Asymmetries in Multi-Core Systems – Or Why We Need Better Performance Measurement Units

Irina Tuduce[2], Zoltan Majo[1], Adrian Gauch[1], Brad Chen[2] and Thomas R. Gross[1]
[1]Institute of Computer Systems, ETH Zurich, 8092 Zurich, Switzerland
[2]Google Inc., Mountain View, CA

Abstract

Future exascale systems will be based on multi-core processors, but even today's multi-core processors can be asymmetric and exhibit limitations and bottlenecks that are different from those found on a symmetric multiprocessor. In this paper we investigate the performance of a cluster node based on the Intel Xeon E5345 quad-core processor and note that despite the symmetry implied by the programming model, the available memory bandwidth is not shared equally among the cores. Consequently, applications experience substantial performance variance and slow-downs when the tasks (threads) are mapped to cores in a naive manner. An operating system scheduler could mitigate these effects by taking into account the memory bus structure but needs accurate information from the performance monitoring unit as the asymmetry is not directly exposed in the processor's instruction set manual. Current performance monitoring units are quite inflexible and change from one processor to the next, so higher levels of the software tool chain are discouraged to use them. The next generation of Nehalem-based multi-core systems poses similar challenges, and the development of portable performance monitoring units will be crucial if applications want to use the performance potential of exascale systems. We expect this situation to remain unchanged as long as memory is slow relative to the processor.

1 Introduction

Building an exascale system (a supercomputer delivering one or more exaflops) requires addressing a number of challenges. But the biggest challenge is to allow applications to realize the performance potential of such systems. As the advances in memory system design lag behind increases in processor performance, it is likely that in future supercomputers managing and using the memory bandwidth will be even more important than in today's petascale systems.

The November 2009 edition of the TOP500 list shows that more than 80% of the supercomputers with the highest performance are based on a cluster-like architecture, where each cluster node is essentially a multiprocessor, built with commodity processors designed for the server market. Currently about 88% of the processors in use are based on the 64-bit extension of the Intel x86 instruction set, and this instruction set architecture has been steadily increasing its share of the machines on the TOP500 list for the last ten years. It is very likely that the trend will continue and future generation supercomputers will be using commodity microprocessors as well. Therefore, if we want to exploit the full performance potential of these future machines, we must learn to use a single cluster node efficiently.

Multi-core systems, also known as chip multiprocessors (CMPs), have become the dominant computing platform. Today's desktops and laptops are commonly built using dual-core processors, and quad-core processors are common in server-class systems, sometimes with multiple sockets leading to eight or sixteen-way multiprocessors. It can be assumed that future exascale systems will be based on multi-core designs as well.

Although the computer hardware industry has embraced the multi-core paradigm, with systems in development that realize much higher core counts, there are substantial unsolved problems on the software side. On the outside, a multi-core processor is a binary compatible replacement of its "single-core" predecessor. This property is clearly attractive since there is a large software base that can be executed on such a multi-core processor. But multi-core and many-core systems (the likely platform for exascale systems) may deliberately sacrifice serial performance for increased parallelism. Such systems will require an entirely different programming model to effectively utilize their full potential performance.

But even today's modest multi-core systems with two or four cores harbor surprises and exhibit unexpected performance variations. Consider a system built with two quad-core processors as shown in Figure 1. There are

eight cores, each with its own L1 cache, and two cores share an L2 cache. At first sight, such an 8-core system resembles a shared-memory symmetric multiprocessor (SMP) with eight processors, and the considerable success (both academic and commercial) of SMP systems suggests that such a system will be a ready-to-use platform for many applications. This paper documents a counter-intuitive behavior of these systems, and we briefly discuss how a prototype CPU scheduler can partially addresses the negative impact of the asymmetric design.

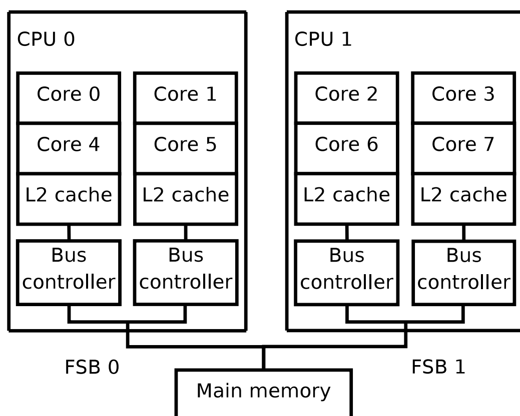

Figure 1: System architecture with two Xeon 5300 series CPUs. Cores numbered according to the convention used by the Linux kernel.

We have investigated a system like the one depicted in Figure 1 and constructed a memory bandwidth performance model for this system. Each processor (CPU) is an Intel Xeon 5345 quad-core processor and is connected via a separate memory bus ("Front Side Bus" (FSB) using Intel's terminology) to main memory. As we explored the effective memory bandwidth available to each core, we noticed that the mapping of tasks to cores has significant impact on performance. Given four identical tasks, the slowest task can take up to four times as long as the fastest task. A mapping of tasks to cores that does not take the system architecture into account may either leave (memory) resources idle or create avoidable bottlenecks. This motivated our development of a scheduler that take the realities of the multi-core system into account.

2 Asymmetry in practice

To assess the effects of possible contention in the system shown in Figure 1, we focus on memory bandwidth because memory bandwidth is a key performance issue for many applications.

2.1 Setup

Our experimental system has two Xeon 5300 series CPUs, each with four cores. Each core has a private 64 KB unified L1 cache; pairs of cores share a unified 4 MB L2 cache. The cores operate at 2.33 GHz. The caches are kept coherent. Each processor has its own memory bus and connects two groups of two cores via a separate bus controller to the memory bus. The main memory size is 16 GB so that we can run benchmark programs with substantial memory requirements.

A Xeon 5345 quad-core CPU consists actually of two dual-core processors that are joined together in a multi-die package. The existence of two bus controllers is probably due to the desire or need to reuse an existing dual-core design. But since multi-core designs have been criticized for lack of suitable memory systems[1] this decision to include multiple bus controllers may actually provide a performance advantage over single-die quad-core designs that force 4 cores to share a bus controller. We have also investigated a system based on the Xeon E5520 (the Nehalem-based single chip quad-core), which uses a different bus architecture and a different memory system, and discuss its asymmetries briefly. Although the detailed memory system performance is different for these two systems, the results obtained on both systems support our plea for better (and uniform!) performance monitoring units.

We use Linux with kernel version 2.6.23 for our experiments, together with *perfmon2* version 3.2. The next section summarizes our use of the performance monitoring unit (PMU); this section is included to allow the reviewer to double-check our approach. We just note that we validated our tools using a suite of synthetic applications with known memory bandwidth demands.

2.1.1 Performance monitoring

We use the hardware PMU to measure the memory-bandwidth demands of applications running on the Intel system. Our measurement methodology is similar to that described by Eranian [3]. Each core has five counters that can be programmed to measure the occurrence of events (with some restrictions). Three of these counters are dedicated to measuring three specific events. The remaining two counters can measure any of the many Intel performance events. We set one of the dedicated counters to measure the number of core cycles when the core is not halted, *CPU_CLK_UNHALTED.CORE*, and one of the general-purpose counters to measure the number of burst (full cache-line) bus transactions, *BUS_TRANS_BURST*. We set the Unit Mask flag to measure only the transactions of the core on which the application is running. The

[1] See the blog `http://abinstein.blogspot.com/2007/-08/not-everything-about-memory-is.html` and its links as examples to illustrate popular interest in this matter.

Xeon 5300 system has a cache line size of 64 bytes, so each transaction transfers this amount. To compute the memory-bus bandwidth achieved by a program we use the following formula:

$$bandwidth = \frac{64 \times bus_transactions \times core_frequency}{elapsed_cycles}$$

2.2 Memory bandwidth micro-benchmark

We used a synthetic micro-benchmark similar to the *triad* workload of the STREAM benchmark[11] suite for our controlled experiments. The core of the benchmark is an infinite loop that reads and writes three arrays `a`, `b`, and `c` (see Figure 2). The size of these arrays is chosen such that they do not fit into the L2 cache and thus cause memory bus transfers.

```
while (1) {
  for (j = 0; j < ARRAY_SIZE; j++) {
    a[j] = b[j] + SCALAR * c[j];
  }
}
```

Figure 2: Synthetic workload.

First, we measure the effective memory bandwidth by instantiating this benchmark task on 1, 2, 3, 4, 5, 6, and 8 cores; Section 2.3 reports these results. We start the benchmark on the appropriate cores (using `sched_affinity()` to control the placement of tasks) and then measure for a period of three seconds the memory bandwidth on these cores, using the hardware PMUs of the system. All numbers reported are the averages of multiple runs (there was little variation among runs and therefore the variance is not shown in the figures to avoid clutter). As our synthetic workload is extremely memory intensive, the achieved performance of the workload tightly correlates with the achieved bandwidth.

Then in Section 2.4 we report results when the cores execute tasks with different bandwidth demands. We hand-optimize the loop body shown above (by adding additional arithmetic operations to the assignment statement) to cut the memory demands in half so that we can combine high-bandwidth jobs (original loop) with low-bandwidth jobs (modified loop).

2.3 Identical tasks

When one instance of the benchmark task is executed on the different cores of the system, each core obtains between 2.88 and 2.92 GB/s, close to the theoretical maximum of the bus (3 GB/s).

If two tasks execute on the system, the mapping of the tasks to cores significantly influences the achievable bandwidth, as shown in Figure 3. If the two tasks are mapped to different processors (and therefore do not share last-level caches, bus controllers, or FSBs) then each task obtains more than 90% of the single-task bandwidth. However, if tasks are mapped to the same CPU, the bandwidth of the FSB limits the achievable bandwidth, and each task obtains half of the single-task memory bandwidth. If the tasks are mapped to cores that share not just the FSB, but also the bus controller and the last-level cache (recall from Figure 1 that there is one bus controller for each L2 cache), then the synchronization overhead is reduced. In this case the tasks realize slightly higher bandwidth as if different bus controllers were used.

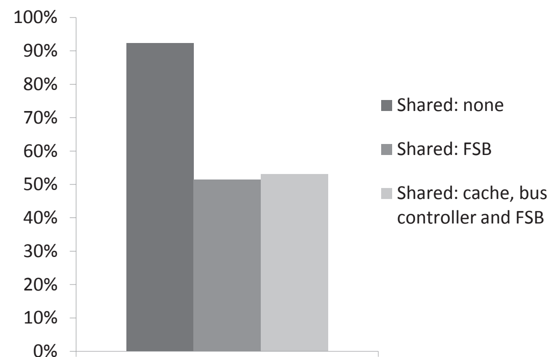

Figure 3: Relative bandwidth for single task in a two-task configuration.

There are three ways to schedule three tasks on this system; each schedule results in tasks using the memory system in a different configuration: (1) all three tasks on the same processor, (2) two tasks on one processor, one task on the other processor, with the tasks on the first processor sharing the bus controller and the last-level cache, and (3) two tasks on one processor, one task on the other processor, with the memory bus being the only resource shared between tasks on the first processor. Figure 4 shows the bandwidth obtained by the individual tasks for these schedules relative to the single-task bandwidth. These measurements confirm the model of bandwidth sharing: if memory requests are issued on both buses (Schedules 1 and 2), the cores behind each bus obtain half of the total bandwidth. In all cases, the bandwidth obtained by a single bus is equally divided between the two bus controllers of the processor. If only a single memory bus is used (Schedule 3), then the bandwidth, limited by the FSB's transfer rate, is shared between the two bus controllers of the processor. The overhead of bus arbitration (and possibly synchronization) is also visible: the overall bandwidth obtained by Schedule 2 is slightly more than

that obtained by Schedule 1, because in case of Schedule 2 two of the tasks share the last-level cache and the bus controller. Likewise, in the case of Schedule 3, the cores connected to FSB 0 obtain more bandwidth because only one bus is used, therefore bus arbitration is not necessary.

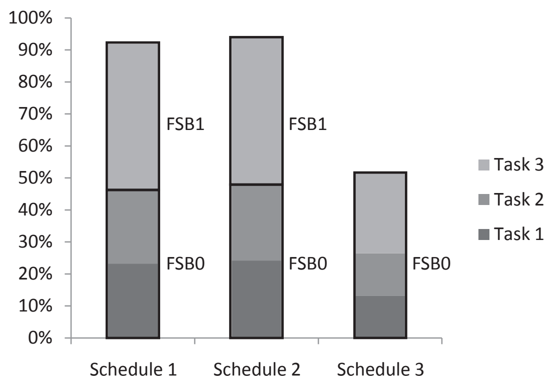

Figure 4: Relative bandwidth for three tasks.

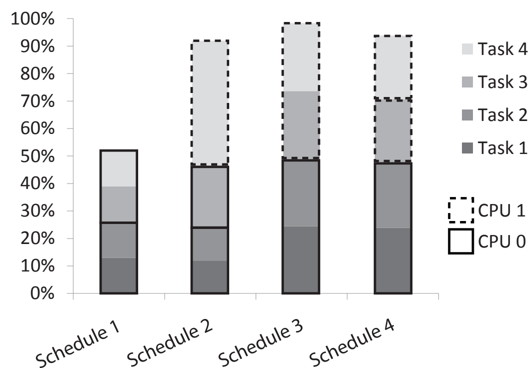

Figure 5: Relative bandwidth of four tasks. The relative bandwidth of memory controllers connected to FSB 1 is shown in dashed boxes; the relative bandwidth of FSB 0 memory controllers is shown in solid boxes.

The relative bandwidth of four tasks in different scheduling scenarios is shown in Figure 5. The measurement data further justify our claim that bandwidth is shared fairly well with regard to the FSBs/bus controllers/cores, but it is *not* shared fairly with regard to cores alone, as one would expect in an SMP. In Figure 5, the relative memory bandwidth obtained by each bus controller in indicated by a box with bold lines. The boxes of CPU 1 use dashed lines; the boxes for CPU 0's memory controllers are shown with solid lines. Bus arbitration between two processor-local bus controllers is clearly noticeable: Schedule 4 achieves less bandwidth than Schedule 3 because Schedule 4 places two tasks so that they are serviced by two different bus controllers on CPU 1.

The results for combinations of a larger number of tasks are omitted; there are no further surprises and the measurement data confirm the earlier observations. We used a synthetic benchmark to obtain the effective bandwidth for the different cores. Real programs will access a fixed amount of data, so a reduction in bandwidth translates immediately into an extended execution time. For example, if two tasks that would consume 3.0 GB/s in an unloaded system are mapped to Core 0 and Core 1, then the execution time is almost twice as long compared to a mapping of these tasks onto Core 0 and Core 2.

2.4 Mix of high- and low-bandwidth tasks

In a real system, not all tasks will demand the maximum bandwidth. To investigate how the system behaves if the tasks have different bandwidth requirements, we produced a variation of the benchmark program that demands approximately one half of the bandwidth. When run on Core 0, this task consumes 1445 MB/s (on different cores we obtain slight variations but these data are omitted to save space). Figure 6 presents the relative bandwidth achieved by the two tasks with different memory requirements. We have examined three different schedules which result in different usage of the memory system: (1) all possible resources are shared, (2) just the FSB is shared, and (3) not shared resources between the two tasks. We see that the low-bandwidth task is favored: if a high-bandwidth task and a low-bandwidth task are mapped to different processors, the low-bandwidth task manages to obtain all the memory bandwidth that it requires (1455 MB/s), whereas the high-bandwidth task loses about 7%. If a high-bandwidth task and a low-bandwidth task reside on the same processor, then the degradation of the memory intensive task is much more dramatic: 39% relative to 12% in the case of the low-bandwidth task. If such a mix of tasks shares a bus controller, the high-bandwidth tasks is able to benefit from the reduced arbitration overhead and increases its share of the bandwidth to 67%, thus experiencing less degradation. The degradation of the low-bandwidth task decreases somewhat, as its share of the aggregate bandwidth drops with 2%.

These results (and additional measurements for larger number of tasks which have been omitted for space reasons) show that a performance model of this system must include the bus controllers to obtain the bandwidth share of a core. A naive mapping of tasks is likely to underutilize the resources of this system, and consequently slow down the execution of some tasks. In a setting with tasks that have different or changing memory bandwidth requirements, the situation can be even more complex, and it is not clear that a programmer can be expected to take the performance model into account. A solution must

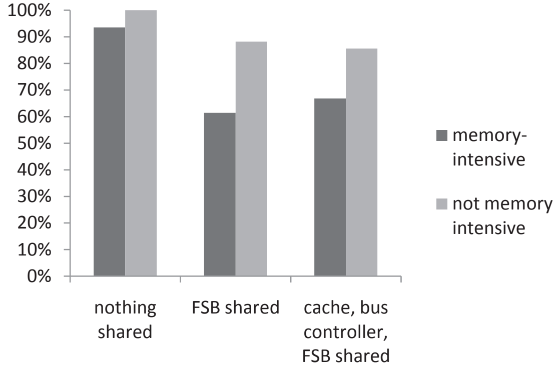

Figure 6: Relative bandwidth of tasks with different memory requirements.

therefore be provided by the system software, i.e. either the operating system or a suitable library, that must be able to classify tasks according to their memory bandwidth requirements.

3 Exploiting asymmetry

We use the SPEC CPU2006 benchmark suite for our evaluations. Our experiments show that 24 benchmarks have low-bandwidth demands (0–1.5 GB/s) and five have high demands (1.5–3 GB/s). The *gcc* benchmark triggers unexpected behavior of the PMU, as the peak bandwidth value we measured was 5.0 GB/s. Because this is the only benchmark that showed this behavior, we exclude it from the measurements (more precisely, from the set of benchmarks with high demands).

To investigate the effect of bus sharing on the application's performance, we divide our experiments in two sets. In the first set we measure the slowdown of applications with high-bandwidth demands when they share the memory bus with other applications. In the second set, we measure the slowdown of applications with low requirements.

To investigate the effect of sharing the L2 cache on the application's performance, we divide each experiment in two parts. In the first part we map one application to Core 0 and another application to Core 1. This experiment captures the effect of an unshared L2 cache (but shared FSB) on an application's performance. In the second part of the experiments we run the applications on cores that share the L2 cache (and hence the bus controller), namely on Core 0 and Core 4. This setup captures the effects of the L2 cache contention on application's performance.

In the first set of experiments we measure the performance of high-demand benchmarks. We measure the performance of each high-demand benchmark when it shares the FSB with another high-bandwidth benchmark. The results, summarized in Table 1, show that for this setup the high-bandwidth applications are slowed down by about 40%.

Core0	*Core1*				**Average**
	mcf	milc	soplex	lbm	
mcf	23.52	47.16	24.79	43.16	34.65
milc	18.62	60.96	23.96	39.65	35.8
soplex	21.5	54.07	33.92	41.09	37.64
lbm	35.62	73.65	19.53	81.04	52.46
all					**40.14**

Table 1: Average slowdowns [%] - Shared FSB.

We repeat the measurements with core assignments that share the L2 cache. In Table 2 we present the slowdown of high-bandwidth applications when they share the L2 cache. The effect of the cache contention is visible: the benchmarks are slowed down by about 43% compared to 40% when they do not share the L2 cache.

Core0	*Core4*				**Average**
	mcf	milc	soplex	lbm	
mcf	30.11	53.76	31.77	57.87	43.38
milc	9.41	39.7	27.92	25.94	25.74
soplex	26.16	76.19	35.16	43.96	45.37
lbm	43.82	73.2	21.48	97.53	59.01
all					**43.37**

Table 2: Average slowdowns [%] - Shared L2 cache.

To capture the interaction between different types of applications, we measure the slowdown of high-bandwidth applications when they share the memory bus with low-bandwidth applications. On average, the applications run 3.78% slower with separate L2 caches and 7.09% slower using the same L2 cache.

To sum up, the first set of experiments shows that, on average, applications with high-bandwidth demands are slowed down by 40.14–43.37% when they share the bus with other high-bandwidth applications. When the bus is shared with low-bandwidth applications, the slowdown is 3.78–7.09% on average.

In the second set of experiments we repeat the measurements for applications with low-bandwidth demands. The results show that low-bandwidth applications are slowed down by 16.78–28.39% when they share the bus with high-bandwidth applications, and by 1.86–5.26% when they share the bus with other low-bandwidth applications. The results confirm the observation that sharing the L2 cache decreases the application's performance. Running low- and high-demand applications on cores that share the

L2 cache results in a 28.39% slowdown on average, compared with 16.78% when they run using different caches. For the low- and low-demand applications the slowdown is 5.26% with shared caches and 1.86% using different caches.

These experiments quantify the potential performance improvements of mapping the SPEC benchmarks to cores according to their memory-bus demands and taking the system architecture into account. The results show that the performance of high-bandwidth benchmarks can be improved by 3% to 40.14% and the performance of low-bandwidth benchmarks by 1% to 28%.

3.1 Memory-bus aware scheduling

We now describe a scheduler that attempts to improve the throughput of applications run on asymmetric multi-core systems over a naive mapping that does not take the memory bus structure into account. The scheduler aims at ensuring enough bandwidth to applications with high bus-bandwidth demands (and hence attempts to improve their runtime). The use of memory-bus bandwidth is maximized by scheduling applications with low bus-bandwidth demands simultaneously with application with high demands. The scheduler strives to be fair, to avoid starvation, and not to break CPU affinity.

To cope with the asymmetric behavior of the memory bus, applications are characterized as either low-bandwidth or high-bandwidth applications. To ensure fairness, the scheduler organizes the applications as a list. Additionally, each application in the list is tagged with its memory-bus consumption characteristic (low or high). Applications at the beginning of the list are scheduled first, and then added back at the end of the list. The scheduler uses knowledge about the architecture of the system: how many cores are available, and which cores share the same L2 caches and/or FSB.

The scheduling algorithm requires information about the memory-bus demands of each application. To measure the bandwidth demands of an application we sample the PMU as described in Section 2.1.1.

When several applications are scheduled on the same processor, the memory bandwidth measurements show the current bandwidth usage and not the demands of the individual applications. To measure the bandwidth requirements of a single application, we schedule that application alone on a CPU for a scheduling cycle. Since applications might run through several phases with different bandwidth demands, we must monitor the bandwidth demand changes. To monitor these changes we look at the percentage of load/store instructions that generate bus traffic. For one phase of the application, the percentage of instructions that generate bus traffic remains the same whether the application shares the memory bus or not.

The scheduler considers the memory-bus demand changed when a demand variation of more than 4% is detected. This threshold is a tradeoff between filtering out short bursts and responding to application changes in a timely manner. A large value filters out burst with small durations, but at the same time it reduces the responsiveness to true changes in the bus-bandwidth demands. This threshold of 4% was obtained empirically for the SPEC suite; this value is the smallest value that still captures phase changes. When the application's memory-bus demands change, the scheduler marks the application for demand measurement. In the next scheduling cycle, the application is run alone on the CPU and its bus demands are measured and updated. When an application is scheduled for the first time, its memory demands are set to a default value of zero. For the majority of the applications this default value is unrealistic. Hence, a bandwidth measurement is triggered in the next scheduling cycle.

3.2 Implementation

We have prototyped our memory-bus aware scheduler for Linux entirely in user space. The scheduling decisions are made by a high-priority control process. To control the execution of the tasks (i.e., to start, suspend, resume tasks), we use `ptrace()` Linux system call. To pin tasks to specific cores, we use `sched_setaffinity()` Linux system call. The scheduler uses the *libpfm* library of the *perfmon2* monitoring interface to access the PMU and `SIGSTOP/SIGCONT POSIX` signals to perform the context switch. The scheduler uses a scheduling interval of 200 msec. Given context switch overhead of 2–5 μsec this limits our scheduling overhead to less than 0.1 %.

3.3 Results

We run all experiments on the system depicted in Figure 1. For both performance as well as fairness, we measure the time to complete the execution of a workload under the default Linux scheduler and under the memory-bus aware scheduler. We select two SPEC CPU2006 benchmarks with high-bandwidth demands, *lbm* and *soplex*, and two benchmarks with low demands, *gromacs* and *hmmer*. The workload consists of eight tasks: two instances of each of these applications. We report the performance of the selected SPEC applications when they do not share the L2 cache but share the FSB (i.e., use Cores $0, 1, 2, 3$).

To quantify the overall system performance and fairness, we use the definitions proposed by Zhang et al. [18]. The normalized application performance is the application execution time under ideal condition divided by the execution time under the current condition. The ideal execution time is the performance achieved when the appli-

cation runs alone, with no resource contention. Table 3 summarizes the results for the four applications. Applications that share the FSB pay a considerable performance penalty (49%) using the default Linux scheduler. The memory-bus aware scheduler improves the system performance by 8.75% (from 0.49 to 0.54). The normalized performance of the two applications with high memory-bandwidth demands, *lbm* and *soplex*, improves the most.

Workload	*Linux*	*Membus*
lbm	0.54	0.63
soplex	0.38	0.44
gromacs	0.60	0.59
hmmer	0.47	0.51
all	0.49	0.54

Table 3: Normalized application and system performance w/o shared L2 caches.

As seen in the previous sections, application performance depends heavily on the memory demands of the other applications that run on the same CPU. Because the Linux scheduler implements cache-affinity, the initial assignment of applications to CPUs remains until the applications finish execution. Hence, the system throughput under Linux depends on the initial allocation to processors, which is a random process. As a result, application performance under Linux varies widely between consecutive executions of the same workload.

To validate the fairness of our scheduler, we compute the unfairness factor metric [18] for both schedulers. The unfairness factor is the coefficient of variation of all applications performance. Ideally, the unfairness factor should be 0, i.e., all applications should be affected by the same amount. We compute the unfairness factors for the two schedulers. The results show that, compared to the Linux scheduler, the memory-bus aware scheduler reduces unfairness by 4.17% (from 0.25 to 0.24), and the average coefficient of variation is reduced from 6.4% to 1.05%. The fairness results indicate that although the low-bandwidth applications are slightly slowed down, this does not induce unfairness in the system.

Our evaluation in Section 3 shows that the performance of high-bandwidth benchmarks can be improved by 3% to 40%. The demand-aware scheduler improves the performance of high-demand applications from 8% to 19%. For low-demand applications the measured potential for improvement is 1% to 28%. In some of the experiments, our scheduler imposes a performance penalty of up to 3% on low-demand applications. The performance improvements on these type of applications is up to 10%. All experiments indicate that our memory-bus–aware scheduler improves the system performance. Moreover, the scheduler's fairness is improved slightly.

4 The next generation

When we started this investigation, we did not single out the Xeon 5345 processor for its specific architecture; it was the first multi-processor system that incorporated quad-core processors that we could get access to. We focused on memory bandwidth since we expect that future exascale systems will demand a better coupling of applications and memory system than is realized today. The bandwidth asymmetries that we report here were not disclosed in the processor's data sheets and are likely to cause difficulties for software engineers who seek to achieve predictable performance from these multi-core systems. Given the range of options for the design of the memory/CPU interface of multi-core systems (and the constraints imposed by packages), we expect future exascale systems to exhibit different surprises for their users. Runtime information provided by a PMU offers an attractive approach to avoid performance pitfalls.

We recently evaluated a dual-processor system based on the Xeon E5520 (using a Nehalem-based single chip quad-core with a cache-coherent non-uniform memory architecture (NUMA)). When we measured the performance of this system, we encountered a new PMU with new challenges. This system has an on-chip integrated memory controller (IMC) on each processor connecting the processor to its share of the memory. The two processors are connected via a QuickPath Interface (QPI), and our measurements indicate that the memory bandwidth of the IMC and QPI is shared fairly between processor cores. In addition, the absolute performance is significantly increased; Figure 7 compares the aggregate memory bandwidth of an 8-core 5520-based system with an 8-core 5345-based system. We used the same memory intensive workload as in our previous experiments and varied the number of simultaneously executed tasks. E.g., the bandwidth achieved with eight tasks on the 5520-based system is more than six times the bandwidth obtained on the 5345-based system.

However, a NUMA system like the Nehalem-based system poses new challenges for applications that demand high memory bandwidth. And the trends in processor and memory design indicate that future exascale systems will embrace NUMA designs. The varying distance of the processors to the memory can significantly influence the performance. We obtained the throughput reported in Figure 7 by carefully scheduling the tasks and allocating the memory they use such that memory buses are not oversaturated, and cross-node traffic is avoided. Setting policies manually requires extensive knowledge of the memory behavior programs and the internals of the microarchitecture. And due to the default "first touch" memory allocation policy of many operating systems, the allocation of memory pages to NUMA nodes is highly non-

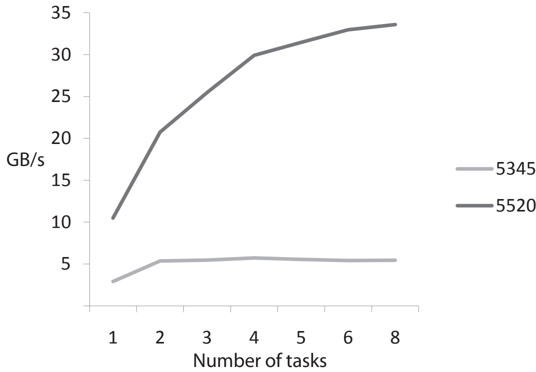

Figure 7: Aggregate bandwidth of 5520 versus 5345 based systems.

deterministic, and consequently programs frequently end up in system states where remote access to the memory is required. Information from the PMU can guide the task scheduler to address this challenge.

To assess how the performance of memory intensive tasks is hurt when they are scheduled on remote cores, we have measured how the bandwidth of the IMC is divided between local and remote cores accessing memory allocated on a single node, using the benchmark and methodology described above. We call instances of the benchmark accessing the memory locally *L tasks*, instances of the benchmark accessing remote memory *R tasks*. Both types of tasks read and write data from and to the memory of the same node. The number of L and R tasks varies from zero to four. E.g., a configuration of L = 2 and R = 3 means that two tasks are accessing memory locally and three remotely.

# of L tasks	# of R tasks 1	2	3	4
1	65%	96%	97%	96%
2	87%	63%	64%	64%
3	36%	51%	52%	51%
4	33%	48%	48%	48%

Table 4: Ratio of remote and local read bandwidth.

Table 4 shows the ratio of the read bandwidth obtained by remote and local tasks in different configurations (total remote bandwidth divided by total local bandwidth). The results show considerable variation for the different setups (and this variation would be even more pronounced if we compared the bandwidth obtained by individual tasks). Our experiments explored the performance of the system in many cases, and some of these may be cornercases, but the performance of the 5520-based system (and of many high-performance systems) largely depends on the saturation of the memory interconnects and the portion of the program's memory that is allocated on the local node or on the remote node. These parameters cannot be discovered at compile-time, the role of runtime monitoring is crucial in such cases.

5 Related work

The topic of performance measurements on multi-core system has been investigated by various projects. Molka et al. [12] investigate the overhead caused by the cache coherency protocol. Mytkowicz et al. and Weaver et al. investigate the effects of the Linux environment size or the benefits of static linking [14, 16], and Zaparanuks et al. [17] consider the effects of frequency scaling. All these aspects are not considered here since the benchmark programs are small and place a high demand on the memory system (the cost of memory access dominates), but different application settings and architectures will require a re-evaluation of these aspects.

Many researchers have investigated approaches to map applications onto multi-core systems, and appropriate information from a PMU is a crucial building block for these efforts. Bellosa [2] investigates the effect of limited memory bandwidth on process execution in the context of soft real-time systems. He presents a technique to satisfy guaranteed memory bandwidth demands and to throttle processes without guarantees. Each thread with a guarantee is assigned to its own CPU. The remaining bandwidth is equally shared among the free processors, and is used by threads without guarantees. If a thread exceeds the imposed limit in memory bandwidth, it is throttled.

Similar to our approach, Bellosa is using the kernel scheduler to balance among processes a resource different than CPU time. He also uses the PMU to gather information about the memory bandwidth. In contrast to our approach, Bellosa strives to ensure that the processes with guarantees are provided the required memory bandwidth and not to maximize the throughput on the memory bus. His work is based on the assumption that the memory bandwidth is shared symmetrically among CPUs. As described in our paper, this assumption does not hold on multi-core systems. Bellosa's algorithm could run into the situation where a process is scheduled on a core where the required memory bandwidth cannot be achieved.

Processor cores using the same memory controller simultaneously can also issue memory requests in such a way that they experience unexpected performance degradation. Hardware solutions for improving the throughput and fairness of the memory controller have been the subject of research to overcome this limitation. Mutlu et al. propose a scheduling algorithm that forms batches of

memory requests and then prioritizes the requests within a batch to maximize intra-thread bank-parallelism and also to maintain row-buffer locality. This approach may also improve memory-level parallelism and thus improve single-threaded performance [13]. Recently the authors improved their algorithm by employing cross-memory controller coordination and using an improved heuristic based on the least-attained-service obtained by the threads [6]. All these algorithms are tightly integrated with system software and take into consideration user-specified thread priorities. However, it is not clear that these approaches improve performance in NUMA systems that associate part of the memory space with a specific node. And our focus is on dealing with the realities of commodity processors with integrated memory and does not address possible changes to the (micro)architecture of these processors.

Antonopolous [1] describes a system that schedules jobs on an SMP system taking into account the bus bandwidth consumption. For each task, a fitness function is computed and tasks are scheduled based on these values. However, the approach introduces unfairness into the scheduling, since sets of tasks with a high fitness value are favored. Koukis and Koziris [8, 9] have an approach similar approach to Antonopoulos's. The fitness function however, captures beside the bandwidth between the memory and CPU, the bandwidth between the memory and network card as well. Their system suffers from the same unfairness problem and the assumption that the bandwidth is shared symmetrically between CPUs. More accurate performance monitoring data might allow a better mapping.

To use the L2 cache efficiently, Fedorova et al. [4] propose a cache-conscious scheduling algorithm. By scheduling threads in groups that have low cache miss ratios, the authors keep the miss ratio low and the rate of instructions per cycle (IPC) high. The authors present preliminary results from a prototype implemented partly at user lever, partly inside a simulator. Fedorova et al. also propose an IPC scheduler that strives to pair high-IPC tasks with low-IPC tasks to reduce resource contention [5]. However, Zhang et al. [18] show that an IPC scheduler does not always accurately reflect the utilization of the shared bus resource. Our scheduler uses the memory-bandwidth demands explicitly to schedule the applications.

Knauerhase et al. [7] demonstrated that runtime observation of task behavior can ameliorate the variability of application performance as well as increase performance. Zhuravlev et al. [19] compare several classification schemes and find that a scheme based on runtime hardware information provides reasonably accurate classification. Because this scheme is also computationally feasible, the authors use it in a prototype scheduler that improves performance and QoS experienced by programs.

Of course, multi-core systems may exhibit a different form of asymmetry if they employ heterogeneous cores. These architectures use cores with different performance characteristics. The main challenge of these architectures is to find the proper mapping between application and core. We mention the effort by Lakshiminarayana et al. [10] that maps the longest job to a fast core first. Saez et al. [15] describe an approach which leverages both efficiency specialization and thread-level specialization to exploit the performance of asymmetric multi-core systems. Again, better information about the properties of the target cores (and the characteristics of the applications) would provide a better platform to address the mapping problem for heterogeneous systems.

6 Concluding remarks

The performance observations reported in this paper are specific to the platforms that we used for our experiments. Future exascale systems will employ nodes that share many characteristics with these platforms. As long as memory bandwidth remains an issue in mapping applications to high-performance systems, the approach presented here (a scheduler that uses the PMU to deal with the system's limitations and bottlenecks) presents a promising baseline. An exascale system might require a different scheduler benefit function, and a new set of experiments will be needed to characterize the system. Certainly, the importance of OS tuning and specialization will increase in the future, and new multi-core designs will keep the OS community busy, placing a premium on flexible designs that allow customization.

As processor designs evolve (e.g., including an L3 cache on the die), the processor-OS interface may require a fresh look. Additional cache levels and larger caches certainly help some applications, but not all. And those applications with memory demands that cannot be satisfied by caches will benefit from schedulers that take properties of the memory systems (cache properties, asymmetries, memory bus structure) into account. Finding a good way to express these properties and to communicate them from the processor manufacturer to the OS developers is an open issue.

The PMU of our experimental system made it possible to obtain the data required to implement our on-line multi-core aware scheduler. That being said, there are many opportunities to improve PMU design for such use, and we are not at all confident that future PMU designs will adequately support our needs. There are three directions for hardware PMUs in exascale systems that —based on the experiments reported here—we feel should be pursued. First, hardware should allow simultaneous counting of a larger number of events, with fewer restrictions on what

can be counted simultaneously. Getting a handle on events that make sense in the context of an application or the OS requires often the use of multiple counters, e.g., to correlate the occurrence of one event on one core with the absence of some other event on another core. Second, the PMUs must provide more stability, both in terms of design and implementation. We have come to expect arbitrary additions, omissions, modifications, and defects in counter functionality, creating a significant barrier to stabilizing any sort of adaptive systems above the PMU. In our experiments we found an application as common as *gcc* triggered incorrect behavior. In the future, as compilers and operating systems rely more and more on the data provided by hardware PMUs, the behavioral consistency and correctness of the PMU will approach the importance of functional properties like the correctness of the floating point multiplication. Third, raising the level of interaction or observation from low-level events to higher-level events (that make sense for an application or the OS) would make it easier to stabilize the software. The trend to provide "architectural" performance counters in recent Intel and AMD systems is a positive step. Using this information, however, will remain a challenge for OS and tool developers in the years to come. We hope our simple prototype scheduler will encourage others to explore this frontier.

Acknowledgments

We thank Patrik Reali, Florian Schneider and Balázs Szintai for their comments and advice. We thank the reviewers for timely feedback.

References

[1] C. Antonopoulos, D. Nikolopoulos, and T. Papatheodorou. Scheduling Algorithms with Bus Bandwidth Considerations for SMPs. In *Proc. ICPP*, pages 547–554. IEEE Computer Society, 2003.

[2] F. Bellosa. Process Cruise Control: Throttling Memory Access in a Soft Real-Time Environment. Technical Report TR-I4-97-02, Univ. of Erlangen-Nuernberg, 1997.

[3] S. Eranian. What Can Performance Counters Do For Memory Subsystem Analysis? In *Proc. MSPC '08*, pages 26–30. ACM.

[4] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of Multithreaded Chip Multiprocessors And Implications For Operating System Design. In *Proc. USENIX 2005 Annual Technical Conference*. USENIX Association.

[5] A. Fedorova, C. Small, D. Nussbaum, and M. Seltzer. Chip Multithreading Systems Need A New Operating System Scheduler. In *Proc. ACM SIGOPS European Workshop*. ACM, 2004.

[6] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *Proceedings of HPCA-16*, 2010.

[7] R. Knauerhase, P. Brett, B. Hohlt, Tong Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *Micro, IEEE*, (3):54–66, May-June 2008.

[8] E. Koukis and N. Koziris. Memory Bandwidth Aware Scheduling for SMP Cluster Nodes. In *Proc Euromicro-PDP*, pages 187–196. IEEE Computer Society, 2005.

[9] E. Koukis and N. Koziris. Memory and Network Bandwidth Aware Scheduling of Multiprogrammed Workloads on Clusters of SMPs. In *Proc. ICPADS*, pages 345–354. IEEE Computer Society, 2006.

[10] N. Lakshiminarayana, S. Rao, and H. Kim. Asymmetry Aware Scheduling Algorithms for Asymmetric Processors. In *Proc. ISCA - Workshop on Operating Systems Computer Architecture*. ACM, 2008.

[11] J. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[12] D. Molka, D. Hackenberg, R. Schne, and M. S. Müller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. *PACT 2009*, pages 261–270, 2009.

[13] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Shared Memory Controllers. *IEEE Micro*, 29:22–32, 2009.

[14] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. of ASPLOS 2009*, pages 265–276. ACM, 2009.

[15] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Processors. In *Proc. EuroSys 2010*, 2010.

[16] V. M. Weaver and S. A. Mckee. Can hardware performance counters be trusted? In *Proc. of IISWC 2008*, pages 141–150, 2008.

[17] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of Performance Counter Measurements. Technical Report 2008/05, University of Lugano, September 2008.

[18] X. Zhang, S. Dwarkadas, G. Folkmanis, and K. Shen. Processor Hardware Counter Statistics as a First-Class System Resource. In *Proc. HOTOS Workshop*, pages 1–6. USENIX Association, 2007.

[19] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proc. of ASPLOS 2010*.