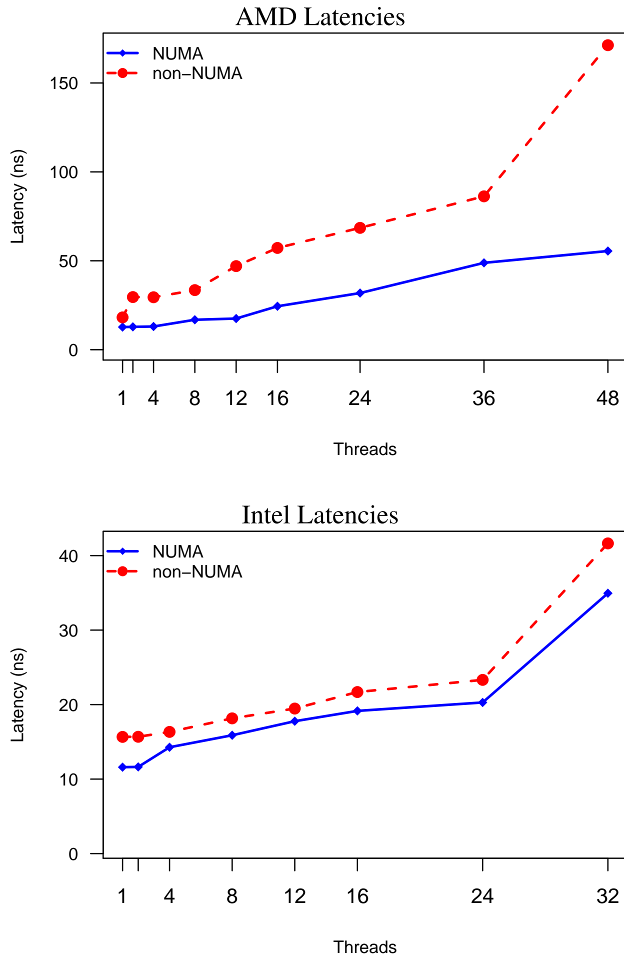


Figure 5. Latency times versus the number of threads on the STREAM benchmark, comparing strided configurations. Smaller latency times are better.

tem provides greater total bandwidth for NUMA-aware applications.

6.1 Related work

The largest effort in measuring and assisting programmers in tuning their codes for better NUMA access is the *Memphis* project [MV10]. It does not seek to measure and report metrics for the underlying hardware, but instead uses processor hardware performance counters to report locality issues with memory usage in code on a given platform. They have successfully applied this tool to high-performance computing codes, resulting in performance improvements between 13% and 24%.

Acknowledgments

Thanks to members of the technical staff at Twitter, Metarstation, and several anonymous financial institutions for additional testing and feedback on a wide variety of NUMA hardware. Andreas Vollemy provided valuable feedback on my changes to the benchmark code.

Thanks also to Bradford Beckmann for reviewing the breakdown of the AMD G34 socket. The AMD machine used for the benchmarks was supported by National Science Foundation Grant CCF-1010568 and this work is additionally supported in part by National Science Foundation Grant CCF-0811389. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

Access to the Intel machine was provided by Intel Research. Thanks to the management, staff, and facilities of the Intel Manycore Testing Lab.[4]

References

[Car10] Carver, T. Magny-cours and direct connect architecture 2.0, March 2010. Available from http://developer.amd.com/documentation/articles/pages/Magny-Cours-Direct-Connect-\-Architecture-2.0.aspx.

[CDK+01] Chandra, R., L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, New York, NY, 2001.

[CKD+10] Conway, P., N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *Micro*, **30**, 2010, pp. 16–29.

[Inta] Intel. Intel Many Integrated Core Architecture.

[Intb] Intel. Intel Xeon Processor X7560. Specifications at http://ark.intel.com/Product.aspx?id=46499.

[Kle04] Kleen, A. A NUMA API for Linux. *Technical report*, SUSE Labs, August 2004. Available from http://halobates.de/numaapi3.pdf.

[McC07] McCalpin, J. D. Stream: Sustainable memory bandwidth in high performance computers. *Technical report*, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[MV10] McCurdy, C. and J. Vetter. Memphis: Finding and Fixing NUMA-related Performance Problems on Multicore Platforms. In *ISPASS '10*. IEEE Computer Society Press, 2010.

[QSS] QSSC. QSSC-S4R Technical Product Specification. Available from http://www.qsscit.com/language_config/down.php?hDFile=S4R_TPS_1.0.pdf.

[4] Manycore Testing Lab Home:
http://www.intel.com/software/manycoretestinglab
Intel Software Network:
http://www.intel.com/software