# Process Cruise Control
## Throttling Memory Access in a Soft Real-Time Environment


Frank Bellosa
bellosa@informatik.uni-erlangen.de

IMMD IV - Department of Computer Science (Operating Systems)
University of Erlangen-Nürnberg


## Abstract


The advances in memory technology concerning performance have not been able to keep pace with those in processor technology. Processors clocked with hundreds of megahertz exceed the speed of affordable memory by factors. Caches can decouple the speed of the processing unit from the speed of the memory system if applications show a high locality of reference. Unfortunately, operations on data streams – frequently found in soft real-time multimedia applications – do not show this benign behavior. Thus, applications working on data streams rely heavily upon a guaranteed memory bandwidth to meet specific timing requirements. In multiprocessor systems the available memory bandwidth is shared by all processing units and DMA devices. Consequently, the processing units can interfere and slow each other down when accessing memory. Up to now, this effect, called *memory preemption*, is not covered by real-time operating systems using timer and I/O related information.
Our novel approach to resource management is based on knowledge derived from counters in the memory subsystem. We demonstrate that the use of information related to cache- and main-memory access opens new dimensions of resource management in shared-memory architectures. The introduction of memory-bandwidth guarantees adds a further resource to capacity-reservation models and therefore enhances the quality of service.

Soft real-time processes can request memory bandwidth guarantees. Processes without guarantees are throttled, so that they do not withhold valuable main memory bandwidth from real-time applications. To dynamically slow down processes that exceed a certain number of main memory operations per time frame, the TLB-miss handler executes additional NOP instructions. Thus, the TLB-fill will be delayed, fewer memory pages can be touched and fewer cache misses per time frame will occur. This new mechanism, called *Process Cruise Control,* maintains the execution speed of soft real-time applications in a multiprocessor environment. Applications of other scheduling classes (e.g., Time-Sharing), which operate with low memory demands, run at full speed, whereas applications with high memory demands will be throttled in their speed of execution and executed with lower priority.

Measurements conducted on a prototype implementation using the Solaris operating system clearly demonstrate the benefit of the memory throttle for a video conferencing application running in a multiprogrammed multiprocessor environment.

# 1 Motivation

In the last two decades memory technology has advanced concerning the volume of data which can be stored. But the latency to access data in affordable memory has been stagnating. Today, processors with wide data paths (64 bit) which are clocked with hundreds of megahertz place extreme stress on the memory system. The extent to which multiple interleaved memory banks can narrow the gap between the increasing memory demands of advanced processor architectures and the bandwidth of the main memory is limited. Multiple levels of caches are only useful, if applications show a high locality of reference. Operations on data streams e.g., audio or video-image processing, draw no benefit from caches. The execution speed of theses operations is closely coupled to the latency and bandwidth of the main memory.
Shared-memory multiprocessor architectures are frequently used in the field of multimedia, because they offer high processing power, low-latency inter-process communication and good behavior in the presence of a high interrupt load from multiple I/O devices. But it is a characteristic of these computer architectures that the available bandwidth of the main memory is shared among the processing units and DMA devices. Normally, the summarized peak number of memory requests that all processors can issue within a given time frame exceeds the number of requests the memory can satisfy. Consequently processors can stall each other by demanding a high volume of data from memory. To demonstrate this effect, we have measured the copy-rate a CPU can achieve if 1 – 4 copies of the STREAMS benchmark run on a SUN E3000 server in parallel (see TAB. 1.1.).

| Number of CPUs | 2 Memory Banks | 4 Memory Banks |
|---|---|---|
| 1 | 197 MB/s total ; 197 MB/s per CPU | 197 MB/s total ; 197 MB/s per CPU |
| 2 | 322 MB/s total ; 166 MB/s per CPU | 362 MB/s total ; 181 MB/s per CPU |
| 3 | 382 MB/s total ; 127 MB/s per CPU | 485 MB/s total ; 161 MB/s per CPU |
| 4 | 406 MB/s total ; 101 MB/s per CPU | 582 MB/s total ; 145 MB/s per CPU |

TAB. 1.1. Copy rate of the STREAMS-Benchmark (stream_d.c version) running on a SUN E3000 (4 CPUs clocked at 167 MHz) under Solaris 2.5.1

The copy-rate decreases to 50% of the value in the single CPU case if only 2 memory banks are available. If all memory banks are fully equipped, the copy-rate decreases to 75%. CPUs clocked with a higher frequency can issue more load/store instruction and therefore increase this effect.

The STREMAS benchmark has demonstrated that the execution speed of an application with high memory demands can depend on the memory demands of other applications running on other CPUs. This effect, which we call *memory preemption*, is especially negative for real-time applications, which rely on a guaranteed execution environment. High scheduling priorities, locked memory pages and preferred I/O handling cannot prevent this effect. We have clearly demonstrated the consequences of memory preemption on a SUN E3000 server with 4 CPUs, where the execution speed of a video-conferencing application running on a dedicated CPU drops from 25 to 20 frames per second if the remaining CPUs demand a lot of megabytes per second (see FIG. 1.1.).

In conclusion, we have to realize that the available bandwidth of the memory is a valuable resource, which has to be managed by the operating system to meet the needs of real-time applications with high memory demands.

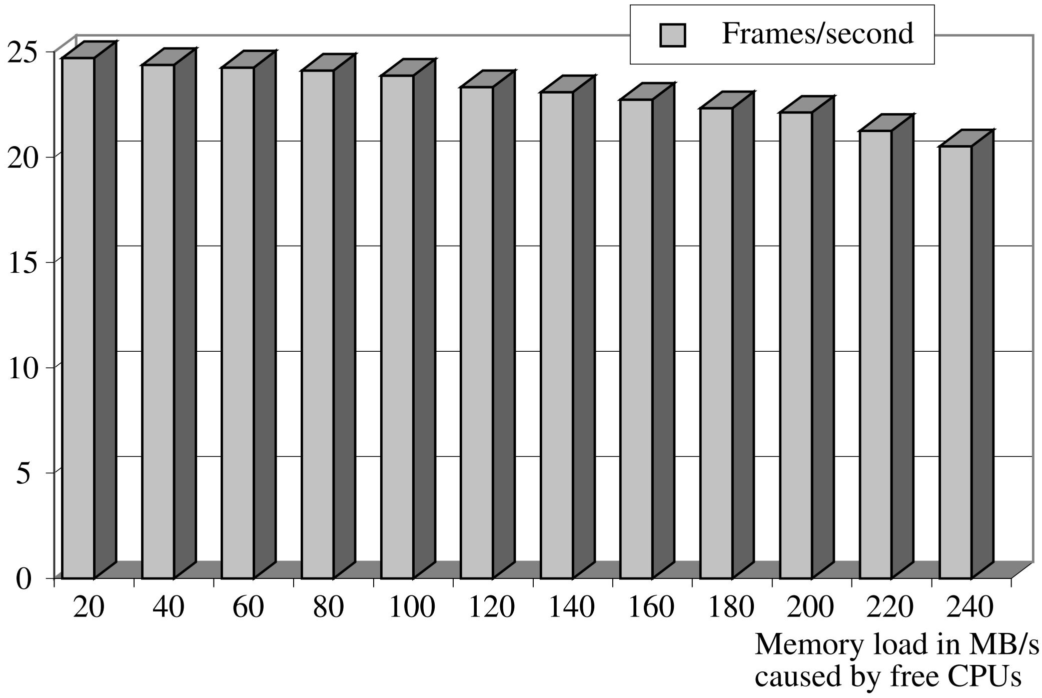


FIG. 1.1. Reduced execution speed caused by memory preemption

Management needs information. In the next sections of this paper we demonstrate how to collect data from hardware devices, how operating system policies can process this information and influence novel mechanisms in the memory management system to control the speed of execution. Finally we present first measurements proving the proposed concept.

## 2 Memory-Bandwidth Reservation and Throttling Model

To manage memory-bandwidth, the resource with which we are concerned, we have to design a resource capacity reservation model which is adapted to the special needs of the memory subsystem. As in the general model described in [LRM96], our memory-bandwidth reservation scheme consists of several components:

- Usage measurement compiles the data base of policies, that have to decide which threads are allowed to execute and how intensively they have to be throttled. Therefore the number of memory requests in a specified time interval has to be counted and assigned to the responsible entity (e.g., the number of cache misses a thread experienced during the last time-slice).
- A programming interface allows applications to request a specified reasonable number of main memory requests per specified interval (such as 90,000 cache misses every 100 ms)

- An admission control policy is used to decide if a request can be accepted and when it must be denied. This policy assumes that the total number of main-memory access operations which can be served within a given interval is known. As the hardware differs from machine to machine, we propose to measure the memory properties during the boot phase of the machine.
- A throttling policy decides, which threads have to be throttled and how much these threads have to be throttled. Information from measurements provides feed-back to continuously recalibrate the degree of throttling.
- A throttling mechanism slows down specific threads on a fine-grained level. Frequent context switches to threads with low memory demands (in extreme cases the idle thread) are not the solution, because context switches imply additional memory load and furthermore, because context switching happens on a coarse-grained level in the range of milliseconds. (You don't repeatedly open and close the faucet to regulate the throughput of water in the shower, but throttle the flow of water.)

In the next sections, we propose concepts and implementation details to include memory bandwidth as an additional resource in a capacity reservation model.

## 2.1 Measurement of Memory Access

System designers have always observed the behavior of computer systems, using hardware monitors to detect bottlenecks and to check the performance gain of architectural improvements. Because of increased clock speed of up to several hundred megahertz, external hardware monitors became more and more expensive for electrical reasons. Internal event monitors, which count events at full clock rate, are an affordable alternative to external monitoring hardware. Thus, monitoring hardware is embedded in advanced computer architectures to count events occurring inside the processor, the memory system or the I/O-subsystem.
If we look into today's multiprocessor systems we can see counters in some of the regions mentioned above. On the one hand, we can make the best of the available counters and use them in operating system policies and resource reservation schemes; on the other hand, we can influence system designers to enrich computer architectures with counters that are easy to access and fast to read.

Memory access information is the totality of countable events related to memory operations. Examples are load/store-operations, cache misses, cache invalidations, issued main-memory requests or processor stall cycles due to memory requests. A countable event is only useful if it can be assigned to the object responsible for the event. We show the locations in shared-memory architectures where memory access information can be gathered by event counters, such that this information is usable in resource capacity reservation models.

### 2.1.1 Locations to Collect Memory Access Information

There are four potential regions in a shared-memory architecture suitable for the placement of memory-related event counters:

- The processor:
  Counters inside the CPU can register issued load/store operations. If the first level caches reside inside the CPU, the effects of cache-access operations (e.g., data- and instruction-cache hits/misses, CPU stall cycles due to cache misses, etc.) can be registered inside the processor and assigned to the process currently running on this CPU.
- The cache controller for external caches:
  Cache misses, cache invalidations and stall cycles due to cache misses can be registered inside the cache controller. If the external cache is dedicated to a single CPU, the information can be assigned to the currently running process. Architectures with shared caches do not offer this feature.
- The interface between the processor-cache module and the interconnection network:
  Cache misses initiated by a CPU imply main memory access but not vice versa. Memory regions marked as uncachable (e.g., DMA buffers) do not become encached so they cause no cache events when being accessed. Furthermore, some CPUs (e.g., Pentium MMX [Intel 96], UltraSPARC [SUN 95]) offer special load/store operations that bypass the cache to prevent cache corruption by operations working on data streams (e.g., MPEG operations). These load/store operations initiate main-memory operations without interfering with the cache content.
  Counters inside the interconnect interface (bus- or crossbar-connector) can register all main-memory related transfers. The information can be assigned to the currently running process only if the interconnect interface is not shared by multiple CPUs.
- The interconnect interface of memory banks:
  Each memory bank can only service a limited number of requests in a certain time frame. If the number of serviced requests can be registered in counters, the load of the memory subsystem is known.
- The interconnect interface of I/O boards:
  DMA devices located on I/O boards initiate data transfers between the main memory and external devices. These transfers are usually anonymous and cannot be assigned to a single thread of control, but they contribute to the memory load and should therefore influence the memory-bandwidth reservation model.

In the next subsection, we propose a resource measurement approach to register main-memory access by evaluating the information derived from cache miss counters. We demonstrate how the available event counters which are embedded in the SUN Enterprise X000 Server Architecture can be used to collect memory access information by the Solaris operating system. A detailed description of the sampling technique can be found in [Bell97]. Furthermore we propose architectural improvements which could overcome the limits imposed by the current hardware architecture.

### 2.1.2 SUN Enterprise X000 Server Architecture

The SUN Enterprise X000 Server Architecture [SUN 96] is the hardware platform for our memory-bandwidth reservation model using memory access information. Each Ultra-I CPU [SUN 95] includes two counters, which can register two events out of a set of more than 22 event-types. The MMU and the external cache controller is on-chip, so that external cache events like cache hits or cache references are countable events as well as issued instructions and clock ticks (see figure FIG. 2.1.).

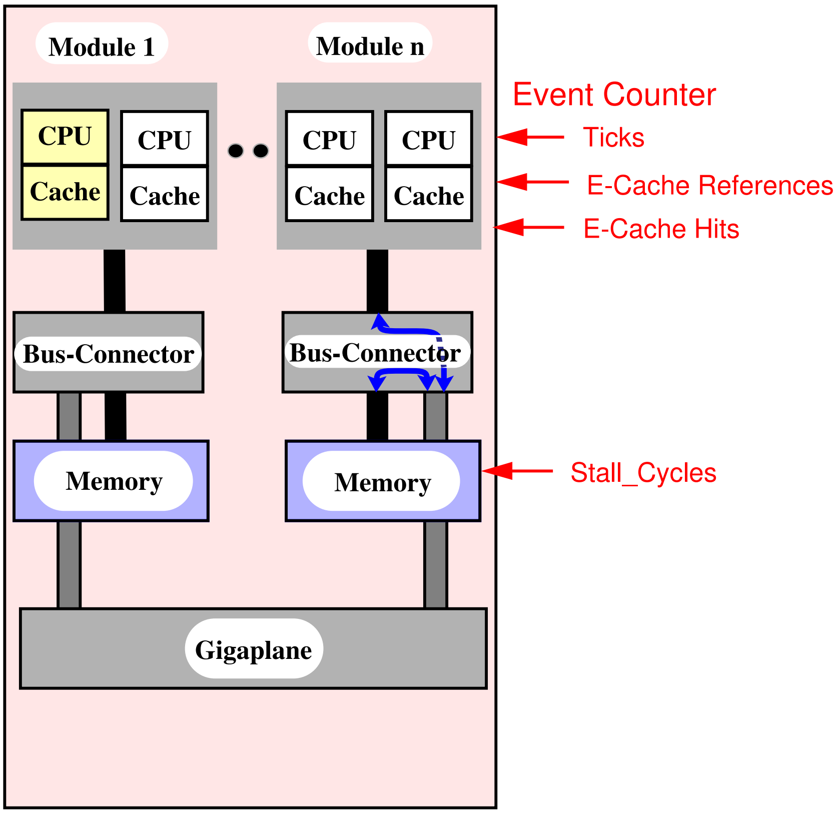


FIG. 2.1. Ultra Enterprise X000 Architecture

The bus-connector is implemented as a switch (**U**ltra **P**ort **A**rchitecture switch), connecting a pair of CPUs and two memory banks with the bus system (GigaPlane). Two configurable counters can register events such as issued addresses, issued data packets or memory stall cycles of each memory bank. As the bus-connector is shared by two CPUs, the counter values cannot be assigned to a specific CPU nor to a specific process.

Our approach to bandwidth measurement uses the information derived from both CPU event counters. The first counter notifies references to the external cache, the second one registers hits in the external cache. From both counters we can deduce the number of cache misses a CPU is experiencing. As a side-effect, block-transfers of 64 bytes issued by the multimedia **V**isual **I**nstruction **S**et (VIS) increase the cache-reference count although these transfers bypass the cache. As each cache-line which has to be transferred as the consequence of a cache miss, has a length of 64 bytes as well, counting cache-references and cache-hits represents the complete number of memory transfers issued by a specific kernel thread.

To assign counter values to kernel threads, both counter values as well as the counter control register are saved and restored in the switch routine swtch(). To handle the overflow of the 32 bit counters, we introduce virtual 64 bit counters, which are updated during the context switch (swtch()) and during the return from a trap occurring in user-mode (trap_ret()). The minimum sampling frequency is the frequency of the timer interrupt.
To calculate the number of memory-transfers per time-frame for which a thread is responsible, we additionally save the CPU TICK-register which counts the CPU clock cycles and update a virtual clock-counter. The sampled data is stored in a data structure local to the currently running thread (kthread_t). Furthermore we add up the counter-values for each CPU (stored in cpu_t).
The sampled data forms the information base of the time sharing scheduler (TS), which calculates the memory bandwidth each thread running under this scheduling class is consuming expressed in megabytes per second and assuming, that a cache miss involves a transfer of 64 bytes.
It was our intention to access the sampled data from user-space without any system call. Our approach is an enhancement of the /proc-filesystem, which provides kernel-virtual addresses of those regions where sampling data is stored. After mapping those regions from /dev/kmem into the address-space of a profiling-daemon, there is no need for system intervention to access sampling data. Now, an unmodified application can be observed from a dedicated CPU without any influence on the application's execution.

As the UPA-switch is shared by two CPUs, we cannot use the two embedded counters to register read and write packets issued to the bus-system, as the collected data cannot be assigned do a single entity responsible for the bus transaction. That is why we configured the counters in the UPA-switch to register memory stall cycles of both memory-banks. Now, the operating system could detect an overload of the memory subsystem. The detection of overload and its avoidance is the topic of further improvements we plan to incorporate in the Solaris operating system.
Up to now, our approach neglects DMA transfers. The counters inside the I/O boards register DMA operations, but we do not use this information in our current implementation, because we focus on memory-related information that can be assigned to kernel threads. DMA transfers will be under investigation in further research efforts.

From the point of view of operating system design, the following architectural features would facilitate the design of memory-conscious operating system policies:

- 64 bit counters to register cache-references, cache-hits, instructions and clock-ticks.
- 64 bit counters in all ports of the bus-connector to detect memory transfers issued by CPUs as well as DMA devices. All memory requests related to a CPU could be assigned to a kernel thread, whereas the memory requests issued by DMA devices could flow into the system-wide calculation of the current memory load.
  These counters have to be rapidly readable (< 10 clock cycles) to reduce the overhead of sampling to a minimum. Therefore multiport registers are necessary to avoid stalling of the CPU after a read request.

In this section we have proposed an approach to measure the memory access of individual threads in a shared-memory architecture. In addition to neglecting DMA requests, all memory requests can be assigned to a thread and can therefore be used in a resource capacity reservation model focusing on memory bandwidth.

## 2.2 Programming Interface

Processes can request a reasonable share of the available memory-bandwidth from the operating system. An additional system-call or an enhanced system-call has to be provided by the operating system.

In our prototype implementation, we enhanced the priocontrol system-call of Unix System V Release 4 and the dependent utility applications. In addition to its ability to modify the priorities of the time sharing scheduling class (TS), our priocontrol can request a certain memory bandwidth - expressed in megabytes per second - from the scheduler. If the call returns without error, the request is accepted. The granted bandwidth values can be retrieved by a priocontrol call as well.

## 2.3 Throttling Mechanism

The aim of our efforts is to reduce the number of memory-access operations of those processing units, where non-critical applications without guarantees are running. There are three possible ways to reach this goal:

(1) Frequent switching between threads of different memory-access characteristics can result in a specified number of memory-access operations in a defined time frame. The problem is to elect a group of runnable threads which run by turns in very short time-slices. If this policy fails and the memory-load exceeds the limit imposed by the throttling policy, the idle thread has to be elected for execution to nearly stop memory-access.
A severe disadvantage of this throttling mechanism is the frequent occurrence of context switches which imply additional memory-access operations. Context switches happen several hundred times per second. The time frame a resource can be guaranteed is therefore in the range of tenths of a second. This value is too long to guarantee, for example, the continuous processing of a video stream with 25 frames per second.

(2) Slowing down a process running on a CPU is possible by inserting additional operations into the thread of control.

- The most fine-grained approach would be the insertion of No-Operation (NOP) instructions after each load/store-operation so that some CPU cycles are lost after each memory-access operation. Unfortunately the dynamic modification of code under execution is a non-trivial task, as the code segments have to be modified and pointers have to be relocated.

- Some CPU architectures (e.g., Alpha processors [DEC95]) provide event counters which generate an interrupt whenever the counter overflows. If a thread should be throttled, the event counter is tuned in such a way that it generates an interrupt after a determined number of cache misses. The interrupt handler executes so many NOP instructions that the number of memory-access operations cannot go beyond a specified limit. As the number of interrupts happening because of counter overflows depends on the number of cache misses, the share of available bandwidth a thread holds during execution can be precisely adjusted. The interrupts can happen several thousand times per second. Therefore the time-frame within which the memory-bandwidth can be throttled is in the range of several milliseconds. This value is sufficient to guarantee the continuous processing of most soft real-time applications.
- Cache misses can occur for four reasons: The requested data has never been accessed before (compulsory misses); the requested data has been accesses before, but the CPUs working set size exceeds the cache size (capacity misses); the requested data had been in the cache, but was displaced by an intervening reference to another address (conflict miss); the requested data had been in the cache, but was invalidated by an other CPU (invalidate misses).
  The size of the address space covered by the translation-lookaside-buffer (TLB) is normally in the same range as the size of the external cache. For example, the 64 entries of the SUN Ultra-1 CPU cover 512 kBytes (Solaris 2.5.1 uses 8 KBytes page size) and the size of the external caches ranges from 256 kbytes to 1 MegaBytes. Compulsory misses, capacity misses and conflict misses frequently coincide with TLB misses. Therefore the TLB-miss handler executed with a high frequency in the presence of a high number of cache misses can be used to execute NOP operations or idle loops. The number of NOP operations or idle loops can be adjusted by the throttling policy so that a specified number of cache misses will not be exceeded. By delaying the TLB-miss handler, fewer pages per time frame can be touched and the number of compulsory, capacity and conflict misses is reduced. Nevertheless the thread remains running on the CPU and becomes CPU intensive. All such threads are given lower execution priority. Threads causing few cache misses, such that they do not exceed the imposed limits, run the TLB-miss handler without any delay.
- Hardware support for throttling is the neatest solution. A special *throttle register* indicates how many NOPs have to be inserted into the stream of instructions after each load/store instruction.The instruction fetch unit of the CPU is responsible for this task. The throttle register is part of the CPU context, and will be handled like other processor status registers. As the throttle register can be implemented with minimal hardware requirements and without any compatibility problems, we recommend adding this architectural feature to processors frequently used in real-time processing.

As the Ultra-I processor architecture used for our implementation offers neither event counters generating interrupts nor any architectural support in the form of a throttle register, we have chosen the approach which throttles the memory-access by inserting an idle-loop into the TLB-miss handler. As we have no virtual memory available inside the TLB-miss handler, we have to use physical memory or a register to fetch the idle-loop count. In our prototype implementation, we have misused a register which is normally used for asynchronous system faults like ECC-errors of the memory. This register is saved/restored at each context switch and updated after each trap in user-mode.

To demonstrate the effect of throttling, we have measured the copy-rate of the STREAMS benchmark depending on the loop-count of the idle-loop inside the TLB-miss handler.

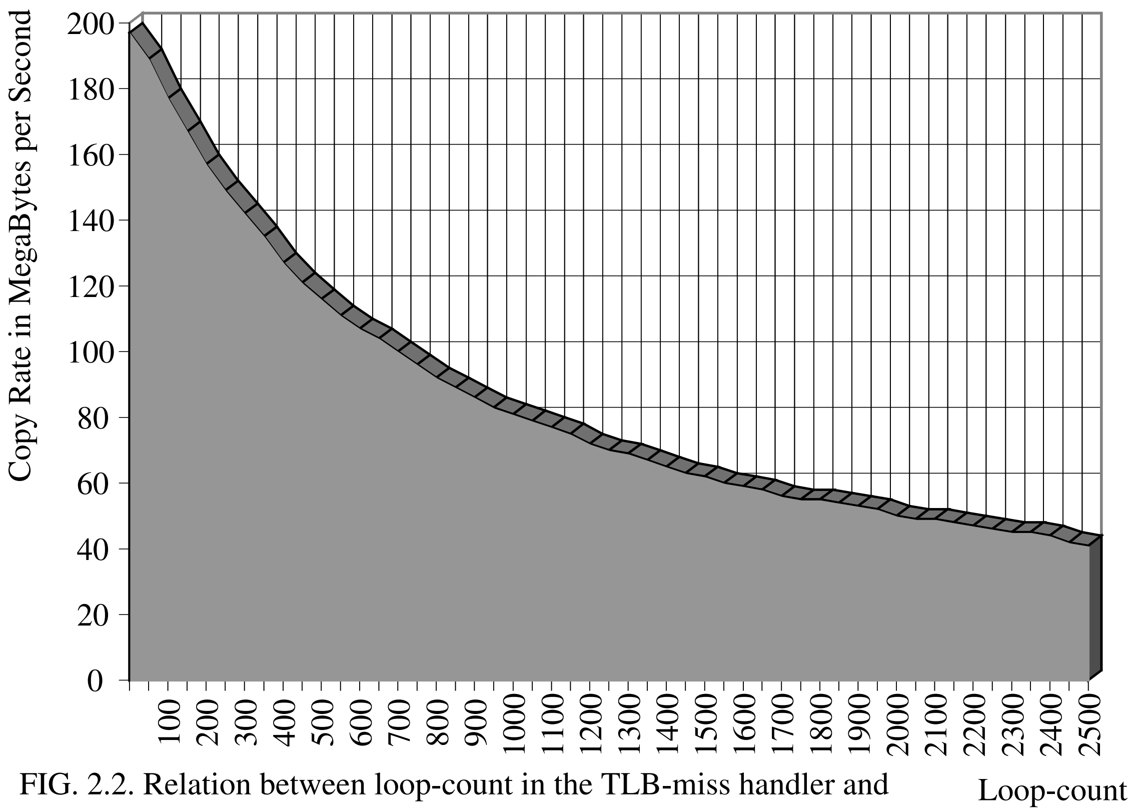


FIG. 2.2. Relation between loop-count in the TLB-miss handler and the copy-rate of the STREAMS benchmark

We can see that the memory-bandwidth a process is using can be exactly tuned by the idle-count in the TLB-miss handler.

## 2.4 Admission Control and Throttling Policy

The admission control policy has to decide whether a requested memory bandwidth can be guaranteed. This decision has to be made in cooperation with the throttling policy, which has to decide which threads to throttle and to determine the degree of throttling.

The decisions taken by our policy are based on the following information:

- Available memory bandwidth of the multiprocessor system
- Number of processors
- Memory bandwidth which is already guaranteed to an application

- Cache-miss counter of currently running threads
- Idle-loop counter used in the TLB-miss handler of currently running threads

### 2.4.1 Implementation Details

We had to find a simple model which precisely describes the properties of the memory hardware. Our quite conservative heuristic approach to bandwidth reservation obeys the following rules:

- The maximum bandwidth which can be requested is 90% of the copy-rate the STREAMS-benchmark measures if all processing units execute the benchmark in parallel.
  $MaxRequest = 0.9 \cdot StreamCopyRate$
  Processes requesting the limit in memory bandwidth are very sensitive to any memory operations issued by other CPUs. Because of this observation, we added this 90% rule.
- The free bandwidth can be used by threads without guarantees. Before any request is accepted, the free bandwidth is set to the total copy-rate of the STREAMS-benchmark.The free bandwidth is calculated according to the following formula:
  $$FreeBandwidth' = (FreeBandwidth - RequBandwidth) \cdot \left(1 - \frac{RequBandwidth}{StreamCopyRate}\right)$$
  This rule is motivated by the observation that the application which wants to request some memory needs a certain fraction $\frac{RequBandwidth}{StreamCopyRate}$ of the available memory cycles. Other threads are allowed to use the remaining fraction $1 - \frac{RequBandwidth}{StreamCopyRate}$ of the remaining bandwidth $(FreeBandwidth - RequBandwidth)$
- The free bandwidth has to be greater than zero.
  $FreeBandwidth > 0$
  If the free bandwidth would become negative as a result of a request, the request is denied.
- Each thread with a guarantee is assigned its own processor
  $NumGuarantees + 1 < NumCPUs$; $FreeCPUs = NumCPUs - NumGurantees$
- The free bandwidth is equally shared among the free processors
  $$BandwithLimit = \frac{FreeBandwidth}{FreeCPUs}$$

Based on the measurements of the STREAMS-benchmark with 2, 3 and 4 CPUs and 2 memory banks (see TAB. 1.1.) we have plotted the bandwidth limits depending on the number of CPUs, which has to be chosen in the presence of one thread requesting a certain amount or bandwidth (see FIG. 2.3.). For example, a request for 60 megabytes per second limits the bandwidth of all other threads to 35 megabytes per second in a machine with 4 CPUs, to 56 megabytes per second when 3 CPUs are available and to 87 megabytes per second in a dual-processor configuration.

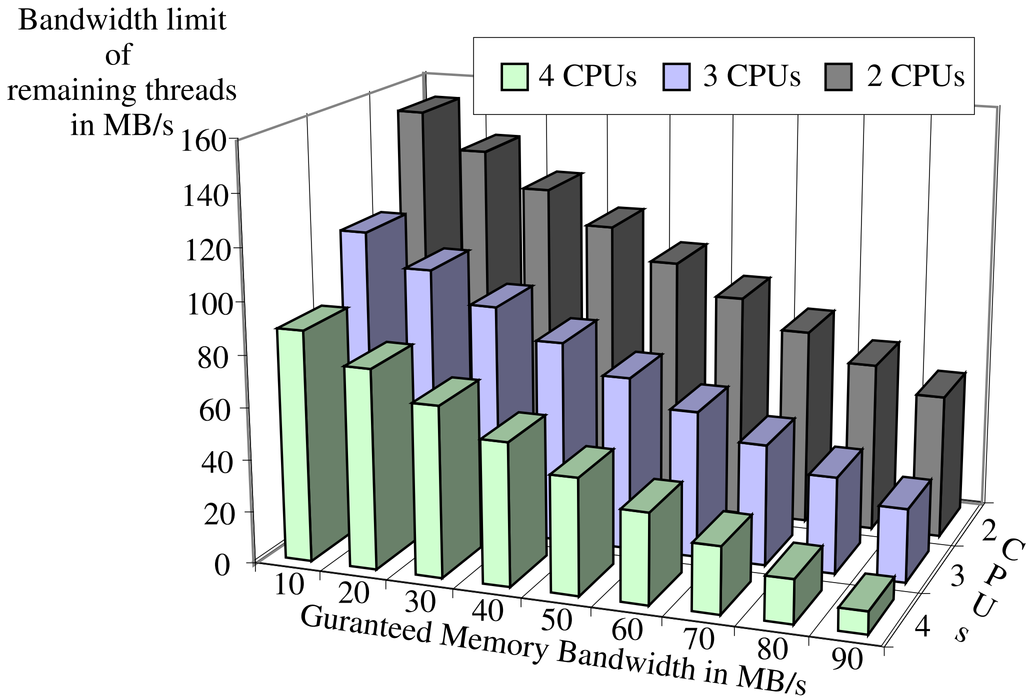


FIG. 2.3. Memory Bandwidth Capacity Reservation

It is clear that these rules of thumb based on simple assumptions impose a lot of restrictions in the presence of a real-time application requesting bandwidth reservations. But our observations did not allow less restrictive rules. Further research will focus on better rules in order to utilize the available bandwidth while simultaneously granting an execution free of memory stalls.

In our prototype implementation we added the admission control policy to the priocontrol functions of the time-sharing scheduler. The tunables like number of processors and available bandwidth have to be determined in the boot procedure.

The throttling policy is added to the tick-processing of the time-sharing scheduler.
If a thread exceeds the imposed limit in memory bandwidth, its idle-loop count for the TLB-miss handler is increased. The update of the CPU register holding the loop-count is done at the next return from a trap. Note that the tick-processing is done by an interrupt thread, which might run on another CPU. Therefore the tick-processing is not allowed to set the CPU-register used in the TLB handler.
After a few ticks, the memory bandwidth is adjusted to the imposed limit. To smooth bursts in the memory access, we added a smoothing function in the calculation of the used bandwidth.

# 3 Proof of Concept

To demonstrate the benefit of process cruise control, we ran the video-image application used in Section 1.1. to demonstrate the negative effect of memory preemption. As this application has a demand for 61 megabytes per second, 108 megabytes per second of free bandwidth can be used by the remaining 3 CPUs. This implies a bandwidth limit of 36 megabytes per second for all remaining processes. Our measurement (see FIG. 3.1.) clearly demonstrated that a specified execution rate of 24-25 frames per second can be maintained even if processes are running on other CPUs which try to reach a memory load of more than 200 megabytes per second.

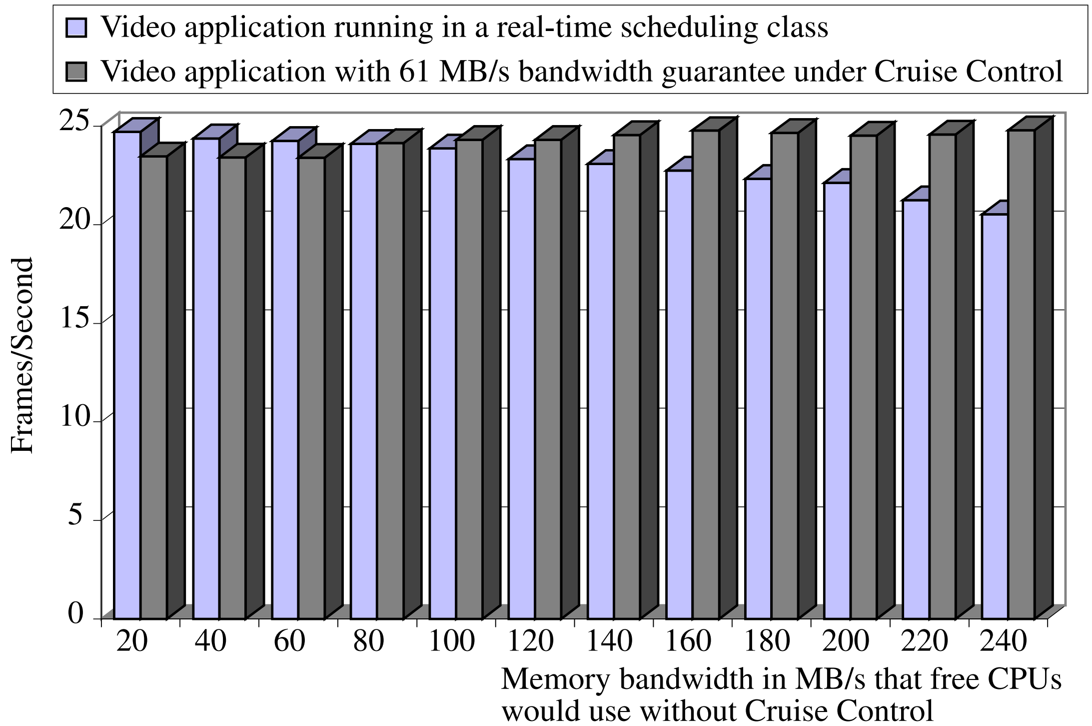


FIG. 3.1. Soft-realtime video-image processing with Process Cruise Control

# 4 Conclusion

Operating systems need information to control operation. To avoid the malicious effects of memory preemption, throttling policies have to decide which threads are allowed to run at full speed and which have to throttled. The basis of these decisions is information derived from event counters in the hardware.

We have demonstrated where counters could usefully be placed in a shared-memory architecture, and how counters available in today's architectures can be used. We have to realize that the management of resources like memory bandwidth, which are not physically partitionable, cannot be solved merely by means of scheduling. Novel resource management policies placed between scheduling and memory management have to be developed. Process Cruise Control is the first approach to prove itself with unmodified applications in a production environment.

We have to bear in mind, that computation is more than modifying data. It comprises data modification as well as data movement. Consequently the memory system is a critical component of any high-performance computer system. Scheduling and memory management should now start to respect this development.

## 5 Acknowledgments

I would like to thank the HP/Convex and Sun Microsystems operating system group for providing their software and background in OS development. The ELiTE project in cooperation with HP/Convex and Sun is supported by the Bavarian Consortium for High Performance Scientific Computing (FORTWIHR) and the SUN Technology and Research Excellence Center (SUN-TREC) of the University of Erlangen.

## 6 References

**[Bell96a]** F. Bellosa "The Performance Implications of Locality Information Usage in Shared-Memory Multiprocessors". Journal of Parallel and Distributed Computing, Special Issue on Multithreading for Multiprocessors, Vol. 37 No. 1, Aug. 96

**[Bell96b]** F. Bellosa "Locality-information-based scheduling in shared-memory multiprocessors". In L. Rudolph D. Feitelson, editor, *IPPS'96 Workshop of Job Scheduling Strategies for Parallel Processing*, volume 1162 of *LNCS*, pages 271--289, Honolulu, Hawaii, apr 1996.

**[Bell97]** F. Bellosa "Memory Access - The Third Dimension of Scheduling". Technical Report, Department of Computer Science IV (Operating Systems), TR-I4-97-01, Jan 97

**[DEC95]** "Alpha 21164 Microprocessor Hardware Reference Manual". Digital Equipment Corporation, Maynard Ma, 1995

**[LRM96]** C. Lee, R. Rajkumar, and C. Mercer. "Experiences with processor reservation and dynamic qos in real-time mach". In *Proceedings of Multimedia Japan*, may 1996.

**[Intel 96]** "MMX Technology Developer's Guide". Intel Inc. 1996

**[SUN 95]** "UltraSPARC Programmer Reference Manual, Revision 1.0". SUN Microsystems Inc., 1995

**[SUN 96]** "Ultra Enterprise X000 Server Family: Architecture and Implementation". Technical White Paper, Sun Microsystems Inc., 1995