Preliminary Thoughts On Memory-Bus Scheduling

Jochen Liedtke Marcus Völp Kevin Elphinstone

System Architecture Group, CS Department
Universität Karlsruhe
{liedtke,völp,kevinelp}@ira.uka.de

1 Rationale

Main memory is typically significantly slower than the processors that use it. Such slow memory is then amortized by fast caches. Effective scheduling, particularly for soft or hard real-time, has therefore to include cache control, even on uniprocessors. Although cache scheduling is currently still an open research issue, we assume in this paper that uniprocessors are effectively schedulable in the presence of caches.

In this paper, we focus on SMP-specific memory scheduling. A small SMP multiprocessor typically incorporates multiple processors (with per-processor caches) that work on a global main memory. Processors and main memory are connected by a single memory bus, i.e. all processors share the same bus.

Assume that we have 4 job mixes that can be correctly scheduled on 4 independent uniprocessors. What happens if we put those 4 job mixes on a 4-processor system with a single memory bus? Without any additional scheduling provisions, the shared memory bus can, in the worst case, stretch each schedule by a factor of 4. This is clearly unacceptable. In general, it would mean that the real-time capacity of an n-processor system is only $1/n$ of the capacity of a uniprocessor system. Multiprocessors would be unusable for real-time applications.

Therefore, memory-bus scheduling is desirable. Memory-bus scheduling should enable us to give soft and perhaps even hard guarantees in relation to memory bandwidth and latency to real-time applications. For non real-time applications, it should help optimize a system's overall throughput and/or latency.

Related Work Work in this area seems to be rare. The author's are only aware of Frank Bellosa's work [**?**].

2 The Scenario

Hardware

Our hypothetical SMP system consists of n processors that share a single memory unit/bus. We assume ideal hardware that multiplexes and arbitrates the bus at zero cost. Memory is always accessed in cache-line units. Each such read or write is called a *memory transaction.* To keep our model simple, we also assume that each memory transaction has a constant cost of time c, i.e. m transactions need time mc, independent of whether the transactions are reads or writes, come from a single or from multiple processors, access adjacent or randomly scattered memory locations.

We need some hardware support to monitor and control bus transactions, e.g. performance-counter registers on the Pentium processor. We assume readable performance counters per processor that monitor

- the number of memory transactions executed on processor i,
- the time for which the processor's bus unit was active (including bus wait time) for those transactions.

Performance counters should also be loadable such that they generate a processor interrupt once a predefined value is reached.

Memory transaction quanta can thus be allocated to processors by appropriately loading the the counters such that each processor raises an interrupt as soon as it has exhausted its allocation.

Δ-Windows

Applications specify their requirements and expect to get guarantees from the scheduling system. Typically, the specified numbers are averages that refer to certain windows in time. For example, processor i requires k_i memory transactions in time Δ, however, the transactions can be arbitrarily distributed over the Δ-window.

Note that the sliding Δ-windows do not form a fixed pattern in time, i.e. they do not slide between fixed time intervals. Δ-windows are free to begin at any appropriate time. Requirements and guarantees always refer to the relevent sliding window, e.g. "processor i requires a rate of R memory transactions per Δ-window."

Low and High State

We assume a simple two-priority model for memory requests. Memory transactions of a processor currently in *low state* may be arbitrarily delayed. Typically, low-state processors will compete in a best-effort manner for the memory bus. Conversely, *high-state* processors need to be served with priority to fulfil scheduling guarantees.

Later, it will be an interesting point of discussion whether this simple model suffices for realistic scheduling policies. Here we primarly use it to illustrate the scheduling problem.

Scheduling Problem

Due to practical importance and generality, we focus on a system that runs a mix of real-time and non-real-time applications. Any real-time application specifies its required resources and requests hard or probabilistic guarantees so that it can meet its deadlines, i.e. so that its execution will not be delayed more than a certain upper limit. Non-real-time applications are not controlled by deadlines, and consequently, they attempt to utilize the remaining resources in the best possible way.

Processors currently executing a non-real-time application obviously are in low state. Processors executing a real-time application are in high-state as long as they do not exhaust their guaranteed resources. The relevant questions are:

1. *How do real-time applications specify their memory-bus requirements?*

2. *Given these requirements, how can we implement admission control?*

3. *Given an admitted memory-transaction schedule, how can we enforce it?*

3 A Naive Approach

3.1 Simple Bandwidth Limitation

Key assumption of this approach are:

1. Requirements can be specified as bandwidth requirements.

2. Admission control can be based on adding bandwidth requirements.

3. Enforcing maximum bandwidth enables bandwidth guarantees.

Each processor specifies its requirements as R_i, which is the requested number of memory transactions per Δ-window. Non-real-time applications specify $R_i = 0$ which means they do not have hard requirements but will try to get as much bandwidth as possible. Real-time applications can also exceed their specified requirements. However, they run only in high state while $M_i < R_i$. Once their claimed requirements are exhausted, they switch to low state and are then treated like non-real-time applications for the remaining Δ-window.

How can we ensure that all high-state processors, i.e. those with $M_i < R_i$ get enough memory bandwidth? A naive idea would be to allocate only free memory bandwidth to low-state processors, basically only $\Delta/c - \sum R_i$. This solution has two serious disadvantages: (1) It is wrong (see Section 3.2). (2) It is not very useful in general because of its inflexibility and its insufficient bus utilization when processors do not exhaust their guaranteed bandwidths. In particular, it seems totally infeasible for mixing best-effort and real-time applications.

The fundamental reason for (2) is the fact that we always conclude from lower effective memory bandwidth consumption that the processor did not get enough memory bandwidth, i.e. we assume bus contention. However, effective memory bandwidth can drop without bus contention occuring. Cache miss-rates change when the processor runs multiple threads with different characteristics, and even a single thread will have phases of high and low memory bandwidth consumption.

A processor's bus time tells us how much the processor suffered from bus contention: $S_i = T_i - M_i c$ is the bus stall time for the current Δ-window. Adaptive methods to distribute unused high-state memory-bus bandwidth to low-state processors might thus be possible.

3.2 A Counter-example

Before we discuss possible scheduling algorithms in more detail, we should first focus on whether our assumptions (1-3) were correct.

Assume processor P_1 runs a program that repeatedly accesses memory for one time unit, and then does calculations for two time units. The program on P_2 is similar but needs only one time unit for calculations between two memory accesses. P_1 requires 33% memory bandwidth, and P_2 requires 50%. Combining P_1 and P_2 can result in a systematic behavior as shown in figure 1. Although we have free bus bandwidth left, the resulting total bandwidth consumption is only 66%, not 50%+33% = 83% and P_2 is effectively slowed down by a factor of 1.5

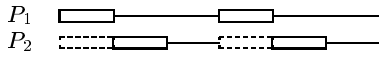

Figure 1: An illustration of how combining two processors (P_1 and P_2), each with memory bandwidth requirements (33% and 50%), does not necessarily produce expected bandwidth consumption.

We have to conclude:

- ***Bandwidth consumption does not necessarily add properly.*** Given n bandwidth requirements b_i, we can not calculate the the resulting global effective bandwidth consumption. Also, a worse conclusion is that we can *not decide whether each processor effectively gets its required bandwidth.* ***Admission control solely based on bandwidth specifications seems impossible.***

A further surprising result is that we might see bus bottlenecks although there is still free bandwidth available, e.g. 33% in the counter-example.

4 Burst-Based Scheduling

4.1 Insisting On Bursts

We have learned that access patterns are at least as important as bandwidth. However, we do not see a realistic possibility to specify, handle, and enforce access patterns.

> Therefore, we propose that all real-time applications access memory only through bursts, i.e. through dense sequences of memory transactions.

All requirements are specified only in relation to bursts. A burst requirement B_i means that processor i wants to be able to execute *one* burst of B_i memory transactions in a Δ-window.

Nicely, overlayed, multiple bursts do not not generate gaps between memory transactions. The memory bus is 100% utilized as long as there is at least one burst. Therefore, the problems of our counter-example disappear. Admission control is simple: all bursts can be satisfied if $\sum B_i < \Delta/c$.

1. *Can we implement all real-time applications such that they access memory only in bursts?*
   Given large L2 caches and assumed a working cache-scheduling policy, such behavior should be possible. The basic idea is to load/store L2 pages in bursts. (However, see also Section 6 which describes how to handle non-bursty programs.)

2. *Can we enforce bursty behavior of real-time applications?*
   Yes. As soon as a processor detects a bus time $t_i < \delta$ for some interval δ, it decides that the burst is over and changes the application's state to low until the end of the Δ-window. Any further memory transactions are low priority and have no guarantees.

3. *Can non-bursty memory transactions of low-state processors violate the full-utilization properties of bursts?*
   No, since they cannot generate additional gaps on the memory bus.

4. *How can we schedule the memory bus such that the admitted bursts are guaranteed and simultaneously as much as possible of the remaining memory-bus bandwidth can be used by low-state processors?*
   See following algorithm.

4.2 A Safe Scheduling Framework

High-state processors are either between two bursts or within a burst. When the first memory access after a non-memory phase occurs a new burst starts and the corresponding Δ-window begins.

> M_i is the number of memory requests executed by processor i from the beginning of its current burst up to now. Outside a burst, M_i is 0.
>
> T_i is the time from the beginning of the current burst on processor i until now. Outside a burst, T_i is 0.
>
> At most $P_i = B_i - M_i$ memory transactions are still outstanding for the current burst.
>
> $\Delta - T_i$ is the available time for executing the outstanding P_i memory transactions.

When a burst starts, an average rate of B_i/Δ memory transactions over the future Δ-window would be sufficient so that the processor gets its required burst of B_i in a Δ-window. Once the burst has started, this changes. In general, processor i is in good shape as long as it will get at least rate

$$r_i = \frac{P_i}{\Delta - T_i} \quad .$$

This can easily be ensured by allocating only free memory bandwidth, i.e. at most $1/c - \sum r_i$ to all low-state processors together. Such strategies, we call *hard*. Unfortunately, hard strategies are far too restrictive to achieve good utilization.

We are looking for more deliberate scheduling strategies that, nevertheless, keep the burst guarantees valid. The key idea is

> We are completely free to distribute memory transactions over the next period of length d if a hard strategy after the period of d could keep all guarantees valid.

Assuming that no high-state processor gets memory bandwidth in the next interval and that all $\Delta - T_i > d$, we have to allocate a memory rate of

$$r_i(d) = \frac{P_i}{\Delta - T_i - d}$$

to high-state processor i. This is possible if $\sum r_i(d) < 1/c$.

> The *deliberate scheduling distance* D is given by the maximum d for which all $r_i(d)$ are defined and $\sum r_i(d) < 1/c$ holds.

Within the deliberate scheduling distance, we can apply arbitrary memory scheduling policies without compromising admitted guarantees. Such *soft* policies should be likely to schedule the system such that all active bursts terminate in time. However, this *likelyhood* makes a big difference when compared to hard policies. Since soft policies *may* fail they can usually better adapt to the effective system behavior and achieve much better utilization.

The deliberate scheduling distance tells us how "risky" a policy we can currently use. Furthermore, D is a measure for success of the so far applied soft policies: if D increases the current soft policy is helpful; a decreasing D signals the danger that hard policies might become necessary soon.

Guaranteeing close to 100% of the bandwidth to be available for high-state bursts is a bad idea. If no burst is busy, all r_i would be B_i/Δ, i.e. $\sum p_i$ would be 1 so that we have no freedom at all, $D = 0$. Consequently, we would have to run a hard policy and give zero bandwidth to low-state processors. (Surprisingly, the situation gets better as soon as a burst starts: D will grow.)

Lower reservations, say only 80%, give us a deliberate scheduling distance of $D = 20\%\Delta$ so that we can use soft policies as long as no burst is active. Once a burst starts, D can shrink or grow.

4.3 Soft Policies

The simplest soft policy is not to schedule at all. It might be a good choice for a large D.

Another soft policy restricts the memory transaction rate of the low-state processor(s) that perform the most memory transactions in the recent past. The restriction typically depends on the current deliberate scheduling distance, the totally guaranteed burst bandwidths, the number of currently active bursts, and the effective total bandwidth in the recent past.

More restrictive and more sophisticated policies allocate memory transaction rates based on reservations and current use. Dependent on the current deliberate scheduling distance, they overbook to achieve good memory-bus utilization.

5 Implementation Issues

5.1 Burst Detection

To detect the beginning of a burst, we can allocate 0 memory transactions to the processor. The first memory request then raises an interrupt signaling the begin of a burst.

A perhaps better suited variant uses a system call for signaling the burst. The processor can then execute memory transactions before the guaranteed burst starts, however it operates in low state until the system call happens. In practice, this solution is beneficial because (1) it permits precise application-controlled burst synchronization, and (2) it automatically handles mixes of real-time and non-real-time applications on the same processor. Non-real-time applications operate as usual and thus run in low-state. Real-time applications will explicitly start bursts and thus run in high state.

Once a burst starts, the processor sets its own T_i and M_i to 0 (performance counters) and allocates B_i memory transactions for itself. Furthermore it sets an alarm to *now*+Δ. Exhaustion of the allocated memory transaction quantum as well as the Δ alarm obviously signal a burst end.

5.2 Burst Gaps

However, it is much harder to detect a burst end before Δ is over and before the memory transaction quantum is exhausted. To get rid of access pattern problems, we defined that a burst ends as soon as a time gap in memory requests occurs. Remember that this is a serious problem. Without burst-gap detection, a non-bursty

program could slow down other processors so that they do not get their guaranteed bandwidth.

A simple burst-gap detector could periodically check the processor's bus-busy time. As soon as it grows slower than processor clock, a gap in the memory-request stream is detected. This algorithm requires some correcting terms in our formulas. For example, the memory-request gaps that are due to the algorithm itself have to be discarded. Since gap detection is done periodically, it can be up to one period late. Resulting effects have to be taken into consideration in our formulas. High accuracy requires short check intervals. The consequent interrupt-handling costs might be unacceptable.

Fortunately, as long as the deliberate scheduling distance D is large enough, there is no need to detect burst gaps quickly. It is sufficient to check for gaps whenever we adapt the current soft policy, or calculate a new D value, etc.

However, once a hard policy is effective, gaps must be detected quickly, preferrably with low software overhead, perhaps through hardware support. A bus-unit-idle performace counter would do the job.

5.3 Inter-Processor Communication

Each processor has to publish its P_i and T_i values periodically. Adding some correcting terms in our formulas —basically calculating a lower bound for D provided the maximum time difference is τ— permits us to do this *unsynchronized*, i.e. inexpensively. Policy schedule decisions can be distributed in a similar way.

6 Getting Rid of the Burst Restriction

6.1 Multiple Bursts

The restriction to one burst per Δ-window made our reasoning simpler but it is not application friendly. First, applications might need multiple bursts per window. Second, a burst might be unintended, resulting in early interruption. Both problems are solved if we permit a limited number of gaps per burst. In addition to the burst length B_i, an application can also specify the maximum number of potential gaps G_i.

However, to ensure that an application has not more than G_i gaps in its memory transaction stream, we need additional hardware support. A performance counter that counts rising or falling edges of the bus-unit busy line would suffice. Such a counter counts the events when either the bus unit switches from idle to busy or from busy to idle, i.e. gaps in the memory transaction stream.

6.2 Intermittent Bursts

An even more powerful mechanism is a counter counting both memory transactions and the bus unit's idle-busy transitions. As illustrated by our counter-example (see Figure 1), each gap could impose additional costs of at most c, the cost of one memory transaction. Such a *memory+gap* counter therefore measures directly an upper bound of the memory costs.

Assume we use this *memory+gap* bandwidth measure $\dot{B}_i = B_i + G_i$ per Δ-window, not the pure memory bandwidth B_i we used so far. Instead requirements of 33% and 50%, we get then requirements of 66% and 100% for P_1 and P_2 in our counter-example that obviously cannot be combined without performance loss.

To get a general model, we systematically include the gaps in requirements and measurements, i.e. we replace the requirements B_i by $\dot{B}_i = B_i + G_i$ and the measured M_i by $\dot{M}_i = M_i + H_i$ where H_i denotes the effective number of gaps in processor i's memory transaction stream. $\dot{B}_i$ requests are then specified by the applications, $\dot{M}_i$ values are measured by the *memory+gap* performance counter.

The modified formulas of Section 4.2 again give us a safe scheduling framework that can now as well be applied to non-bursty real-time applications. Nevertheless, it seems useful to keep the burst abstraction in so far that bursts have definite start and stop points. Otherwise, non-exhaustive bursts (bursts with less memory+gap requests in a Δ-window than reserved, i.e. $\dot{M}_i < \dot{B}_i$) would unnecessarily over-reserve bandwidth after the burst has finished and, even worse, *before the burst has started*. In contrast to the original burst-based method, memory-inactive processors would therefore seriously restrict our scheduling freedom.

So we extend we extend our burst concept to *intermittent burst* that can include gaps. The non-trivial problem of intermittent bursts is how to decide when a non-exhaustive intermittent burst is over. We experiment with two ideas: (1) Burst ends are signaled through a system call similer to burst begins; (2) a burst ends when a gap of specifiable length γ_i is detected.

6.3 (Non-)bursty Application Revisited

Ensuring that an application is 100% bursty in its memory accesses requires strict control of the L2 cache. Intermittent bursts enable us to weaken this condition substantially. For example, an application can now combine requirements such as "needs a 95% L2 cache-hit rate" and "needs intermittent burst rate ($\dot{M}_i/\Delta$) of 20%". The second requirement ensures that cache misses are handled fast enough.

7 Conclusion

We have presented a framework that should permit first experiments with hard and soft real-time scheduling on multiprocessors. In particular, the framework supports mixes of real-time and non-real-time applications on SMPs that share a single memory.

The presented methods can also be applied to systems that use asymmetric processors, e.g. one general-purpose CPU and two DMA processors, such as a disk and a network controller. In the above example, the DMA processors only have to be programmable and incorporate the performance-counter mechanism.