HPL and DGEMM Performance Variability on the Xeon Platinum 8160 Processor

John D. McCalpin
Texas Advanced Computing Center
University of Texas at Austin
Austin, TX 78758
Email: mccalpin@tacc.utexas.edu

***Abstract*—Initial testing of a cluster equipped with Intel Xeon Platinum 8160 processors showed occasional slow single-node HPL benchmark performance – approximately 0.4% of single-node results were more than 10% slower than expected. We describe a systematic series of experiments using simplified benchmarks and hardware performance counters, showing that the increased run times were associated with increased DRAM traffic, that this was caused by increased L2 cache miss rates, and that these were caused by snoop filter evictions. These evictions resulted from associativity conflicts in the snoop filters, and were traced to pathological interactions of data physical addresses with the hash function that distributes addresses across the coherence agents on the processor. For the HPL benchmark, switching from 2MiB hugepages to 1GiB hugepages eliminated the conflict and the associated slowdowns. The snoop filter conflict was later reproduced using a simple array summation kernel, suggesting that other applications could be impacted.**

I. Introduction

As microprocessors increase in complexity, understanding performance is becoming more difficult [1]. Performance variations, especially those occurring rarely, are especially challenging to identify, diagnose, and work around. This report describes our experience attempting to understand and overcome an infrequently-occurring performance drop that we observed during the bring-up testing of a cluster of Intel Xeon Platinum 8160 processors. In the hopes that this will serve as a useful case study, the presentation is largely chronological, with discussions of how each new set of results combined with our slowly improving understanding of the hardware to lead us to the next step in the investigation.

The performance variations that led to this study were initially observed using an Intel optimized version of the HPL benchmark. This is often used as an acceptance test and as a health checker for servers used in HPC work, and it was in these roles that we first saw unexpected slowdowns. Details will be discussed below, but to give an idea of both the frequency and magnitude of the problem, in single-node testing we typically saw about one out of every 200 nodes delivering performance more than 10% below the median performance for that node. In multi-node HPL testing, all tests using substantially more than 200 nodes delivered results below expectations, and those results were roughly consistent with what one would expect if a few of the nodes were running 10%-15% slower than expected.

II. Test Environment

A. Hardware, Operating System, and Benchmark codes

The test environment consists of two clusters of dual-socket servers equipped with Xeon Platinum 8160 processors [2] running versions of the Linux OS.

Several different versions of the HPL benchmark (provided as binary distributions by Intel) were used. We built several versions of a DGEMM benchmark and a synthetic contiguous array summation code using the two most recent major releases of the Intel C/C++ compiler (and its associated MKL library).

No differences in the statistical characteristics of the performance variability were associated with hardware vendor, OS revision, benchmark code version, or compilers.

Detailed descriptions of the hardware, OS, compilers, and benchmark distributions are provided in Appendix A.

B. Performance Monitoring Infrastructure

The performance monitoring code used in this study was locally developed for non-intrusive, system-wide monitoring[1]. It provides both low overhead (typically $< 1\%$ of one logical processor at 1 second sampling intervals) and very precise control over the programming of the many different performance counters in recent Intel processors [3], [4].

To minimize overhead, the monitoring program is currently single-threaded, and pinned to a single logical processor in the system. The program uses the `/dev/cpu/[nn]/msr` device drivers to read the performance counters on core `nn`. Some of the "uncore" units use MSR access for their performance counters, while others use PCI configuration space. In the latter case, the monitoring code calls the `mmap()` function on `/dev/mem` at the base of PCI configuration space, then ordinary 32-bit array references will read or write the corresponding performance counter registers. The binary is tagged with the "setuid root" property to provide read/write access to the MSR and `/dev/mem` device drivers.

The performance monitoring code is started in the background immediately before the workload of interest is started.

[1]Our use of a "home-grown" performance monitoring code is not intended as a criticism of other performance monitoring infrastructures – we simply find it easier to minimize overhead and ensure complete control over the performance monitoring hardware with our own software.

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

The monitoring code reads a set of configuration files and programs the corresponding performance counters, then goes into a loop of reading all the counters, sleeping for a user-selectable interval, and repeating. The performance counter values are stored in memory until the program being monitored completes and a signal is sent to the monitoring process (or a fixed maximum number of samples have been collected). At this point the collected results are written to disk for later post-processing and the program exits. Detection and handling of counter wraparound is left to post-processing software.

The output of the program is formatted as text array element assignment statements, e.g.,:

imc_counts[1][0]["ACT.ALL"][27] = 87368924

This provides the type of information (integrated memory controller performance counters), the socket number (1), the DRAM channel number (0), the name of the event programmed (ACT.ALL), and the sample number (27). The resulting output files are large, but convenient to work with. We use a combination of *ad hoc* processing with standard text-processing utilities (`awk`, `grep`, `sed`), and embedding the output files directly into Lua scripts.

Due to the large number of hardware performance counters in the system, we generated over 240 GiB (over 4 billion lines) of output for the $> 25,000$ runs performed during this study. Although collecting this quantity of data is clearly overkill for each experiment individually, collecting as much data as possible in each experiment allowed us to go back and re-analyze earlier runs as our understanding of the processor and the performance variability phenomena increased.

III. Initial Experiments with HPL

As part of both the acceptance criteria for the system and as a health check on the nodes, each node in the system is required to deliver a performance in excess of 2100 GFLOPS on the HPL benchmark using the Intel-provided `xhpl` binary for problem sizes in the range of N=100,000. We observed that all nodes were capable of this level of performance (with median performance values in the range of 2275 to 2570 GFLOPS), but that in any run, approximately one out of 200 nodes would fall below the 2100 GFLOPS threshold. Although data was limited in the beginning of the study, it was clear that the slow runs were not limited to a few nodes, and that the occurrence of slow runs was independent of the median performance of the node.

At the same time, we saw disappointing performance in all of our multi-node runs using more than a few hundred nodes, and log files from these runs showed that a few nodes were slow in each run. Multi-node performance followed the general rule of "slowest node times number of nodes", so that a slowdown on any one node can lead to a proportional slowdown on the entire cluster. From a more positive perspective, this also suggests that fixing the node-level performance might fix the multi-node performance shortfalls.

We began our study with simple repetition – running a specific benchmark case (N=148,000, NB=384) four times on each of 392 nodes. The median and average performance

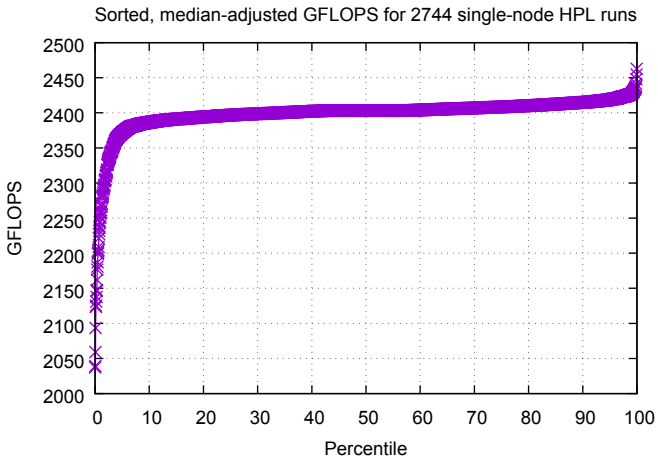

Fig. 1. Sorted median-scaled performance values for 2744 runs of the HPL benchmark (7 runs on each of 392 nodes). The "tail" on the left side indicates a sharp drop in performance for a small fraction of runs, with about 3% of the runs showing some slowdown and about 0.4% of the runs suffering a performance degradation of more than 10%.

of this ensemble were both just above 2400 GFLOPS, with a maximum of 2584 GFLOPS and a standard deviation of 52 GFLOPS. Of the 1568 runs, there were 2 results (from different nodes) below 2100 GFLOPS (more than 5.8 standard deviations below the median), and 7 results (from 6 nodes) below 2200 GFLOPS (more than 3.9 standard deviations below the median). This set of carefully controlled cases was sufficient to convince us that the infrequent slowdowns represented a repeatable feature of the systems, so we set aside time for additional experimentation with hardware performance counter instrumentation.

The second series of HPL runs used the same nodes, the same binary, and the same problem size as the first set, but added the external performance monitoring utility described above for each of the three executions per node. The bulk performance statistics were essentially the same as the first runs. Of the 1176 results, 2 (on different nodes) were below 2100 GFLOPS, and 7 (from 6 nodes) were below 2200 GFLOPS. There was no overlap between the set of nodes that were slow in the first set of runs and the set of nodes that were slow in the second set of runs. This level of agreement in the statistical distribution of results suggests that the performance counter measurements in these runs are from an ensemble showing the phenomenology that we want to investigate.

Analysis of these results was made more difficult by the relatively large range in HPL performance across the nodes – median performance varied by almost 13% across this set of 392 nodes (2277 GFLOPS to 2571 GFLOPS). Once we had accumulated seven HPL results for each node, we were able to scale each performance result by the ratio of the *node median* to the *global median* (2405 GFLOPS) to contract the range of "normal" results and make the slow results more visible. The results are summarized in Figure 1, which shows the sorted, median-scaled performance results for the first 7 runs on each

of the 392 nodes. The tail on the left is the signal of interest. As noted above, the slow runs occur as singular events on different (random?[2]) nodes, and are not due to nodes that are consistently slow.

Our initial performance counter experiments focused on familiar topics: looking for software overheads and looking for variations in DRAM traffic. In a tightly coupled parallel application that uses all the cores of a multicore processor, software overheads can easily have an exaggerated impact, and we had recent experience with tracking down application performance degradation due to unwanted kernel activity. Looking at variations in DRAM traffic seems counterintuitive for a compute-intensive code such as HPL, but the bandwidth required for the HPL benchmark has been increasing along with peak performance, and now requires a significant fraction of the system's memory bandwidth. Performance variation is often associated with cache conflicts, and we had done enough testing to be confident in the accuracy of the hardware performance counters for DRAM accesses.

Optimized implementations of the HPL benchmark will use only one Logical Processor (thread) per Physical Core on recent Intel processors. Each of the "worker" threads of the Intel-optimized benchmark is bound to a separate physical core, but is allowed to migrate between the two Logical Processors that share the core. If the operating system has additional processes to run, it will run them on the "idle" logical processors and cause some level of performance interference with the worker thread sharing the same core. If the sum of the fixed-function CPU_CLK_UNHALTED.REF counters on the two threads of a core exceeds the elapsed TSC cycles, then we know that more than one thread was active during the interval[3].

The first set of runs with performance counters also included a kernel instruction count[4] and an L3 demand data read count for each logical processor[5]. On the "negative results" side, these experiments ruled out kernel activity as the source of performance degradations – kernel cycle and instruction counts were very small and uncorrelated with variations in execution time. The unhalted reference clock counts showed that the average number of threads active per physical core was almost exactly 1.0, as expected for the workload under test. There were positive results as well: the L3 demand data read and DRAM CAS Read counts were slightly correlated with execution time ($r = 0.4$), but more importantly, the two slowest runs (performance reductions of 10% and 8.3%) had the highest counts in both these measures. The L3 demand data read counts seemed too small to account for the magnitude of the slowdown, but this was enough to encourage us to examine traffic through the memory hierarchy in more detail in subsequent experiments.

The second performance counter experiment used a set of core counter events to investigate the role of cache misses. Counters for loads hitting in the L3[6] and loads missing in the L3[7] are straightforward, but must be interpreted carefully because they do not account for data motion caused by the hardware prefetchers – i.e., the data may start in memory, but if one of the hardware prefetchers moves the data into the L3 cache before the demand load arrives, it is counted as an L3 hit, rather than as an L3 miss. With regular memory access patterns, most data traffic through the cache hierarchy is managed by the hardware prefetchers, and a relatively low fraction of demand loads miss in the L2 (or L3) caches. The other two events used in this experiment require a few words of explanation. In modern, out-of-order processors, it is surprisingly difficult to identify the causes of "stalls". Even defining "stalls" can be challenging, as they can occur at instruction fetch, instruction decode, register rename/allocation, micro-op dispatch, or retirement. The last several generations of Intel processors have provided a performance counter event that can be useful in quantifying certain kinds of "stalls". The event causes the counter to increment in any cycle when two conditions are simultaneously true: (1) no micro-ops are dispatched from the reservation station to the execution pipelines, and (2) there is at least one demand load "in flight" that has missed at a selectable level of the cache hierarchy. There is no causality implied by the definition of these events, but in practice, demand loads that miss in the L2 cache or L3 cache require a large number of cycles to complete, and this often causes the processor to exhaust its out-of-order capability and stall. We will refer to these events (imprecisely) as L2 miss stalls[8] and L3 miss stalls[9].

Figure 2 is a scatter plot of (median-adjusted) execution time in seconds vs. DRAM Read traffic for the 1176 results (3 per node) in the second series. The variability of DRAM traffic among the normal results is rather surprising, but the important feature of this plot is the high DRAM Read traffic common to all of the runs more than about 9% slower than the median of 900 seconds. There are 10 such runs (0.85% of the total). A far more interesting correlation is seen with the L2 miss stalls, as shown in Figure 3. After re-scaling by the node's median performance, the execution time shows an extremely high correlation with L2 miss stalls ($r^2 > 0.95$), and a moderate correlation with L3 miss stalls ($r^2 \approx 0.7$). The high correlation with L2 miss stalls, rather than L3 miss stalls, is a critical clue to the mechanism underlying the performance variability that we will discuss in more detail in the following sections.

The third series of single-node HPL runs used the same performance monitoring events as the last two experiments

[2]At this point there were not enough runs to know whether the slowdown could occur on any node, but there was no indication that any nodes might be immune to the problem.

[3]This can be made more precise using a combination of per-thread and ANYTHREAD counts, using straightforward (if not entirely intuitive) algebra.

[4]These footnotes will list the standard performance counter event name from the Intel documentation as well as the specific contents of the performance counter event select register. In this case the event is called INST_RETIRED.KERNEL, and the performance counter event select register is programmed with 0x004200c0.

[5]OFFCORE_REQUESTS.L3_MISS_DEMAND_DATA_READ 0x004310b0.

[6]MEM_LOAD_RETIRED.L3_HIT 0x004304d1.

[7]MEM_LOAD_RETIRED.L3_MISS 0x004320d1.

[8]CYCLE_ACTIVITY_STALLS.L2_MISS 0x054305a3.

[9]CYCLE_ACTIVITY_STALLS.L3_MISS 0x064306a3.

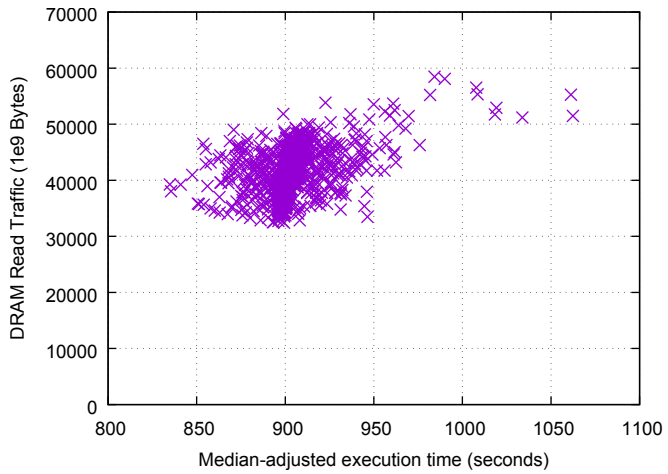

Fig. 2. Median-adjusted HPL execution time vs. DRAM Read Traffic. This represents 1176 executions (3 on each of 392 nodes). Although there is a surprising amount of variability here, it is clear that the ten runs that are more than about 9% slower than the global median have high DRAM Read traffic.

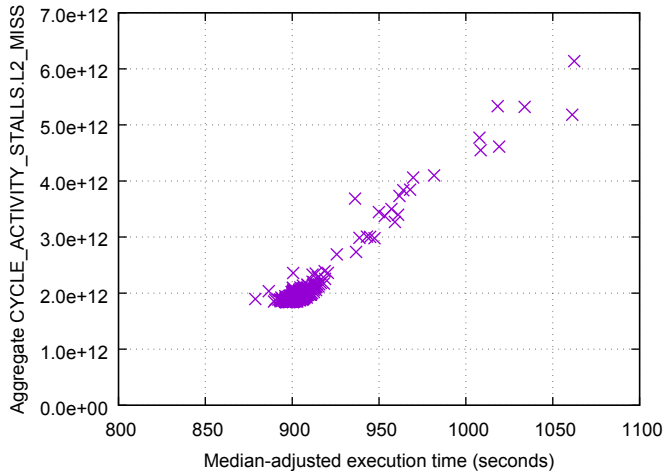

Fig. 3. Normalized HPL execution time as a function of CPU dispatch stalls incurred while at least one demand load was in an L2 miss state. This data represents 784 runs – 2 runs on each of 392 nodes – with 8 runs more than 10% slower than the median and approximately 25 runs showing elevated stalls and correspondingly increased runtime.

in the second series, but used two slightly smaller problem sizes (N=137,000 with NB=384 for two runs and N=86,784 with NB=336 for four runs), and ran the HPL code using one MPI task per socket, rather than launching the executable in its shared-memory mode of execution. These tests confirmed that the infrequent slow runs still exist with different problem sizes and problem decompositions, and that the strongest correlation with execution time comes from the L2 miss stalls performance counter event.

IV. Simplified Experiments with DGEMM

The single-node HPL benchmark has some drawbacks as a benchmark for detailed performance analysis. The source code is not available, the execution consists of a number of dissimilar phases, the problem size (and runtime) must be relatively large to reach asymptotic performance, and the node-level performance depends on power-limited behavior of two independent processor chips (which may have very different average frequencies in the power-limited regime).

To mitigate some of these difficulties, we decided to test a DGEMM benchmark code to see if similar variability in performance was observed. Our code uses the optimized DGEMM function in the Intel MKL library, so the DGEMM function source code is not available, but the algorithm is simpler and the driver is easier to instrument. Table I lists the subset of the DGEMM runs that are discussed in the text. The runs will be referred to by a label of the form "DGEMM 2s A1", where "DGEMM" is the benchmark, "2s" indicates the use of 2 sockets, "A" is the vendor (A or B), and "1" is a sequence number for the set of runs.

First, we needed to see if a two-socket (SMP) DGEMM calculation showed the same performance variability as the two-socket HPL runs. The code was set up to execute 10 calls to DGEMM with N=24,000, using all 48 cores in each node. Timings were recorded for the last 9 calls, and average performance was computed over these 9 calls. Set "DGEMM 2s A1" showed about one of 200 runs delivering 10% to 27% below median performance. The slow runs were all on different nodes. This behavior is clearly consistent with the variability seen with the HPL benchmark.

Set "DGEMM 1s A2" verified that a single-socket DGEMM calculation showed the same performance variability. Just under 1% of the runs delivered 13% to 25% below median performance, and again these slow runs were all on different nodes. This is also clearly consistent with the behavior seen with the dual-socket HPL results.

Set "DGEMM 1s A3" switched from the MKL library distributed with the Intel 17 compiler to the MKL library distributed with the Intel 18 compiler. Again, about 0.6% of runs delivered performance of 10% to 25.0% below the median value. The slow runs were again on different nodes, and there was no overlap between the nodes with slow runs in the single-socket MKL 17 test and the nodes with slow runs in the single-socket MKL 18 test. The median performance increased by about 1%, but the performance variability was unchanged.

Set "DGEMM 1s B4" shifted the runs from the 392 nodes used in the original experiment to a smaller set of 31 nodes from a different vendor to verify that the performance variability remained. We had to discard the first run on each node because the BIOS frequency control was taking too long to ramp the processors up to full speed. This left only two slow runs, but these had the expected signature – in one slow run, each of the 15 DGEMM calls was 28% slower than the median, and in the other slow run, each of the 15 DGEMM calls was 31% slower than the median. These results were sufficient to convince us that the performance variability was present on these nodes as well, leading us to begin the most extensive set of tests.

Label	Problem Size	nodes	runs/node	total runs	median GFLOPS	slow runs	% slow	notes
DGEMM 2s A1	24,000	392	6	2352	2828	12	0.51%	
DGEMM 1s A2	20,000	392	3	1176	1425	11	0.94%	
DGEMM 1s A3	12,000	392	5	1960	1440	12	0.61%	
DGEMM 1s B4	12,000	31	8	248	1406	2	0.81%	
DGEMM 1s B5	20,000	31	273	8463	1421	145	1.71%	
DGEMM 1s B6	20,000	31	62	1922	1268	2	0.10%	reduce to 20 cores
DGEMM 1s B7	20,000	31	103	3193	1421	32	1.00%	
DGEMM 1s B8	20,000	31	41	1271	1419	0	0.00%	1 GiB pages

TABLE I

DGEMM BENCHMARK RUNS MENTIONED IN THE TEXT. THE LABEL INCLUDES THE BENCHMARK, THE NUMBER OF SOCKETS USED, THE VENDOR (A OR B), AND A SEQUENCE NUMBER. THE TOTAL RUNS LISTED ACCOUNTS FOR A SMALL NUMBER OF EXCLUDED RESULTS. SEE TEXT FOR DETAILS.

A. Initial DGEMM instrumented run series

Using a single socket on the target platform, DGEMM delivers close to asymptotic performance levels with problem sizes of N=20,000, which is about an 11 second execution time on a single Xeon Platinum 8160 processor at a sustained performance of 1400 GFLOPS. This reduction in execution time (relative to HPL) allowed us to include multiple DGEMM calls within each benchmark execution. This provided insight into variations within executions vs. variations across executions, which proved to be another critical clue into the nature of the performance problem.

The use of a single-socket DGEMM test case removed much of the uncertainty from the execution of the benchmarks and analysis of the results, providing encouragement for continued experimentation. Although we had suspicions about the source of the performance losses, a large number of tests were still required to rule out other plausible mechanisms.

Set "DGEMM 1s B5" is a composite of 13 sets of 21 executions ("trials") on each node, with each execution containing 21 timed calls to the DGEMM routine. Each run included performance counter instrumentation, with a total of 31 different core performance counters used across the 13 runs. The 21 executions on each of 31 nodes gives a per-run ensemble size of 651 executions. Of these, there were 8–18 executions whose average DGEMM execution time was more than 10% above the median time.

As each run was completed, the performance results were tabulated, with [max, median, average, min, standard deviation] computed across the ensemble of trials for each node and across the ensemble of nodes for each trial. The number of trials delivering average performance below 1350 GFLOPS (about 5% below the median for each run) was collected for each node and accumulated over runs. Performance counter results were summed across all the logical processors of the socket for bulk correlation against execution time.

As we moved through various performance counter groups, the socket-summed performance counters provided these "negative" results:

- L1 Data Cache fills (L1D.REPLACEMENTS) were invariant across runs, so there is no concern about L1 Data Cache conflicts.
- The RESOURCE_STALLS events (Event Code 0xA2) related to the Load Buffer, the Store Buffer, the ReOrder Buffer, the MXCSR register, the Floating Point Control Word, and OTHER are all negligible.

A number of performance counter events showed strong correlations with performance variability, but with counts that are too small to be primary controlling factors. For example, two different performance counter events for demand loads that miss the L3 cache had high correlations with performance variations, but these only accounted for about 3% of DRAM accesses (and only about 3% of the increase in DRAM accesses in the slow runs), so they cannot be the controlling factor here.

We considered, but immediately ruled out, issues related to the TLBs. TLB "reach" was an important factor in DGEMM performance in earlier processors [5], but our experiments already use 2 MiB large pages, and (starting with the "Haswell" core) the second-level TLB (now 1536 entries) can also be used for 2 MiB pages – providing more than $1500x$ the TLB coverage of "Sandy Bridge" or earlier processors using 4 KiB pages.

Several events showed relatively weak global correlations with performance variability, but very strong correlation when considering only the performance outliers. As discussed in several sections above, DRAM reads are moderately correlated with execution time, but all of the outliers have high DRAM traffic. The number of cache lines moved into the L2 cache[10] was measured in five of the thirteen subsets in "DGEMM 1s B5" (3255 trials). All of the slow runs have anomalously high counts here, but not all runs with anomalously high counts are slow. This correlation is shown in Figure 4. The 99% of runs with normal executions times show nearly a 2x range of L2 fill counts, making it difficult for the 1% of the slow runs (all with elevated L2 fills) to create a strong statistical correlation. The dotted line on the right of the figure has a slope of 1 ns per L2 fill. Assuming all 24 cores have one outstanding L2 miss, this corresponds to 24 ns per event. At a frequency of 2 GHz, this is 48 cycles per event, which is comparable to the expected L3 hit latency of 50-70 cycles. This scaling is not meant to imply a conclusion about the specifics of the L2 fills, but it does suggest that the counts are large enough to be a plausible candidate for the primary performance controlling mechanism.

[10]L2_LINES_IN.ALL 0x00431ff1.

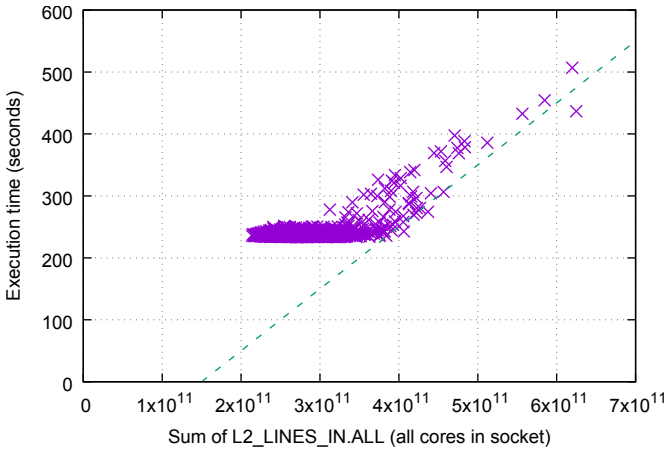

Fig. 4. Sum of all cache lines moved into the L2 caches vs. execution time. This includes 3255 results (105 per node). The dotted line on the right has a slope of 1 ns per count.

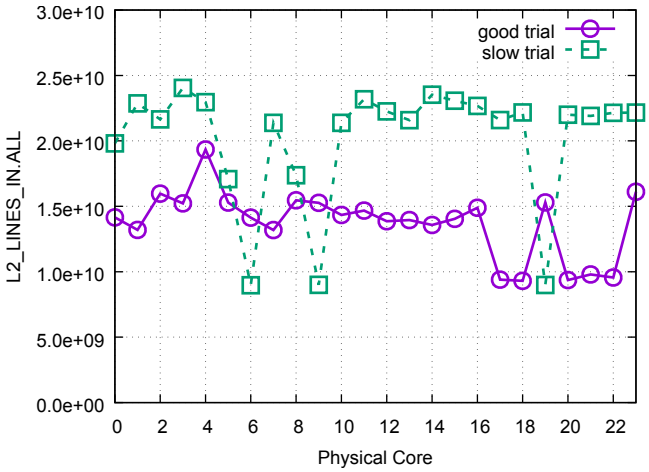

Fig. 5. Variation in L2 cache line fills by physical core for two consecutive executions on the same node. The "slow" trial was 38% slower than the (median-performing) "good" trial.

In addition to the socket-summed performance counter values, for most of the runs, one pair of trials was selected for detailed comparison. These were picked from one node, where a "good" trial (at or above median performance for that node) was followed immediately by a "slow" trial (typically at least 25% slower than the median performance). In the detailed comparisons, all of the available core performance counter data for the socket during the period of the each of the selected trials was aggregated by physical core (rather than by socket), and the patterns were compared. The specific computations depended on the core performance counter events being collected in that run.

As an example using the L2 fill event discussed above, Figure 5 shows the counts per physical core for two consecutive trials on the same node. The "good" trial ran at very close to the median performance, while the "slow" trial required a 61% longer execution time and 44% more DRAM reads. In the "slow" run, cores 6, 9, and 19 ran at higher frequencies, suggesting that they completed early and entered spin-waits[11]. There was no frequency variation across cores in the "good" run, despite the 2x variation in L2 fills.

Finally, we investigated some undocumented counters that Intel has only disclosed by name at the https://download.01.org/perfmon/ site. These events are called "CORE_SNOOP_RESPONSE" (Event 0xEF), with various Umasks to specify particular transactions. Although these events are only documented with the word "tbd", we found the same nomenclature described in the uncore performance monitoring guide [4] in the context of the CHA performance counter events (discussed in more detail in the next section). We tested all the sub-events and found strong correlations between performance variations and sub-events named "RSP_IFWDFE" and "RSP_IHITFSE", and moderate correlations with the sub-event "RSP_IFWDM". These events were very small in the normal runs, but increased by factors of several hundred in the slow runs, until they were comparable to the (elevated) L2 fill counts. Using the descriptions of the CHA events as a guideline, we guessed that these events are related to cache eviction requests processed by the core's L1 and L2 caches, so we decided to add CHA counters to look at these eviction events from their source.

[11]Maximum frequency on the Xeon Scalable processors is a function of the width of the SIMD registers in use. The DGEMM computational code uses 512-bit registers and runs at the lowest frequencies. The spin-waiting code has no reason to use SIMD registers, allowing spin-waiting cores to transition to higher frequencies.

B. Forming a Hypothesis

Two properties of the results thus far provided fairly strict bounds on the classes of mechanisms that might be responsible for the performance variations. First, the performance was constant during each run, with variability between runs. This "meta-stable" behavior suggests that the performance is being controlled by the physical addresses that are assigned when virtual to physical mappings are set up as pages are instantiated at the beginning of each execution. Second, the behavior of the L2 caches varied strongly across cores in the same execution. L2 caches are private, and when using 2 MiB pages there are no translated address bits in the L2 cache set selection that can cause random conflict behavior. Therefore variations in L2 cache hit rates are probably due to eviction of lines from the L2 by an external agent.

In previous Intel processors[12], the L3 inclusion property mandated that cache lines be evicted from all L1 and L2 caches when the line was selected to be victimized from the L3 cache. In practice, the high associativity of the L3 cache slices and the effective hashing of addresses across the L3 cache slices made such evictions extremely rare. We are not aware of any examples of L3 conflicts causing detectable performance

[12]*I.e.*, those based on Nehalem, Westmere, Sandy Bridge, Ivy Bridge, Haswell, and Broadwell cores.

degradations in these processor generations. Following the motivations discussed in [6], the Xeon Platinum processors use a non-inclusive L3 cache [2], which must be augmented by some sort of "snoop filter" to keep snoop requests at each L1 and L2 cache down to a tolerable rate. Snoop filters are analogous to sparse directories in scalable shared-memory architectures, and have a significant academic literature dating back more than 20 years [7]–[10], but there is insufficient documentation of Intel's current implementation to know whether any of the academic studies are directly relevant. As noted by [7] and [8], at high core counts, using a filter to reduce (but not eliminate) broadcast probes is not tolerable, and the only two practical choices are to invalidate lines that cannot be tracked in the snoop filter, or to allow snoop filter entries to "spill" into memory. Here Intel takes the former approach. As with inclusive L3 caches and AMD's HyperTransport Assist [11], the snoop filters in the Intel Scalable processors must track *all* lines in the L1 and L2 caches. Before a snoop filter entry can be victimized, the target cache line must be evicted from all L1 and L2 caches on the chip. As the number, size, and associativity of the L2 caches grow, it becomes increasingly difficult to distribute coherence information across the chip without creating the opportunity for associativity conflicts that can prevent the chip from exploiting its full L2 cache capacity.

As a quick "sanity check", set "DGEMM 1s B6" used 20 threads to show that reducing the number of threads (and the corresponding aggregate L2 footprint) significantly reduced the magnitude of the performance loss in the slowest runs. While the median performance decreased for these cases (as expected), each of the two sets of 651 runs had only one trial that was 10% slower than the median. The slowest run in these sets delivered 1093 GFLOPS, while 12 of the 13 subsets in "DGEMM 1s B5" (all using 24 threads) had at least one run at less than 1000 GFLOPS. Both the less frequent occurrence of slow runs and the smaller performance drops are consistent with the hypothesis that conflicts in the coherence infrastructure outside of the private L2 caches are responsible for the increased L2 miss rates and decreased performance.

C. DGEMM with CHA counters

The previous tests included performance monitoring of the cores, memory controllers, and power control units, but did not include instrumentation of the distributed caching and coherence mechanisms in the uncore. Inspired by the unexpected variation in L2 fills and the large variations in core snoop responses, the next version of the performance monitoring program was enhanced to include counters in the Caching and Home Agent (CHA) boxes. The Xeon Platinum 8160 processor has 24 active CHAs – one per physical core. The CHA performance counter interface allows access to the L3 cache counters, the mesh traffic counters, and the snoop filter.

We suspected that the variation in L2 fills was related to the snoop filters, so the CHA events were programmed to record events related to snoop filter evictions[13]. Set "DGEMM 1s B7" included core and CHA counters in all runs. A very small number of results (14 of 3193) were discarded due to interference with OS processes. As in earlier cases, approximately 1% of runs showed performance reductions of 10% to 45%. Execution time was highly correlated with DRAM CAS Reads ($r > 0.91$), while the CHA counters allowed us to compute a strong correlation with with Snoop Filter Evictions ($r > 0.82$).

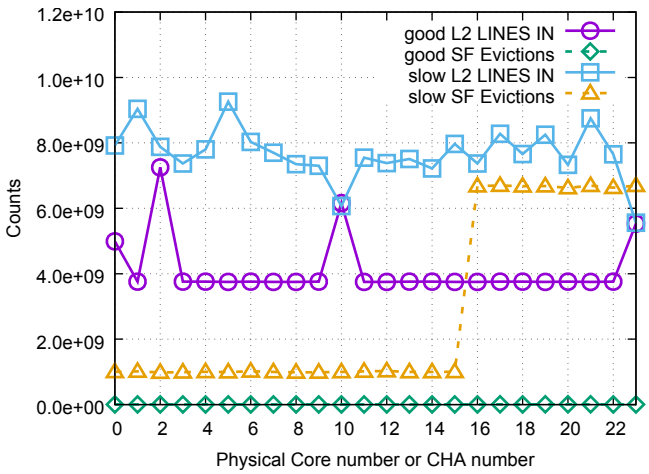

Fig. 6. Variation in L2 fills by physical core and Snoop Filter evictions by CHA for two consecutive trials on the same node. The "slow" trial was 20% slower than the (median-performing) "good" trial and required 28% more DRAM Read traffic. The snoop filter evictions in the good run were only about 0.1% of the L2 fills, so they are indistinguishable from zero on this graph.

A detailed view of the L2 fills by core and the Snoop Filter Evictions by CHA for one pair of trials is presented in Figure 6. Note that the physical cores and CHAs are numbered independently, so any correspondence in the pattern is unlikely to be meaningful. The total number of L2 fills increased by 86% in the slow run. The number of Snoop Filter Evictions increased by a factor of about 270 on the first 16 CHAs and a factor of almost 1100 on the last 8 CHAs. In the good run, the number of Snoop Filter Evictions was less than 0.1% of the L2 fills, while in the slow case, the increase in the number of Snoop Filter Evictions was more than 80% of the increase in the L2 fills. While detailed understanding of these events may not be possible given publicly available information, it seems unambiguous that there is a severe Snoop Filter conflict causing some level of "thrashing" of cache lines between the L2 and L3 caches. A fraction of those cache lines either bypass the L3 or are evicted before being fetched again, and these create long-latency DRAM accesses that are the ultimate source of the performance reductions.

For a more global view, we attempted a variety of heuristic classification schemes for the ratios of counter values in the slow runs before realizing that a simple plot would make the

[13]SF_EVICTION.M_STATE 0x0040013d, and SF_EVICTION.E_STATE 0x0040023d

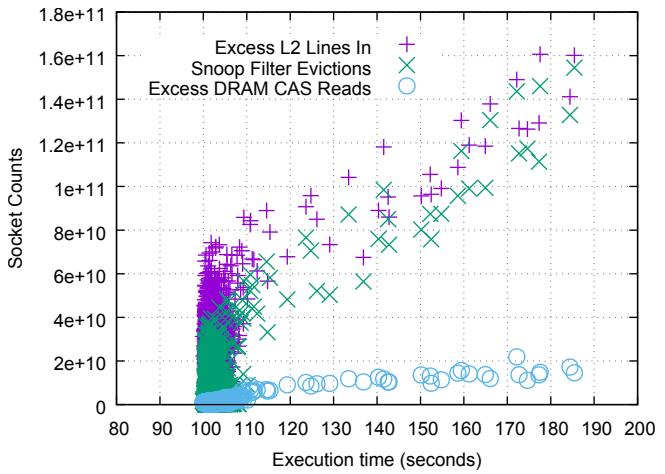

Fig. 7. Excess L2 fills, snoop filter evictions (M plus E states), and excess DRAM reads as a function of execution time for 3179 runs on the 31 test nodes. The 32 runs (1.0% of the total) with execution times more than 10% above the median show a clear pattern of increasing L2 fills, snoop filter evictions, and DRAM reads.

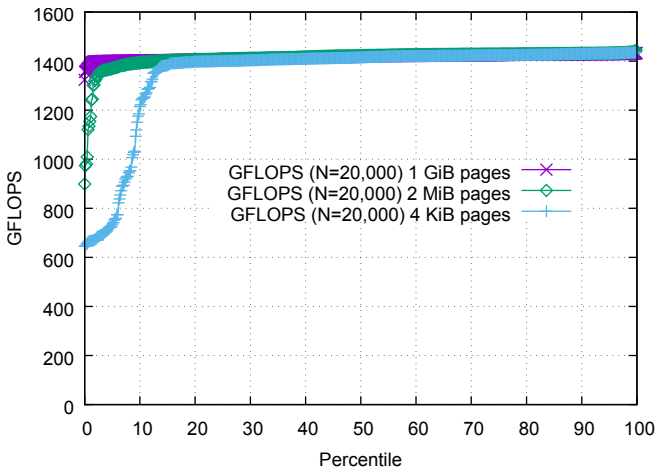

Fig. 8. Sorted single-socket DGEMM performance (N=20,000, 24 cores) for 4 KiB pages, 2 MiB pages, and 1 GiB pages.

relationships apparent. Figure 7 plots the "excess" L2 fills[14], the sum of the M and E state snoop filter evictions, and the "excess" DRAM cacheline reads as a function of the execution time for each of the 3179 runs. Only 32 of the runs have an execution time that is more than 10% slower than the median (i.e., greater than 111.5 seconds), and all of these cases show highly elevated snoop filter evictions, which lead to increased L2 fills, some of which miss in the L3 and become increased DRAM reads.

D. Mitigation: DGEMM on 1GiB pages

The variation in performance between runs appears to be due to infrequent random combinations of high-order address bits that cause the address hash to fail to distribute the active set of addresses across the CHAs with enough uniformity to avoid thrashing. We speculated that the use of 1 GiB pages would provide sufficient control over the addresses that these conflicts would either disappear or be greatly reduced. We quickly discovered that putting the three data arrays on 1 GiB pages made no difference to the performance variability[15]. The conflict must involve addresses dynamically allocated by the MKL library as well as the addresses of the main data arrays. For set "DGEMM 1s B8", we linked the executable to the "libhugetlbfs.so" library[16], causing all dynamically allocated data to be placed on 1 GiB pages. The results are outstanding, with the slowest (of 1271) run only 6.8% below the median, while the cases using 2 MiB pages always had at least one run at 25% below the median.

For the fixed problem size of N=20,000, a comparison of results on 4 KiB pages, 2 MiB pages, and 1 GiB pages is presented in Figure 8. The results with 4 KiB pages have a significantly wider and deeper "tail", as expected due to L2 cache page coloring conflicts[17].

For larger problem sizes (N=50,000), the results with 1 GiB pages are similar, with the slowest run (of 930) at only 3.2% below median performance.

V. Re-testing with HPL

The original motivation for this performance analysis was to understand and, if possible, mitigate performance drops that were seen with the HPL benchmark. Given the success of using 1 GiB pages with the single-socket DGEMM benchmark, we requested a modified version of the `xhpl` benchmark from Intel with support for 1 GiB pages. Intel provided us with such a version, with the caveat that performance would be slightly degraded when run as a single task because of the 1 GiB NUMA placement granularity inherent in the use of 1 GiB pages[18][19].

Upon receipt of the new code, we tested the performance for single-node runs using the standard 2 MiB Transparent Huge Pages used by default. Using a medium HPL test size (N=120,000, NB=384), the ensemble of 651 results displayed a median result of 2218 GFLOPS. One run was at 15% below the median (1880 GFLOPS), and a total of five runs (0.8%) were more than 10% below the median value. This confirms that the new binary displays the same frequency and amplitude of performance variability as the earlier test codes.

The new code has "native" support for 1 GiB pages, controlled by environment variables, so the LD_PRELOAD approach was not required. A large ensemble of 247 runs

[14]For L2 fills and DRAM reads, "excess" is computed by subtracting the smallest count for the corresponding event across the ensemble of 3179 runs.

[15]These tests are a subset of those in "DGEMM 1s B7".

[16]https://github.com/libhugetlbfs/libhugetlbfs

[17]Note that due to the uniform work distribution and lack of load balancing, it is not the average L2 miss rate due to random page coloring that determines the performance – the controlling factor is the worst L2 miss rate across the 24 L2 caches.

[18]Kazushige Goto, Intel, personal communication.

[19]Performance is restored when running one MPI task per socket, but our experiments were run in "SMP mode" for compatibility with the prior results.

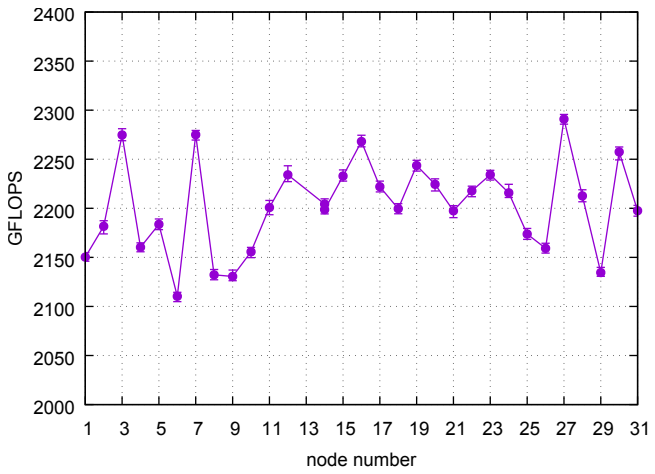

Fig. 9. Performance of the single-node HPL benchmark on the 31 test nodes using 1 GiB pages. For each node, the error bars show the minimum, average, and maximum performance seen across an ensemble of 247 runs.

were performed on each node (7657 runs in total) with the N=120,000 problem size. As we were advised, the median performance dropped very slightly to 2202 GFLOPS. Memory controller performance counter measurements confirmed a modestly uneven distribution of memory accesses that was, based on prior measurements of performance vs. memory traffic, consistent with this slight median performance reduction. The slowest run of the set was at 2105 GFLOPS, about 4.4% below the global median. We immediately noticed that the 247 slowest runs were all on a single node, which had very small run-to-run performance variability, but a low median performance. Comparing each trial with the corresponding node's median performance showed that the slowest result across the ensemble of 7657 runs was only 0.37% below the median performance for the node. This is an extremely low level of performance variability, and clearly indicates that the snoop filter conflict (which would have been expected to impact about 40 runs (one of 200)) is no longer present.

From this final set of runs, we were able to compute run-to-run and node-to-node variability for the HPL benchmark using 1 GiB pages. Figure 9 presents the maximum, median, and minimum performance values across the ensemble of 247 runs for each of the 31 nodes. The ensembles are extremely tight, with the largest standard deviations at 0.13% of the average. The full range of maximum performance values across nodes is about 8.6%, or about an order of magnitude larger than the largest range of max to min values on any of the nodes.

Finally, the reduction in performance variability provided by using 1 GiB pages was a major contributor to our ability to increase our full-system HPL performance by more than 28%[20]. As a rough estimate, we estimate that full-system performance was improved by about 5% by sorting our nodes into four "speed bins" and using the ability of the Intel HPL benchmark code to distribute work unequally across the nodes[21]. The elimination of the snoop filter conflict is estimated to have provided an improvement of approximately 15%. Although we are unable to provide quantitative estimates of how much it improved overall performance, the near complete elimination of run-to-run variability almost certainly made our other tuning efforts (e.g., $P \times Q$ decomposition, block size, MPI tuning parameters, etc.) more effective.

VI. Analysis

The preceding results provide strong evidence that a snoop filter conflict causes the infrequent performance drops in the HPL and DGEMM benchmarks, but provide little insight into why this happens, or whether the phenomenon may be expected to be relevant to other workloads. To address these questions, we reproduced the snoop filter conflict using a much simpler kernel, and analyzed the CHA/L3 address mapping hash to further elucidate the nature of the conflict.

A. Snoop Filter Conflict with an array summation kernel

It is well known that high-performance DGEMM implementations use blocked algorithms that are designed to re-use data in the caches, typically starting with the L2 cache [5], so it is not surprising that a snoop filter conflict that unpredictably evicts data from the L2 cache can cause significant performance loss. To see if this snoop filter eviction phenomenon is likely to be of general interest, we investigated the simplest kernel that addresses the issue of L2 cache data retention – repeated summation of the elements of a contiguous array. The initial array size tested was chosen to be 125,000 (8-Byte) elements per core – requiring approximately 95% of each 1 MiB L2 cache[22]. The test code reads the performance counters on all cores and CHAs, sums the array 1000 times (using an OpenMP parallel loop distributing the data elements across all 24 cores), and reads the performance counters again. In ensembles of 1000 executions of the code, the L2 miss rate varied between the expected near-zero values and values exceeding 40%. The cumulative probability distribution was approximately exponential, with about 25% of the runs showing $> 5\%$ miss rates and about 1% of the runs showing miss rates of 30% or more. In each of these cases, the number of L2 cache misses reported by the core performance counters was a near-perfect match to the number of Snoop Filter Evictions reported by the CHA performance counters, with correlation coefficient indistinguishable from unity (> 0.9999), a slope of near unity (≈ 0.999), and an intercept (number of L2 cache misses not accounted for by Snoop Filter Evictions) of well under 0.5% of the total number of loads. Execution time was approximately linear with snoop filter eviction rate, yielding slowdowns of more than $8x$ for the cases with $> 40\%$ L2 miss rates.

[20] https://www.top500.org/system/179045

[21] https://software.intel.com/en-us/mkl-linux-developer-guide-heterogeneous-support-in-the-intel-distribution-for-linpack-benchmark

[22] All tests used either 2MiB or 1GiB large pages, so contiguous addresses are guaranteed to uniformly cover the 1024 sets of the 16-way-associative 1 MiB L2 cache on this processor, allowing use of the entire cache without suffering random conflict misses.

To further explore the scope of the problem, these tests were repeated using different thread counts (still using 125,000 elements per core). Ensembles of 1000 trials for each thread count showed a minimum of 10 threads (9.54 MiB) was required to see snoop filter conflicts causing L2 miss rates in excess of 5%, with more frequent occurrences of snoop conflicts and higher worst-case miss rates seen for all larger thread counts.

Initial tests with 1 GiB pages showed no evidence of cache misses or snoop filter conflicts for contiguous address ranges of up to ≈ 24 MiB located at the beginning of any 1 GiB page. Snoop filter conflicts do occur for this simple array summation test at higher offsets within 1 GiB pages. Investigations into this phenomenon are ongoing.

B. Properties of the CHA/L3 address mapping hash

The L2 misses caused by snoop filter evictions must be due to uneven mapping of contiguous addresses across the snoop filter entries. This could be due to uneven assignment of lines to CHAs, uneven assignment of lines to the sets within CHAs, or both. Simple testing with the hardware performance counters (similar to [12], but derived independently) showed that every aligned set of 512 contiguous cache lines maps 21 cacheline addresses to each of CHAs 0-15 and 22 cacheline addresses to each of CHAs 16-23. This is consistent with the pattern observed in [13], but extended from 128 cacheline sequences (on a processor with 6 L3 slices) to 512 cacheline sequences (on this processor with 24 L3 slices). Since only contiguous addresses are being used in these simple contiguous summation tests, this near-uniformity ensures that the conflict cannot involve a "bulk" asymmetry of allocations to the various CHAs, and must therefore involve a conflict in set selection within the CHAs. Analysis is ongoing, but it is not yet clear whether it will be possible to invert the set selection equations in sufficient detail to analytically derive snoop filter conflicts from arbitrary physical addresses.

VII. Summary

An infrequent, but significant, performance drop was observed when running the HPL benchmark on clusters of dual-socket servers equipped with Xeon Platinum 8160 processors. We were able to reproduce the frequency and magnitude of the performance drop using a simpler single-socket DGEMM benchmark. Extensive studies using hardware performance counters eventually enabled us to identify the mechanism underlying the performance loss – unfortunate combinations of physical addresses causing a conflict in the Snoop Filters which causes data to be evicted from the L2 caches while it is still in use. At low eviction rates, the majority of these evicted cache lines are held in the L3 for re-use, and the overhead of re-loading them from the L3 to the L2 caches does not materially impact performance. As the eviction rate increases, an increasing fraction of the evicted lines are not found in the L3, and the added latency of retrieving the data from DRAM memory begins to add to the execution time.

When using 2 MiB large pages, a significant ($> 10\%$) performance drop occurs on roughly one out of every 200 sockets. HPL benchmark runs using more than 100 nodes quickly run into a situation in which several nodes are running slowly in every trial, resulting in load imbalance and reduction in the overall system performance.

For this benchmark, the use of 1 GiB large pages reduces both the frequency and magnitude of the performance variability to negligible levels. This, in turn, provided a significant fraction of our full-system HPL performance increase of more than 28%. Unfortunately, this is not a practical approach for production use, as the use of 1 GiB pages often requires application modification, and changing the number of reserved 1 GiB pages (either up or down) requires a system reboot.

Further testing and analysis has shown that the snoop filter conflict uncovered in the analysis of HPL and DGEMM also occurs (with varying frequency and intensity) with any attempt to hold contiguous blocks of 10 MiB or larger for re-use in the aggregate L2 cache of the Xeon Platinum 8160 processor. This phenomenon occurs when using 2 MiB or 1 GiB pages. With 1 GiB pages, the eviction rate is predictable, but for 2 MiB pages we are currently unable to predict what combinations of high-order bits will induce high snoop filter eviction rates.

VIII. Concluding Remarks

With the benefit of several months of review (and the constraint of page limits on papers), it is difficult to avoid describing the path of this investigation in a linear, streamlined way. This is, of course, not how such investigations happen. Performance anomalies that occur on the order of once every one hundred runs could be due to a bewildering variety of factors, many of which are never satisfactorily explained. In this case, we were "helped" by the resilience of the problem – despite numerous changes in software and runtime configuration, the snoop filter conflict remained an infrequent, but unwelcome visitor.

We have not seen evidence of this performance variability impacting other applications on our clusters, but at a frequency of one out of two hundred nodes, it is entirely possible that it has occurred and been ignored. Moving forward, we plan to collect CHA performance counter data on snoop filter evictions as part of our routine system monitoring.

If there is a moral to this story, it is that with increasingly complex processors, assumptions about behavior that have always worked in the past may need review. In this case, we have shown that even though 2 MiB pages ensure freedom from conflicts within a single L2 cache, they do not ensure freedom from conflicts when using many (nominally independent) L2 caches concurrently. Sites with Xeon Scalable processors with different core counts may be interested in repeating this analysis on their systems.

Acknowledgment

This work was funded by the National Science Foundation, award number 1663578.

APPENDIX A
ARTIFACT DESCRIPTION: HPL AND DGEMM PERFORMANCE VARIABILITY ON THE XEON PLATINUM 8160 PROCESSOR

A. Abstract

This appendix describes the test environment and methodology used in the characterization and analysis of HPL and DGEMM variability, and to demonstrate the existence of snoop filter conflicts on a simple contiguous array summation kernel on systems using the Intel Xeon Platinum 8160 processor.

B. Description

1) Check-list (artifact meta information):

- **Algorithms:** dense linear system solver using LU factorization, dense matrix multiplication, parallel array summation
- **Programs:** binary executables (distributed by Intel), serial and OpenMP C programs, launching and post-processing scripts
- **Compilation:** Intel 17.0.4 C/C++ compiler (20170411), Intel 18.0.0 C/C++ compiler (20170811)
- **Binary:** xhpl binaries provided by Intel
- **Run-time environment:** CentOS 7.3 (kernel 3.10.0-513), CentOS 7.4 (kernel 3.10.0-693)
- **Hardware:** Any server configured with at least one Intel Xeon Scalable processor with 24 L3 slices enabled (Xeon Platinum 8160 or Xeon Platinum 8168)
- **Run-time state:** Multi-user mode (runlevel=3), with Transparent Huge Pages enabled
- **Execution:** Memory and thread binding controls in scripts
- **Output:** output files produced by workload under test, performance counter output files (time series)
- **Experiment workflow:** download and install any or all of the three test codes, modify the run scripts to execute a suitable large ensemble of results (recommended to be many hundreds of trials), sort performance results to identify performance outliers, review performance counter results for the periods during which the slow trials were executing
- **Publicly available?:** Partial (see notes below)

2) How software can be obtained (if available): The Intel xhpl binaries used were provided to us directly by Intel. These do not include version numbers, but they do print a build date in the output files.

- The binary used for the results discussed in Section III prints "`built on Oct 11 2017 at 20:00:00.`"
- The binary used for the single-node results discussed in Section V prints "`built on Dec 7 2017 at 12:09:52.`"
- The binary used for the multi-node results discussed in Section V prints "`built on Mar 20 2018 at 07:36:38.`"

The latter two versions include 1 GiB page support. We do not know if this feature is supported in publicly available versions of the code, or whether it will be supported in future versions.

The source code and scripts to reproduce the remainder of the results of this work can be obtained from three public repositories:

- `https://github.com/jdmccalpin/simple-MKL-DGEMM-test` contains a simple driver and scripts for running the DGEMM benchmark code.
- `https://github.com/jdmccalpin/periodic-performance-counters` contains the source code for the program that runs in the background, collecting hardware performance counter data periodically while the code under test is executing.
- `https://github.com/jdmccalpin/SKX-SF-Conflicts` contains the code to repeatedly sum a (nominally) L2-containable array, with built-in interfaces to the hardware performance monitors. This code also has the ability to use the performance counters to determine the mapping of physical addresses to L3 slices (as discussed in section VI.B.).

3) Hardware dependencies: The tests were run on two clusters of dual-socket servers equipped with Xeon Platinum 8160 processors. These are 24-core processors with a nominal frequency of 2.1 GHz, a maximum all-core Turbo frequency of 2.0 GHz when running AVX512 code, and a guaranteed AVX512 frequency of 1.4 GHz. HyperThreading is enabled in the BIOS, but all tests reported here used only one Logical Processor per physical core. The clusters are from different vendors, but are configured with comparable interconnect and DRAM (one dual-rank 16 GiB DDR4/2667 DIMM per channel, for a total of 192 GiB/node).

The phenomenon investigated in this work was demonstrated on many nodes in each of these clusters. There is no evidence that any nodes are "immune" to the phenomenon. In particular, the slowdowns were observed on all 31 nodes from "vendor B", which included 21 different patterns of disabled L3 slices in the CAPID6 register. The phenomenon can be reproduced with a single socket of a two-socket system, and (based on the underlying mechanism) should be independent of the number of sockets populated.

4) Software dependencies: The `xhpl` tests are dependent on binary software distributions from Intel.

The codes from the `simple-MKL-DGEMM-test` and `SKX-SF-Conflicts` require a version of the Intel 2017 or 2018 C/C++ compiler, including the MKL library (which contains the high-performance DGEMM implementation). The code from the `periodic-performance-counters` project requires only a C compiler with OpenMP support (we used the Intel 2018 C/C++ compiler).

Neither the `xhpl` or the DGEMM benchmarks include an infrastructure for reading performance counters, but the DGEMM project includes an example script showing how to use the binary from the `periodic-performance-counters` project to obtain concurrent performance counter measurements. This script can easily be adapted to work with the `xhpl` benchmark codes, if desired.

5) Datasets: No input data is required.

C. Installation

1) SKX-SF-Conflicts: The project must first be cloned to the local machine, using

```
$ git clone https://github.com/\
jdmccalpin/SKX-SF-Conflicts
```

The README.md file in the project contains a description of the two versions of the code: SnoopFilterMapper.c (which is specialized for use with 2 MiB pages), and SF_Test_Offsets.c (which is specialized for use with 1 GiB pages). The README.md file contains an outline of the structure of the code, with extensive porting notes and a description of the expected run-time environment configuration.

2) perf_counters: The perf_counters infrastructure must first be cloned to the local machine, using

```
$ git clone https://github.com/jdmccalpin/
periodic-performance-counters
```

The file "READ.ME" contains a list of configuration steps that must be reviewed for machine-specific or site-specific issues. The code and/or Makefile will need to be updated accordingly.

This code requires root privileges to access the /dev/mem and /dev/cpu/*/msr device drivers. It can either be run as root, or (if OS security policies do not block this) tagged as a setuid root binary.

3) DGEMM: The DGEMM driver code must first be cloned to the local machine, using

```
$ git clone https://github.com/\
jdmccalpin/simple-MKL-DGEMM-test
```

The default target for make builds four binaries from the simple_MKL_DGEMM_test.c source code, differing only in the method used to allocate the three arrays used by the DGEMM routine. The script run_ensemble.sh is set up to run the DGEMM benchmark 200 times, saving the output in 200 separate log files. Each execution of the benchmark code calls the DGEMM routine 12 times, reporting both per-iteration measures and cross-iteration statistics (excluding the first call, which is normally slow due to MKL library initialization overhead).

The script run_with_perf_counters.sh shows how to integrate the ensemble testing with background performance measurements using the perf_counters executable from the periodic-performance-counters project. In this example, the performance counters run in the background during the entire ensemble of runs, so performance counter data from a specific run requires looking up the starting and ending TSC values (provided in the DGEMM output log files) with the TSC values in the performance counter output file.

4) xhpl: After obtaining the xhpl distribution(s) from Intel, follow the instructions provided by Intel for configuring the input data files. Asymptotic performance is typically obtained using problem sizes of 100,000 or larger. Due to the larger problem sizes required to obtain asymptotic performance, these are much slower than the DGEMM runs. The run_with_perf_counters.sh script from the simple_MKL_DGEMM_test project can be easily adapted to run the xhpl binaries with background performance counters. In this case it is recommended that the NUMTRIALS variable in the script be set to 1, so performance counter data is collected for a single execution of the xhpl benchmark.

D. Experiment workflow

The easiest way to demonstrate the snoop filter conflict is with the code from the SKX-SF-Conflicts project. Once configured and built, the script run_ensemble.sh will run the SnoopFilterMapper executable 100 times. The default array size should be L2-containable, so the L2 miss rate should be very close to zero. On a Xeon Platinum 8160 using 2 MiB pages, an ensemble of 100 trials will contain a handful with L2 miss rates in excess of 20%, and with snoop filter eviction counts that are effectively perfectly correlated with the L2 cache miss counts. With the default performance counter settings, the log files will contain the snoop filter eviction count (summed across all CHAs on socket 0) in the output line starting with CHA_PKG_SUMS pkg 0 counter 0, while L2 cache misses are collected by core performance counter 2, and the sums can be found in the output file in lines starting with CORE_PKG_SUMS pkg 0 counter 2.

The DGEMM and HPL tests take much longer to run and require more labor-intensive post-processing, but the workflow is similar: run an ensemble of jobs and look for correlations between snoop filter eviction counts, L2 fill counts, and execution time.

The original tests were launched to the various nodes using system-specific (unportable) scripts. Fortunately, any method of launching jobs will suffice, as long as the nodes being tested are not shared. Typically an experiment consists of running the code under test multiple times (within a single batch job) on many nodes concurrently. Once these jobs have completed, slow runs are identified using simple grep, awk, and sort commands on the output files. For the DGEMM tests, the portion of the performance counter output corresponding to the slow execution is identified using the starting TSC times in the DGEMM output file, and these starting and ending times are used to find the start and end sample numbers in the performance counter output files. The lua script post_process.lua requires the first command-line argument to be the performance counter output file name, but optional second and third arguments provide the starting sample number (defaults to the first sample) and ending sample number (defaults to the final sample) for the analysis.

E. Evaluation and expected result

All of the three test codes should show a wide range of snoop filter eviction rates, though with different performance sensitivities. For DGEMM and HPL, roughly 1% of runs are expected to be 15% slow due to snoop filter evictions. With the SnoopFilterMapper code, roughly 1% of runs are expected to show L2 miss rates (due to snoop filter conflicts) of greater than 30%.

F. Experiment customization

Numerous possibilities for customization are implied in the discussions above.

G. Notes

None.

REFERENCES

[1] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel, "Cache coherence protocol and memory performance of the Intel Haswell-EP architecture," in *2015 44th International Conference on Parallel Processing*, Sept 2015, pp. 739–748.

[2] A. Kumar. (2017, August) The new Intel Xeon Scalable Processor (formerly Skylake-SP). Intel Corporation. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.22-Tuesday-Pub/HC29.22.90-Server-Pub/HC29.22.930-Xeon-Skylake-sp-Kumar-Intel.pdf

[3] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, Intel Corporation, December 2017, document 325384-065US.

[4] *Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual*, Intel Corporation, July 2017, document 336274-001US.

[5] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1356052.1356053

[6] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer, "High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 343–353.

[7] D. E. Lenoski and W.-D. Weber, *Scalable Shared-Memory Multiprocessing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994.

[8] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel, "WAYPOINT: Scaling coherence to thousand-core architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854291

[9] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling ways and associativity," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2010, pp. 187–198.

[10] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 169–180.

[11] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache hierarchy and memory subsystem of the AMD Opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, March 2010.

[12] C. Maurice, N. Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, ser. RAID 2015. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 48–65. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26362-5_3

[13] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser. (2015) Mapping the Intel last-level cache. IACR Cryptology ePrint Archive. [Online]. Available: https://eprint.iacr.org/2015/905.pdf