Memory Bandwidth Aware Scheduling for SMP Cluster Nodes

Evangelos Koukis and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
Zografou Campus, Zografou 15773, Greece
{vkoukis, nkoziris}@cslab.ece.ntua.gr

Abstract

Clusters of SMPs are becoming increasingly common. However, the shared memory design of SMPs and the consequential contention between system processors for access to main memory can limit their efficiency significantly. Moreover, the continuous improvement of modern cluster interconnection technologies leads to the network bandwidth being a significant fraction of the total memory bandwidth of the machine, thus the NIC of an SMP cluster node can also become a major consumer of shared memory bus bandwidth. In this paper we first provide experimental evidence that contention on the shared memory bus can have major impact on the total execution time of processes even when no processor sharing is involved, then present the design and implementation of an informed scheduling algorithm for multiprogrammed workloads, which tries to carefully select processes to be co-scheduled so that bus saturation is avoided. The input data needed by our scheduler are acquired dynamically, at run-time, using architecture-specific performance monitoring counters and a modified version of the NIC firmware, with no changes to existing application binaries. Experimental comparison between our scheduler and the standard Linux 2.6 O(1) scheduler shows average system throughput improvements in the range of 5-25%.

1. Introduction

Symmetric multiprocessors, or SMPs for short, have emerged as a cost-effective solution for parallel standalone servers or as building blocks for scalable clustered systems, when interconnected over a low-latency networking infrastructure. However, they have a fundamental architectural bottleneck, that hinders their ability to scale to large numbers of processors, namely their shared memory bus.

The most important feature of an SMP architecture is the sharing of all of main memory over a shared memory bus or other internal interconnection network, with the access latency to all memory locations being the same for every processor in it. This simplifies the process of programming the system, enabling the use of a shared address space with uniform access cost. On the other hand, the existence of a shared memory bus is also the most important barrier to system scalability.

Researchers have focused for long on the imbalance between the increase in CPU speed versus the increase in memory speed in modern systems. As the CPU speed of the fastest available microprocessors increases exponentially, at a rate of about 80% per year while the speed of memory devices is growing at a much slower rate of about 7% per year [17], the ratio of CPU to memory performance, or "Machine Balance" [15] becomes a deciding factor to determining overall system performance. Memory performance is defined not only in terms of available bandwidth but of memory latency as well. Pipeline stalls, because the required data is not yet available due to memory latency, impose a major performance hit in modern pipelined processors. To alleviate the latency problem and exploit temporal and spatial locality, processors in an SMP system do not communicate with main memory directly, but rather through a private hierarchy of caches, loading and storing cache line sized blocks. Using large caches and memory latency tolerance techniques, such as prefetching and speculative loads may hide latency but leads to higher bandwidth consumption [6]. There is a latency-bandwidth tradeoff, since much more data are transferred from main memory to the processor that are actually going to be used, in order to minimize access latency. However, without sufficient bandwidth, latency tolerance techniques become ineffective.

The limited memory bus bandwidth problem in SMPs is aggravated when SMP systems are combined to form large, distributed memory clustered systems. Cluster nodes

Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euromicro-PDP'05)
1066-6192/05 $ 20.00 IEEE

are commonly interconnected over advanced, high performance interconnection networks such as Myrinet [5] or SCI [9, 8] Their NICs feature embedded microprocessors and DMA engines, in order to offload the communication load from the CPUs and allow them to perform useful processing while they undertake moving messages directly to and from system memory. However, as their available communication bandwidth is constantly increasing, in the order of Gbps, so does their memory bus bandwidth consumption relative to the CPUs of the system. This affects the degree of computation-to-communication overlapping that can be achieved and limits overall application performance.

Most software for SMPs does not address the problem of limited memory bandwidth directly, rather it tries to minimize its effect by exploiting the cache hierarchy. On one hand, there are efforts [1, 20, 21] to make better use of available caches by employing sub-blocking and partitioning techniques, in order to improve the locality of references and minimize the cache miss rate. On the other hand, scheduling in SMP enabled Operating Systems takes into account CPU affinity constraints [23, 22, 19], by heavily penalizing inter-processor migrations when context-switching: a process migrating to a different CPU in the system would have to rebuild its state information in the new processor's private cache hierarchy. Thus it would exhibit a high cache miss ratio and increase the load on the shared memory bus. However, the problem still remains, if after applying these optimizations the available memory bandwidth does not suffice.

In this paper, we try to address the problem directly, by carefully selecting the processes that are assigned to processors and executed in parallel, so as to minimize their interference on the memory bus. We assume a multiprogrammed SMP cluster node, on which the memory bandwidth consumption of each process is a function not only of the cache miss ratio of its code executing on the CPU, but of its communication load as well. Thus, we account for the memory bandwidth consumption by the DMA engines on the cluster interconnect NIC, which are programmed directly by processes under a User Level Networking scheme.

Our goal is to monitor the memory bandwidth consumption of each process at run-time and use this information to assign processes to processors in such way, so as to neither saturate nor underutilize the shared memory bus. The scheduler is designed so that no code changes to executing applications are necessary. Monitoring data used by the scheduler are gathered transparently to application code. Memory bandwidth consumption by the CPU is monitored by exploiting the performance counters provided by most modern microprocessors [24], while memory bandwidth consumption by the DMA engines on the NIC is monitored by extending the firmware executing on it to support such functionality.

The effect of limited memory bandwidth on process execution in the context of soft- and hard- realtime systems has been investigated in [13] and in [4], where techniques are presented to satisfy guaranteed memory bandwidth demands and to throttle lower priority processes so that they do not interfere with the execution of higher priority ones. Our work also focuses on the effect of interference on the shared memory bus, however we seek to increase system throughput in multiprogrammed nodes, rather than meet strict deadlines.

The work in [3] is similar to ours, describing a system which uses source code changes to track the memory bandwidth usage of each application and coordinate their execution. Our approach is based on a monitoring framework which requires no application source code changes, but instead relies on OS and NIC firmware mechanisms in order to transparently monitor memory bandwidth usage. Furthermore, we try to monitor and take into account the pressure on the memory bus imposed by the NIC, when executing communication intensive applications.

In the rest of this paper, we first demonstrate the problem imposed by memory bus saturation by measuring the execution slowdown imposed on mixed benchmark workloads (Section 2), then based on our observations, propose memory bandwidth aware scheduling policies to alleviate it (Section 3). Section 4 describes the performance monitoring framework we designed, both at the CPU as well as at the NIC firmware side and the actual scheduler implementation. Finally we present an experimental evaluation of our scheduling policy compared to the standard Linux scheduler (Section 5) and our conclusions (Section 6).

2. Application slowdown due to memory bus saturation

In this section, we demonstrate the impact of memory bus saturation on overall system performance by quantifying the slowdown imposed on the execution of computationally intensive workloads. We take special care to ensure that the processes being executed do not share processor time or other system resources, and only contend for memory bus bandwidth. For benchmarking, we used the `bzip2`, `gzip` and `crafty` applications from the SPEC CPU2000 benchmark suite [10], the `BLAST` application for searching protein and nucleotide databases [2] and the `queens` computational kernel. We also developed two microbenchmarks, `membench` and `myribench` which are used to evaluate the impact of memory bus consumption by the CPU and the NIC respectively.

Our experimental platform is an SMP cluster node with two Pentium III@1266MHz processors, based on the Supermicro P3TDE6 motherboard (Serverworks ServerSet III HC-SL chipset). The chipset allows for dual channel ac-

cess to the two PC133 SDRAM 512MB DIMMs that are used, for a total of 1GB shared main memory. Each of the processors has a 16KB L1 Instruction Cache, a 16KB L1 Data Cache and a 512KB unified L2 Cache, with 32 bytes per cache line. For cluster interconnection, a Myrinet M3F-PCI64B-2 NIC is installed. It uses the LANai9 embedded microprocessor clocked at 133MHz, and 2MB of SRAM. The NIC is installed in a 64bit/66MHz PCI slot, which has a theoretical peak bandwidth of 528MB/s.

The system runs the Linux 2.6.6 kernel. We decided to use a 2.6 kernel version mainly for its use of a new enterprise-class O(1) scheduler, which has been completely rewritten since 2.4, with cache affinity and SMP scalability in mind. All benchmarks were compiled with the GCC v3.3.3 compiler, with maximum optimization (`-O3 -fomit-frame-pointers`). To estimate memory bandwidth consumption we used the perfctr library for manipulation of the CPU performance counters, as described in greater detail in Section 4.

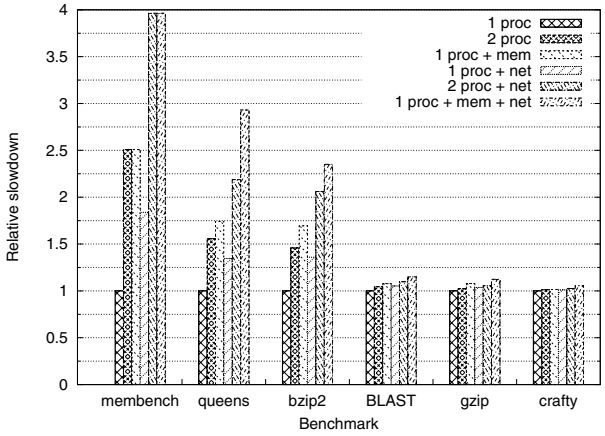

Figure 1. Application slowdown due to limited memory bandwidth

We ran six different sets of experiments (fig. 1). First, for each benchmark, we ran a single process of the application on one of the two processors. In this case the process runs with negligible interference from other processes, being the only runnable process in the system. Thus we can have a good estimate on the memory bandwidth it consumes on average, as depicted in table 1.

To isolate the work done in the CPU and the memory subsystem, and ensure that the processes remain purely computationally intensive and never leave the CPU to wait on disk I/O, we made sure that all data exchange from and to input and output files took place on an in-memory file system.

Benchmark	Description	BW req.
`bzip2`	Compression tool	460MB/s
`queens`	N-Queens solver	452MB/s
`BLAST`	Protein sequence search	95MB/s
`gzip`	Compression tool	45MB/s
`crafty`	Chess engine	21MB/s

Table 1. Description of benchmarks used

Along the application benchmarks, we also use the `membench` microbenchmark, to estimate memory bus bandwidth using strided array accesses. It allocates a block of B words, which is then accessed multiple times in an unrolled loop. If the access stride is s, then only words with indices $s, 1s, 2s, \ldots$ are references. By manipulating s it is possible to change the cache hit ratio of the benchmark. If L is the cache line size in words, then we can have two extremes: If $s = 1$, then when the first word of a cache line is first referenced a cache miss will occur, the line will be transferred into the cache and all $L - 1$ remaining accesses will be cache hits. In this case our microbenchmark exhibits excellent locality of reference. If $s = L$, then *all* accesses made by the microbenchmark will reference a different cache line. If B is much larger than the size of the L2 cache, we ensure that code execution causes back-to-back transfers of whole cache lines from main memory.

The first set of experiments shows that our benchmarks have quite diverse demands for memory bus bandwidth demands. Two of them, `bzip2` and `queens` pose heavy load on the memory subsystem, in the order of 400-500MB/s. This is attributed to `bzip2` performing indexed array accesses in a large working space of about 8MB, much larger than our L2 cache. `queens`, on the other hand, employs random hill climbing to solve the N-Queens problem for 10^6 queens, using a linked list representation for the search space. Thus, it performs mainly irregular pointer-chasing memory accesses. The `BLAST` benchmark has medium bandwidth requirements, while `gzip` and `crafty` are in the lower end, needing no more than 20-50MB/s on our platform.

Running `membench` with $s = L$ reports that the peak memory bus bandwidth on our platform is 981MB/s, a result which was verified by the STREAM memory benchmark [14]. Thus, running an application other than `membench` using only one process does not lead to memory bus saturation.

For the second set of experiments, we ran two processes of each benchmark in parallel. The number of runnable processes does not exceed the number of available processors, so there is no processor sharing involved. However, we can

see that the two memory-intensive benchmarks, `bzip2` and `queens` suffer a major performance hit, in the order of 45-55%, with the performance hit decreasing according to the memory bandwidth requirements of each benchmark. This performance degradation is the result of interference between the processes, because of contention on the shared memory bus.

The third set of experiments involves running a benchmark process in parallel with a `membench` process, in order to push the memory bus to saturation and pronounce the effect of the memory bottleneck. The execution results are also indicative of the performance the applications would exhibit on a 4-way SMP system, where it would be much easier to saturate the memory bus. Again, `bzip2` and `queens` slow down considerably, 74.2% and 69.5% respectively, but this time `BLAST` and `gzip` are also affected, running 7.7% and 7.9% slower. On the other hand, crafty, which has very low memory bandwidth demands, is not affected significantly, with a slowdown of only 1.5%.

For the final sets of experiments, we repeated the three previous measurements. This time however, the firmware on the Myrinet NIC was programmed to perform packet transmission and reception from and to main memory while the workloads were executing, in order to demonstrate that the NIC of a modern cluster interconnect can also be a major memory bus bandwidth consumer and quantify the effect of overlapping computation with communication to overall system performance. This does not impact the CPU time that is available to the workload, since all memory accesses are coordinated by the firmware executing on the LANai and performed using DMA engines on the NIC.

The results of the experiments involving network traffic show that network traffic can limit significantly the bandwidth that is available to the CPUs. For `bzip2` and `queens`, the performance degradation when one process executes in the presence of network traffic (34-36%) is comparable to the slowdown imposed by two concurrently executing processes. In the last set of experiments, when one benchmark process contends for access to memory with an instance of `membench` and the NIC, the impact of memory bus saturation ranges from a 5.5% slowdown for crafty, to a 230% slowdown for `bzip2`.

We should also note that performance degrades even when the cumulative bandwidth demand of both CPUs and the NIC does not exceed 981Mb/s, the peak bandwidth of our memory bus (for example, when two instances of `queens` execute in parallel). To better demonstrate this, we have plotted in fig. 2 the bandwidth consumption of two instances of `membench` executing concurrently, as well as their cumulative bandwidth usage. The first instance issues memory operations at the highest possible rate, which the second thread is throttled by inserting exponentially-distributed idle periods between consecutive

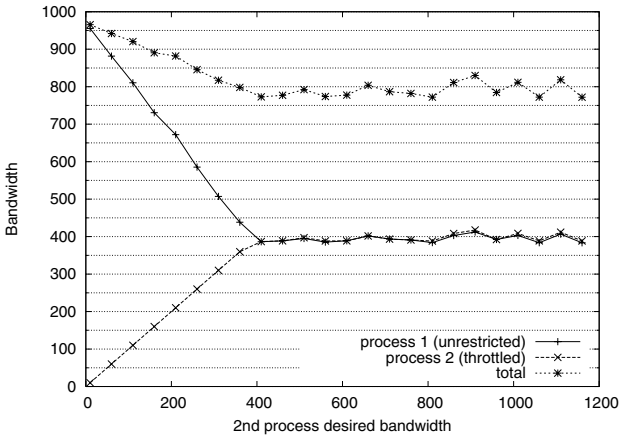

Figure 2. Memory contention and bus arbitration overhead

block accesses. We notice that as the demands of the second process increase, the cumulative bandwidth decreases disproportionately. This indicates that the effective memory bandwidth can become substantially smaller than its peak value, most probably due to the overhead of bus arbitration.

3. Memory bandwidth aware scheduling policies

Motivated by these observations of the slowdown imposed by contention on the shared memory bus, we design a new scheduling policy that aims at maximizing throughput when executing computationally intensive workloads by making more efficient use of the available memory bus bandwidth.

The experimental results presented in the previous section indicate that it is the applications with high memory bandwidth requirements (`bzip2` and `queens`) that are affected the most when competing for bandwidth on the shared memory bus. On the other hand, applications which have low cache miss rates (`gzip`, `crafty`) are largely unaffected by bus contention. However, generic operating system schedulers, such as the O(1) scheduler included in current Linux 2.6 kernels, aim at maximizing processor utilization and minimizing response time for interactive applications, without taking into account the effects of contention on the memory bus. Thus it makes sense, in the context of multiprogrammed SMP cluster nodes executing computationally intensive workloads, to employ a scheduling policy which tries to avoid memory bus saturation by carefully selecting processes to be co-scheduled on the available processors. This way serialization on the memory bus is avoided and a high degree of parallelism can be

sustained, reducing the total execution time of applications and increasing system throughput.

Our algorithm accepts as input a set of n applications (tasks, jobs) to be scheduled. Each *application* comprises a set of related processes, with p_i being the number of processes belonging to the i-th application, $0 \leq i \leq n$. A *process* is the schedulable entity, as seen from the point of view of the underlying operating system, e.g. the Linux kernel. Often an application executes on only one process. However, especially in the case of MPI jobs executing on a cluster of SMPs, it is possible that an application executes on more than one processes, in order to take advantage of more than one CPUs. These processes usually perform intra-node communication via shared memory areas, obtained using System V IPC mechanisms. In the context of fine-grained parallel applications, it is important that related processes are scheduled to execute simultaneously ("gang scheduling", [7, 11, 12]), so as to reduce the context switching overhead, minimize synchronization latencies and ensure that the parallel job makes sufficient progress. This is taken into account by our scheduling policy, which makes sure that the required number of processors is available for all related processes of an application to execute concurrently, if this application is to be scheduled on the next time quantum.

A real-time monitoring framework is assumed, that allows the algorithm to sample, for the j-th process of application i, the memory bandwidth BW_{ij}^{C} consumed by the process code executing on a CPU as well as the bandwidth BW_{ij}^{N} consumed by the NIC(s) on which the process has open ports.

The generic form of the scheduling algorithm is presented in algorithms 1, 2. The applications are organized in a doubly linked list. The algorithm runs once before every time quantum of length q and decides on the processes that will be executed on all processors of the system for its duration, thus all CPUs context switch simultaneously.

First, the scheduler makes sure that all processes executing in the previous time quantum are stopped, then uses the performance monitoring framework to collect statistical data on their memory bandwidth consumption. This information is used as input to a heuristic, which allocates the P CPUs in the system to applications, by selecting the members of the set of applications which will run in the next time quantum. Initially, the set of applications to be co-scheduled is empty. The first application on the list is always added to it, in order to prevent processor starvation and ensure that every process will eventually get scheduled to execute. In the sequel, the heuristic continues selecting applications and adding them to the set, keeping count of the yet un-allocated processors and the remaining memory bus bandwidth (initially M), as each application that is added to the set consumes a portion of the available memory bandwidth. The effect of memory bus contention presented in fig. 2 must be taken into account when determining M.

Algorithm 1: reschedule

```
1  begin
2      while the list of applications is not empty do
3          stop all previously executing processes
4          sample the performance monitoring counters
5          select a new set of apps to be scheduled
6          set a timer to expire after tq seconds
7          foreach application i do
8              if selected_to_run_i do
9                  foreach process j of application i do
10                     signal proc. j to begin execution
11                 end foreach
12             end if
13         end foreach
14     end while
15 end
```

Since it is impossible to know the memory bandwidth demands of an application beforehand, a prediction is made based on the bandwidth consumption of its processes during the w previous time quanta. We use the notation $BW_{ij}^{N,-k}$ to denote the memory bus bandwidth consumed by the NIC on behalf of process j of application i in the k-th previous time quantum. The run-time parameter w defines a sliding window over which the memory access rate of every process is averaged. The choice of w could affect the efficiency of our scheduler: It needs to be large enough to smooth out short bursts of memory activity, while at the same time being small enough for our scheduler to be able to adapt quickly to changes in the memory behaviour of processes.

In order to select the next application to be added to the set, an "inverse fitness" value is computed for all jobs not yet in the set. The heuristic divides the available memory bandwidth among the yet un-allocated processors (BW_{rem}/p_{rem}) and then selects the application whose average memory bandwidth consumption per process best matches that:

$$inv\_fitness(i) = \\ = \left| \frac{1}{w} \frac{1}{p_i} \sum_{k=1}^{w} \sum_{i=1}^{p_i} \left(BW_{ij}^{C,-k} + BW_{ij}^{N,-k} \right) - \frac{BW_{rem}}{p_{rem}} \right|$$

The use of this metric favors co-scheduling jobs with high bandwidth demands along with jobs that do not consume a lot of memory bandwidth. Once demanding applications have been selected, the ratio BW_{rem}/p_{rem} becomes very low, so jobs with low demands are more likely to be selected next. The inverse also holds.

Algorithm 2: select_applications_for_next_tq

```
1  begin
2      foreach application i in list do
3          selected_to_run_i ⟵ false
4      end foreach
5      /* The first application always gets scheduled */
       selected_to_run_0 ⟵ true
6      p_rem ⟵ P − p_0
7      BW_rem ⟵ M − BW_0
8      while p_rem > 0 do
9          bestf_i ⟵ +∞
10         bestapp ⟵ −1
11         foreach application i in list do
12             f_i ⟵ inv_fitness(i, p_rem, BW_rem)
13             if p_i ≤ p_rem ∧ f_i < bestf_i do
14                 bestf_i ⟵ f_i
15                 bestapp ⟵ i
16             end if
17         end foreach
18         selected_to_run_bestapp ⟵ true
19         move bestapp at end of list
20         p_rem ⟵ p_rem − p_bestapp
21         BW_rem ⟵ BW_rem − BW_bestapp
22     end while
23 end
```

4. Scheduler implementation details

The scheduling policy described in the previous section is well suited to an implementation in userspace, according to which scheduling decisions are taken by a process running with escalated privileges, and the signaling mechanism provided by the underlying OS is used to pause and resume the execution of processes being managed. The userspace scheduler ensures that the number of runnable processes never exceeds the number of CPUs, so that no context-switching is done by the OS kernelspace scheduler and there is no time-sharing involved.

The main advantage of this approach is that a userspace implementation is much simpler than one in kernelspace. Moreover, while the kernel views and schedules processes independently of each other, a userspace scheduler can be much better informed on higher-level semantic relationships between processes, taking into account, for example, that certain processes all belong to the same MPI job. The biggest drawback of such implementation is that there is not a simple mechanism for the userspace scheduler to be notified when the processes being managed leave the CPU and block on I/O. In this case the CPU is left idle, while a kernel scheduler would context-switch to a different runnable process. However, in our case we are concerned with computationally intensive workloads and computation-to-communication overlapping, meaning that processes do not block on I/O.

This section describes a userspace implementation of our scheduling policy, the MEMory Bandwidth aware Userspace Scheduler (*MemBUS*). MemBUS runs as a privileged process, so as to be able to create processes belonging to different users, and listens for job creation requests at a UNIX domain socket. Whenever a new request is received, it creates the needed processes and attaches to them using the `ptrace` system call. This allows MemBUS to control a process completely, creating and sampling virtual performance counters connected to it, and also monitoring the signals it receives. During its operation, at each time quantum, MemBUS samples its input data, then uses the standard `SIGSTOP` and `SIGCONT` signals to perform the context switch.

4.1. Monitoring CPU memory bandwidth consumption

Since CPUs in SMPs do not communicate with main memory directly but rather through a multi-level hierarchy of cache memories, estimating the memory bandwidth consumption of a CPU essentially means being able to monitor the bus ransactions used to load and store cache lines to and from the highest level cache, that is closest to main memory.

To monitor the memory behavior of applications without needing any modifications to their source code, we decided to use the performance monitoring feature, as provided by most modern microprocessors in the form of performance monitoring counters. These are machine-specific registers which can be assigned to monitor specific events, usually through the use of privileged instructions (`RDMSR` and `WRMSR` in the case of the x86 architecture). The counters may track the frequency or duration of events taking place in various functional units of the processsor, such as the number of floating-point instructions executed, or the number of branch mispredictions. In our case, we are interested in monitoring the Data Cache Unit, and more specifically the number of bus transactions to transfer whole cache lines between the main memory and the L2 cache (cache fill or writeback operations).

There are two obstacles for performance monitoring counters to be used effectively by MemBUS. First, the performance counter manipulation are usually privileged and can only be issued from kernelspace. Second, they are a per CPU, not a per process resource. If we are to count bus transactions and other events individually per process, i.e. only when that process is executing on a (random) CPU, the counters need to be *virtualized*, similarly to the way each process has a private view of the processor's register set, although it runs in a timesharing fashion and may mi-

grate to other processors. Thus, the Operating System needs to be extended, so that it sets up monitoring of a process's events before context switching into it, and samples the performance monitoring counters when its time quantum expires.

In our case, the virtual performance counter functionality was provided under Linux using the perfctr library [18]. Perfctr comprises a linux kernel module and a userspace library. The kernel module code runs in privileged kernelspace and performs functions such as programming the performance counters and sampling their values at every context switch, while the userspace library communicates with the kernel module via the system call layer and exposes a portable interface to programmers, in order to set up and monitor virtual performance counters.

4.2. Monitoring NIC memory bandwidth consumption

To determine the memory bus bandwidth demands accurately, our scheduler needs to take into account the contention between the system CPUs and the NIC for access to data residing in main memory. However, the OS-bypass, User Level Networking characteristics of modern cluster interconnection architectures make it difficult to intercept the communication process and monitor the communication load in a way that is transparent to the application. The greatest part of the communication functionality now resides within the NIC and is implemented in firmware executing on embedded microprocessors onboard it. Since the Operating System is not in the critical path of communication, we have to make modifications to the NIC firmware and provide a method for MemBUS to access the monitoring information directly.

Our testbed is based on Myrinet NICs and the GM message passing API for communication between nodes. Myrinet uses point-to-point 2+2Gbps fiber optic links, and wormhole-routing crossbar switching. In contrast to conventional networking technologies such as TCP/IP, it aims at minimizing latency by offloading communication functions to an embedded RISC microprocessor onboard the NIC, called the LANai, and removing the Operating System from the critical path.

GM [16] is the low-level message passing interface for Myrinet, providing high bandwidth, low latency ordered delivery of messages. It comprises the firmware executing on the LANai, a Linux kernel module and a userspace library. GM allows a process to exchange messages directly from userspace, by mapping part of LANai memory (a so called *GM port*) to its virtual address space. This is a privileged operation that is done via a system call to the GM kernel module, but afterwards the process may directly manipulate the corresponding send and receive queues in order to request DMA transfers to and from pinnned userspace buffers. The OS is bypassed entirely, since all protection and communication functions (memory range checking, message matching, packet (re-)transmissions), are undertaken by the LANai. Moreover, since all message data are transferred over DMA, the CPU is free to perform useful computation, and computation-to-communication overlapping is possible.

To monitor the memory bandwidth consumption of the NIC, we modified the firmware portion of GM-2, adding two 64-bit counters per GM port, which reside in LANai memory. The value of the counters is updated by the firmware whenever a DMA transaction completes from or to host main memory, so that they reflect the total amount of data transferred in each direction. The kernel module portion of GM-2 was also extended to include a "read counters" request, which is used by the scheduler in order to periodically sample the values of these counters and copy them to userspace.

5. Experimental evaluation

To evaluate the efficiency of our scheduling policy, we experimentally compared our userspace scheduler implementation to a modern SMP scheduler, the O(1) scheduler that is part of the Linux 2.6 kernel series. We measured the time required to complete the execution of a series of multiprogrammed workloads both under the Linux scheduler and MemBUS. The workloads consisted of a mixture of low and high bandwidth applications, together with instances of `membench` and `myribench`.

In the first set of experiments, each workload consisted of two instances of `membench` along with two instances of an application benchmark. The execution time of each `membench` instance was selected so that it matched the execution time of a benchmark process when executing on its own. Since the bandwidth demands of the benchmarks vary considerably, we can observe the behaviour of our scheduler in various combination of high-bandwidth (i.e. the instances of `membench`) and low bandwidth jobs. All experiments were run with a time quantum of $tq = 0.7s$ for MemBUS, which is about two or three times the time quantum of the underlying Linux scheduler (100-200ms). Lower values of tq would not make much sense, since they could interfere with the decisions made by the Linux scheduler and might increase the scheduling overhead considerably. We experimented with various values of the sliding window w used to determine the bandwidth demands of applications, but did not observe significant differences in the behaviour of our scheduler, probably because the memory behaviour of our benchmarks changes relatively slowly over time. The results presented in this section use a value of $w = 1$. The execution times, averaged over ten runs, are

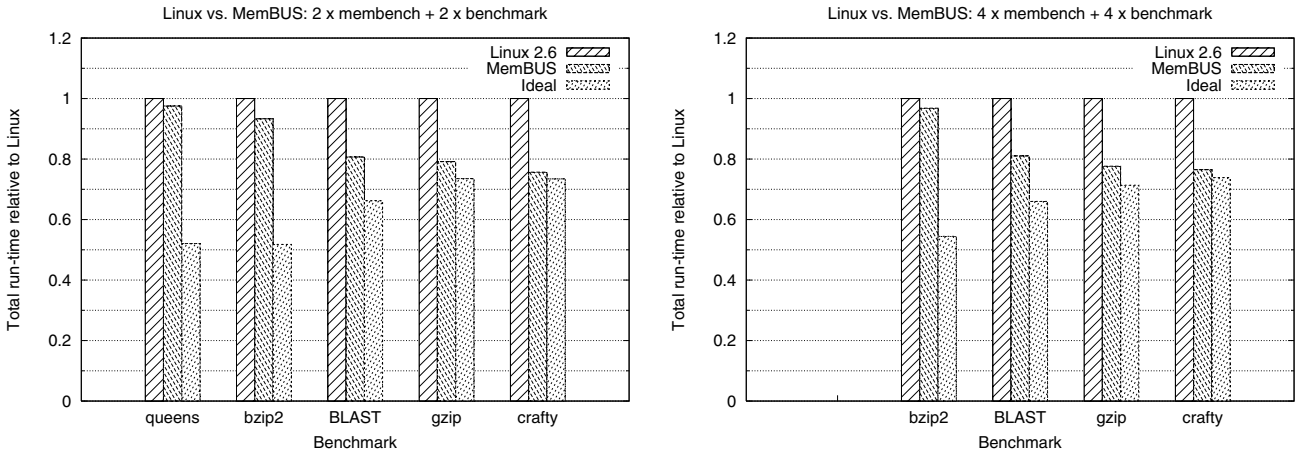

Figure 3. Comparison of workload execution times

presented in the left part of fig. 3 and are normalized relative to the execution time of the Linux scheduler.

We can see that the improvements in the total execution time are significant, ranging from about 4% in the case of queens to 25% in the case of crafty. The relative performance of MemBUS increases as the bandwidth demands of the jobs in the workload become more diverse, as this allows it greater flexibility to match low with high bandwidth demanding jobs and eliminate or at least reduce the effects of memory bus saturation. On the other hand, the improvement is smaller in the case of `queens` and `bzip2` since the performance gain of co-scheduling an instance of `bzip2` with one of `membench` is relatively small compared to co-scheduling two instances of `bzip2` or `membench`. It is also interesting to compare the performance of MemBUS to the ideal execution time. This is the expected execution time if there was *no* interference between processes, computed as the sum of CPU times divided in half, since we are using a dual processor system. In the cases where memory bus saturation can be avoided almost completely (`gzip` and `crafty`), the performance of MemBUS actually comes very close to this prediction.

In the second set of experiments, we increase the degree of multiprogramming, by executing workloads consisting of four instances of membench and four instances of a benchmark. The results are presented in the right part of fig. 3. Unfortunately we were unable to run this experiment for `queens`, since the total memory requirement of four instances exceeded the total memory of the node. Again, MemBUS delivers average throughput increases in the range of 5-25%.

Finally, in fig. 4 we have plotted the execution times for workloads containing instances of `myribench`, which emulates a communication intensive application, causing the NIC to consume a large portion of the available memory bus bandwidth (approx. 350MB/s).

The way MemBUS reduces completion times for multiprogrammed workloads can be better seen in fig. 5, where we have plotted the CPU time consumed by each process individually, both under Linux and under MemBUS, for various workload executions. When executing under the Linux scheduler we see that it is the memory intensive applications, the instances of `membench`, that suffer the greatest slowdown, while the low bandwidth demanding applications remain relatively unnafected. On the other hand MemBUS minimizes interference on the shared memory bus, reducing the time that memory intensive applications stall on memory references and thus lowering their total CPU time requirement.

The results on the throughput improvement of MemBUS compared to the standard Linux scheduler are averaged over multiple runs. It is worth noting, however, that the execution time of the workloads varied widely when run directly over the Linux scheduler, while the execution time under MemBUS remained almost constant. Some examples are presented in fig. 6. This can be attributed to the cache affine properties of the scheduler: Interprocessor migrations of processes are penalized heavily when making scheduling decisions, and are rare. If two memory intensive processes are affine to the same processor, they cannot interfere with each other, since they do not run simultaneously. Thus, the execution time under Linux depends significantly on the way the processes of the workload are distributed initially among processors, which is essentially random.

6. Conclusions

Contention on the shared memory bus can significantly limit the performance of an SMP cluster node when executing multiprogrammed workloads. Memory bus satu-

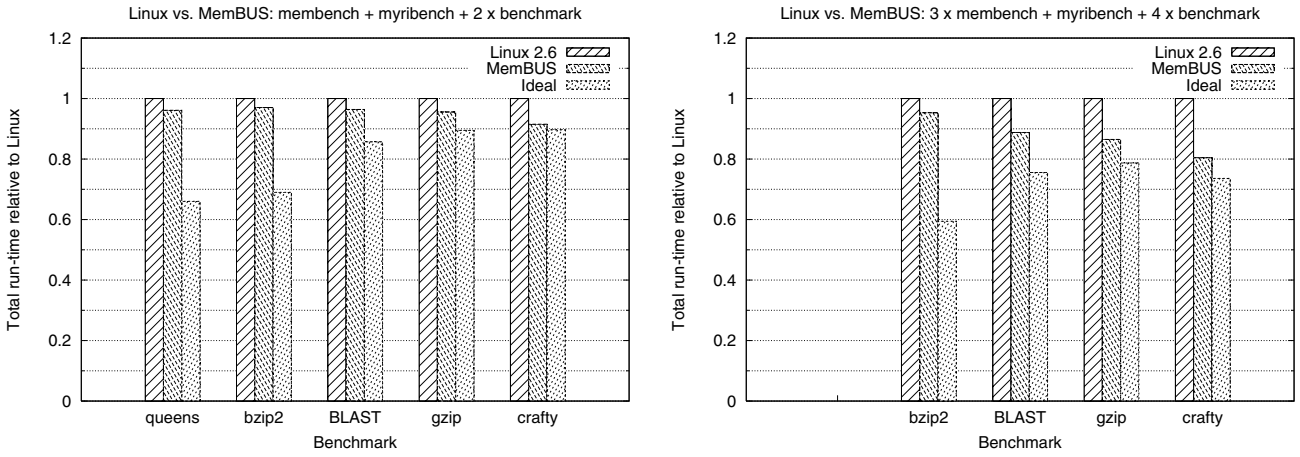

Figure 4. Comparison of workload execution times (including myribench)

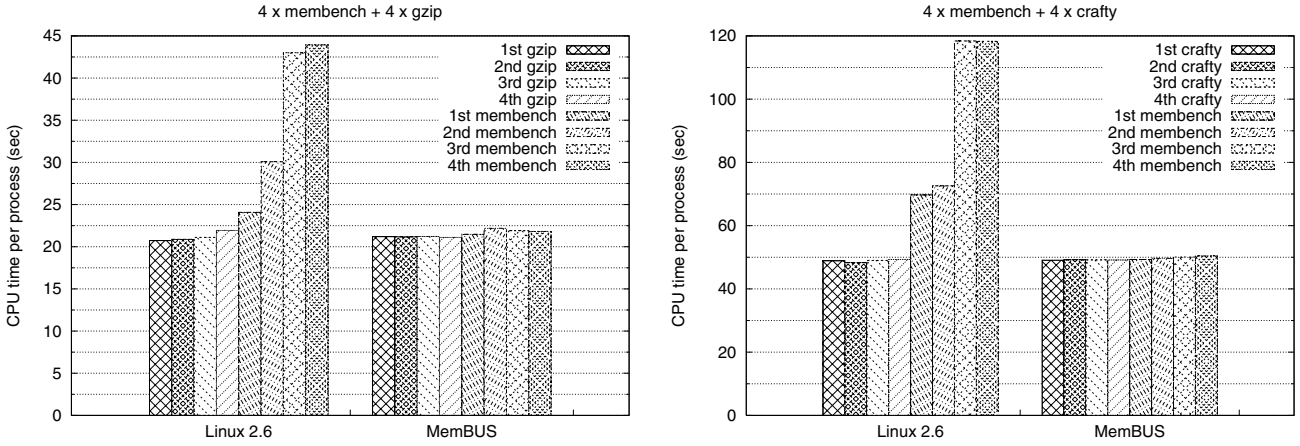

Figure 5. CPU times for each process of the workload individually

ration can lead to significant execution slowdowns, since high bandwidth demanding processes executing in parallel are effectively serialized when performing memory accesses. Motivated by our observations, we introduced a performance monitoring framework, which allows for realtime monitoring of CPU and NIC bandwidth consumption, then used it to implement a memory bandwidth aware scheduler. Experimental comparison between our scheduler and the standard Linux 2.6 scheduler showed a significant reduction in the CPU time required by high bandwidth processes, leading to an average 5-25% increase in system throughput, as well as more predictable execution times. We expect the performance improvement to increase with the number of processors in the system.

In the future, we will continue in two directions. We plan to extend our scheduling algorithm so that contention for shared resources apart from the memory bus is taken into account. For example, in a multiprogrammed SMP node processes may also contend for access to the interconnection link bandwidth, or to functional units and shared levels of the cache hierarchy in the case of SMTs. Also, we plan to investigate moving part of our scheduler implementation to kernelspace, in order to make it applicable to workloads featuring not only computationally intensive but also I/O intensive applications.

References

[1] D. Agarwal and D. Yeung. Exploiting Application-Level Information to Reduce Memory Bandwidth Consumption. In *Proceedings of the 4th Workshop on Complexity-Effective Design, held in conjunction with the 30th International Symposium on Computer Architecture (ISCA-30)*, Jun 2003.

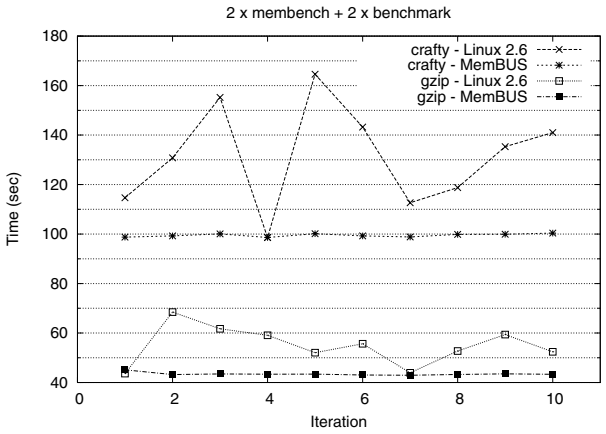

Figure 6. Variation of execution times under the Linux 2.6 scheduler and MemBUS

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Meyers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, Oct 1990.

[3] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling Algorithms with Bus Bandwidth Considerations for SMPs. In *Proceedings of the 2003 International Conference on Parallel Processing (ICPP 2003)*, page 547, Oct 2003.

[4] F. Bellosa. Process Cruise Control: Throttling Memory Access in a Soft Real-Time Environment. Technical Report TR-I4-02-97, IMMD IV - Department of Computer Science, University of Erlangen-Nürnberg, Jul 1997.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb 1995.

[6] D. Burger, J. R. Goodman, and A. Kagi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA-23)*, pages 78–89, May 1996.

[7] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, Dec 1992.

[8] F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B. Johnsen, H. Kohmann, R. Nordstrom, and P. Werner. Low Level SCI software functional specification-Software Infrastructure for SCI. ESPRIT Project 23174. http://www.dolphinics.com/ downloads/ nt/ pdf_zip/ SISCI_API-2_1_1.pdf.

[9] H. Hellwagner. The SCI Standard and Applications of SCI. In H. Hellwagner and A. Reinefield, editors, *Scalable Coherent Interface (SCI): Architecture and Software for High-Performance Computer Clusters*, pages 3–34. Springer-Verlag, Sep 1999.

[10] J. L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, 2000.

[11] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda. Implementation of Gang-Scheduling on Workstation Cluster. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 126–139. Springer-Verlag, 1996.

[12] M. A. Jette. Performance Characteristics of Gang Scheduling in Multiprogrammed Environments. In *Proceedings of the 1998 IEEE/ACM Supercomputing Conference on High Performance Networking and Computing (SC98)*, San Jose, California, Nov 1997.

[13] J. Liedtke, M. Völp, and K. Elphinstone. Preliminary Thoughts on Memory-Bus Scheduling. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 207–210. ACM Press, 2000.

[14] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

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

[16] Myricom. GM: A Message-Passing System for Myrinet Networks, 2003. http://www.myri.com/scs/GM-2/doc/html/.

[17] D. Patterson and J.Hennessy. *Computer Architecture. A Quantitative Approach*, pages 373–504. Morgan Kaufmann Pub., San Francisco, CA, 3rd edition, 2002.

[18] M. Pettersson. The Perfctr Linux Performance Monitoring Counters Driver, 2004. http://user.it.uu.se/ mikpe/linux/perfctr/.

[19] M. S. Squiillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, 1993.

[20] G. E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 117–, 2002.

[21] G. E. Suh, L. Rudolph, and S. Devadas. Effects of Memory Performance on Parallel Job Scheduling. *Lecture Notes in Computer Science*, 2221:116–, 2001.

[22] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, 1995.

[23] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 26–40. ACM Press, 1991.

[24] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis using the MIPS R10000 Performance Counters. In *Proceedings of the 1996 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC96)*, page 16. ACM Press, Nov 1996.