Three Dimensions of Scheduling

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Frank Bellosa

Erlangen - 1998

Als Dissertation genehmigt von
der Technischen Fakultät der
Friedrich-Alexander Universität Erlangen-Nürnberg

Tag der Einreichung:	22. Juni.1998
Tag der Promotion:	27. November 1998
Dekan:	Prof. Dr. G. Herold
Berichterstatter:	Prof. Dr. F. Hofmann
	Prof. Dr. M. Dal Cin

ABSTRACT

Three Dimensions of Scheduling

by

Frank Bellosa

The research presented in this dissertation has concentrated on the gathering and usage of memory access patterns in the area of user-level as well as kernel-level scheduling, yet covering both time-sharing and real-time aspects. Beside the questions when a task has to be executed and which CPU should be used, the freedom of scheduling is enlarged to a third dimension, the speed of execution.

To alleviate the cache effects perceivable after a context switch, *Follow-On Scheduling* extracts information gathered by the virtual memory subsystem to identify threads which share many pages. These related threads are scheduled so that they follow upon each other when being executed.The benefit of using memory access patterns on the coarse level of page access lies in the reduction of the number of cache misses a thread experiences after the switch if a related thread has run on the same CPU before.

The trade-off between scheduling overhead and performance gain due to better locality of reference favors strategies using memory access patterns on the level of cache access in architectures with non-uniform memory access. A promising strategy using cache-miss information is based on a Markov model to estimate the cost to establish a thread's footprint in the cache after restarting it. This strategy offers the best process reordering and makes a fine-grained architecture-independent programming style possible.

A novel approach, called *Process Cruise Control*, effectively isolates real-time threads from the timing and memory-access characteristics of other threads running on different processing units in a multiprocessor environment. *Process Cruise Control* avoids the malicious effects of memory preemption by a complete memory-bandwidth reservation scheme based on information derived from memory-access counters in the hardware. The execution speed of soft real-time applications is maintained while other applications with high memory demands are throttled in their speed of execution.

Table of Contents

Table of Contents . . . . *i*

List of Figures . . . . *iii*

List of Tables . . . . *v*

1. Introduction 1

1.1 The CPU Scheduler – An Autonomous Resource Manager? *1*

1.1.1 Processor Caches *2*
1.1.2 Main Memory *2*
1.1.3 Power Supply *3*
1.1.4 I/O *4*

1.2 Three Dimensions of Scheduling *4*

1.3 Sources of Memory Access Information *6*

2. Kernel Scheduling 9

2.1 Criteria of Traditional Kernel Scheduling *9*

2.2 Follow-On Scheduling *10*

2.2.1 The Impact of Scheduling on Caches *10*
2.2.2 Promotion of Cache Reuse by Shared Pages *11*
2.2.3 TLB-Miss Guided Enqueueing *13*
2.2.4 Measurements *15*
2.2.5 Prerequisites of Follow-On Scheduling *17*

2.3 Conclusions *21*

3. User-Level Scheduling 23

3.1 Scheduling Demands of High Performance Computing *23*

3.2 Erlangen Lightweight Thread Environment (ELiTE) *25*

3.2.1 Architecture of the CONVEX SPP *26*
3.2.2 Scheduling Architecture of ELiTE *27*
3.2.3 Implementation Details *30*

3.3 Affinity Scheduling *33*

3.3.1 Scheduling Strategies *34*
3.3.2 Interpretation of Measurements *40*

3.4 User-Kernel Interaction *45*

3.5 Conclusions *46*

4. Soft Real-Time Scheduling 47

4.1 The Neglected Resources of Real-Time Scheduling *47*

4.1.1 Caches *49*
4.1.2 Power *51*
4.1.3 System interconnection *53*
4.1.4 Main Memory Bandwidth *53*

4.2 Process Cruise Control *54*

4.2.1 Memory-Bandwidth Reservation and Throttling Model *55*
4.2.2 Admission Control and Throttling Policy *64*
4.2.3 Proof of Concept *66*

4.3 Conclusions *66*

5. Perspectives 69

5.1 Memory-Conscious Data Structures *69*

5.1.1 Cacheability *69*
5.1.2 Prefetchability *70*
5.1.3 Mapability *71*
5.1.4 Example *71*

5.2 Total Speed Control *74*

5.3 Towards Optimal Affinity Scheduling *75*

6. Conclusions 77

6.1 Contributions of this Research *77*

6.2 Future Directions *78*

Bibliography *79*

List of Figures

Introduction

Fig. 1.1: The Three Dimensions of Scheduling 5
Fig. 1.2: Two-level cache architecture 6

Kernel Scheduling

Fig. 2.1: With physically indexed caches, cache entries of shared pages can be reused. . . 12
Fig. 2.2: Reordering of run-queues by Follow-On Scheduling decisions 12
Fig. 2.3: TLB-miss guided enqueueing 14
Fig. 2.4: Enhancing the thread- and CPU-context by cache misses and clock ticks 15
Fig. 2.5: CacheExplorer97 16
Fig. 2.6: Page placement. 18
Fig. 2.7: Additional page conflicts in a direct mapped physical cache 19
Fig. 2.8: Page conflicts in a direct mapped physical cache 20

User-Level Scheduling

Fig. 3.1: Stages required to access various levels of the memory hierarchy 27
Fig. 3.2: Scheduling Architecture ≈ Memory Architecture 28
Fig. 3.3: Central vs. distributed hierarchical structures 29
Fig. 3.4: Distributed lock structures 30
Fig. 3.5: Fast synchronization and context switch with 4096 threads 32
Fig. 3.6: Running threads increase the number of valid cache lines 35
Fig. 3.7: Footprint growth due to cache misses 37
Fig. 3.8: Stopped threads lose cache lines due to cache misses of running threads 38
Fig. 3.9: Displacement of the footprint. 38
Fig. 3.10: Reload Transient. 40
Fig. 3.11: Performance of applications with regular execution order 41
Fig. 3.12: Cache misses/synchronizations of applications with regular execution order . . . 42
Fig. 3.13: Active Threads Smoother. 43
Fig. 3.14: Performance of adaptive applications with irregular execution order 43
Fig. 3.15: Speedup of adaptive applications with irregular execution order 44

Soft Real-Time Scheduling

Fig. 4.1: Reduced execution speed of the video tool caused by memory preemption 56
Fig. 4.2: Memory Bandwidth Measurement. 58
Fig. 4.3: Adaptive numerical applications 60
Fig. 4.4: Relation between TLB-miss loop-count and STREAMS copy-rate 63
Fig. 4.5: Memory-bandwidth capacity reservation. 65
Fig. 4.6: Soft-realtime video-image processing with Process Cruise Control 67

Perspectives

Fig. 5.1: Sequential and merged allocation of data records 70
Fig. 5.2: Linkage of the `kthread` structure 71
Fig. 5.3: Static array of compound thread records 73
Fig. 5.4: Attracting and distracting threads. 76

Conclusions

List of Tables

Introduction

Tab. 1.1: Cache hierarchy of contemporary 64 bit processors 6

Kernel Scheduling

Tab. 2.1: Snapshot of runqueue 59 14

User-Level Scheduling

Tab. 3.1: Context switch latency in ELiTE 30
Tab. 3.2: Share of remote sync objects and uncached memory. 32
Tab. 3.3: Scheduling strategies 34

Soft Real-Time Scheduling

Tab. 4.1: Strong ARM SA 110 power/clock modes 52
Tab. 4.2: Copy rate of the STREAMS-benchmark (stream_d.c) 55
Tab. 4.3: Memory-Access Patterns (SUN E3000, 4 Proc., 128 MB 512 K E-Cache) 60

Perspectives

Tab. 5.1: Size of kernel structures 72

Conclusions

1 Introduction

The environment of a computer system within which applications can do useful work is determined by the hardware resources and the software managing these resources. The process scheduler is the heart of the resource management. It controls the activity of the processing units and keeps the sequence of execution up to date. In the first computer systems the process scheduler was the dominant instance. Other resource management entities had subordinate importance only. During the past 40 years the relationship between the process scheduler and other resource managers has changed. More and more parameters had to be considered to determine a schedule for the processing units. This complexity impedes an efficient and deterministic control of execution with current operating systems in which adjusting scheduling parameters to achieve specific results is at best a black art [WW95].

The aim of this thesis is to investigate advanced scheduling policies and their interaction with other resource management entities. We spread the freedom of choice in process scheduling to three dimensions: the time, the location and the speed of execution. Special focus is on the bilateral influence of memory access and CPU activity.

1.1 The CPU Scheduler – An Autonomous Resource Manager?

The exact quality of a schedule can only be determined if the schedule is predictable. However a predictable schedule lasts only as long as the world remains unchanged, and therefore a judgement concerning quality may be of limited value in a highly dynamic environment [LM90].

Executing a process implies operations on many resources. If we can predict, measure and control those implications, we are able to reduce the dynamics and therefore we can improve the quality of scheduling.

In the following subsections we describe the interaction between the CPU scheduler and the other main system resources: caches, main memory, power supply and I/O.

1.1.1 Processor Caches

Over the history of semiconductor circuits, processor speed and memory capacity have almost increased exponentially, whereas the memory latency has not improved significantly. The memory access time limits the system performance more and more thus leading to a phenomenon know as the *Memory Wall* [WMc94]. Multiple levels of caches are a common approach to deal with this problem. Processor caches are designed to store the most recently used subset of the main memory and to feed fast microprocessors at low latencies. From the CPU's point of view, caches are transparent devices to bridge the widening gap between CPU and memory. The sequence of memory references issued by the CPU determines which memory regions are stored in the caches. An encached memory region can be accessed with a low latency. If a referenced memory region is not stored in the caches, the CPU has to wait for the transfer of code or data from the main memory to the cache and the CPU registers. While the CPU is waiting, it has to stall for several cycles.
The CPU determines the content of the caches by its references, whereas the caches determine the number of stall cycles the CPU is doing when accessing memory. The more stall cycles happen the more the speed of execution is reduced. Therefore the CPU influences its own efficiency by executing a specific stream of instructions with its corresponding memory access patterns.
The CPU scheduler projects the sequence of execution. The dispatcher assigns the tasks to the processing units according to the decisions of the scheduler. The sequence of execution and thus the sequence of instructions is determined by the scheduler. Consequently the scheduler influences the effects of caches on the speed of execution.
While the effects of scheduling strategies on caches have been investigated comprehensively, [MB90][SL93][TTG95] the feedback of cache-related information to the operating system has not been considered. If we know about the interaction between CPU and cache and if we can observe it, we can also control the instruction stream in the scheduler in a way that the cache and CPU timing behavior is more predictable and the efficiency is improved.

1.1.2 Main Memory

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 can issue many more memory requests than the memory can service. The memory bandwidth is the volume of data that can be transferred from/to memory in a time interval. The number of requests the main memory can service per time interval and therefore the memory bandwidth is limited. A CPU that issues more memory requests than can be serviced by main memory has to stall. Stall cycles reduce the efficiency of execution. In multiprocessor systems the available memory bandwidth is shared by all processing units and DMA devices. Consequently, the processing units can interfere and influence their efficiency mutually when accessing memory. We call this effect *memory preemption.*
The number of memory requests that a processing unit can issue at best is determined by the clock frequency and the memory access patterns of the task running on the CPU (see Section 1.1.1 *Processor Caches*). The scheduler in a multiprocessor system determines the tasks run-

ning in parallel on the processing units. The interference of the parallel tasks caused by memory preemption influences the efficiency of execution. The effect of bandwidth limitations is continuously being investigated by many system-benchmarks to characterize the architectural properties of computer systems [McC95][BS97]. But sustained bandwidth is deduced from synthetic benchmark results and not from measurements inside the hardware, so the behavior of real-world applications cannot be observed.

If we know the characteristic limits of the main memory and can observe the memory access of the processing units, we can consider the memory access patterns of individual tasks in the scheduler to predict and control the effects of memory preemption. Throttling the memory access or tuning the clock frequency of individual processing units would be a further means of control to increase the predictability of schedules and to improve the efficiency of execution of tasks with high priority.

1.1.3 Power Supply

Power supply and cooling are an environmental condition that was assumed to be available independent of the device operations. But this assumption does not hold for portable devices with a limited battery-power and passive cooling- capacity. Therefore power is an essential resource with an emerging impact on power-sensitive devices requesting high service rate objectives (e.g., in hand writing recognition on personal digital assistants).

The power that integrated circuits need for operation is proportional to the number of gates and the clock frequency. The high power consumption leads to the problem of power supply, power dissipation and cooling. A high performance processor consumes between 26W (UltraSPARC-II at 250Mhz) and 60W (alpha 21264 at 300 MHz). Facing the trend of rising clock frequency and chip complexity two questions arise. Can future systems be supplied with enough power (e.g., in notebook computers or PDAs) and can the power be dissipated by cooling elements and fans? Is it possible to reduce the power consumption by sophisticated scheduling and power management techniques?
Some of today's processor architectures offer the feature of reducable clock speed to save power [INTEL96B]. Reducing the clock speed causes a linear reduction of energy consumption, but a similar reduction of performance. So the measure of the energy performance defined in millions of instructions per joule (MIPJ) is unchanged. However a reduced clock speed creates the opportunity for quadratic energy savings as the energy is proportional to the square of the voltage. The voltage level can again be reduced within a range of 2.8 Volts to 1.4 Volts as long as the clock rate is reduced in the same manner [WWDS95].

Although the problems of power and cooling do not represent the focus of this thesis, we have realized that the control of the speed of execution that is necessary to deal with the problems of memory preemption (see subsection 1.1.2) simultaneously solves the problem of power control. Consequently there is a bilateral relationship between scheduler decisions and the power consumption of the system which we should keep in mind.

1.1.4 I/O

Operating systems establish layers of abstraction to hide complex details, to virtualize and to protect hardware devices which are used to in- and output data. Device drivers or device managers have been developed to make the handling of devices feasible. A drawback of this approach is a lack of information exchange between process manager and device manager. A running process may submit a request to a device independent of its priority. The device handles the request without any knowledge of priority. If the request has been serviced, the submitting task is awakened and enqueued according to the policy of the scheduler. On a monoprocessor architecture the handling of requests reflects the policy of the scheduler.
In a multiprocessor with enough processing units a device can be saturated by a low priority task running on a free CPU, because the scheduling tries to keep the processing units busy without any knowledge of the state of a device.
A remedy is to associate every activity in a computer system with an active entity like a task or thread. The scheduler is responsible for the management of threads. Therefore it should have all available knowledge to account I/O operations to the tasks initiating them. This presumes an information exchange between CPU scheduler and I/O devices to make the schedules more predictable and to increase the quality of scheduling decisions.

1.2 Three Dimensions of Scheduling

Process scheduling has goals that have to be fulfilled by making decisions with respect to necessary resources. In the last subsection we have outlined the interaction between the CPU and the essential resources memory, caches, power supply, and I/O. Scheduling decisions influence those resources, but the usage of them has an impact on the execution environment of running processes and therefore on future scheduling decisions as well.
In the past, scheduling had two dimensions. In a uniprocessor the scheduler had to decide which thread of control should run on the CPU. In multiprocessors another dimension was added: not only the decision *when* a thread will run, but also *where* it will run i.e. on which CPU [Fei97]. Beside the questions when a task has to be executed and which CPU should be used, we enlarge the freedom of scheduling to a third dimension, the speed of execution to control memory-preemption and power-consumption effects. Thus parallel systems allow a *three-dimensional division of resources* among competing threads: in time (*when?*), in space (*where?*) and in velocity (*how fast?*). Within the space of decision, the scheduler has to control the execution in a way that the scheduling goals are achieved (see figure 1.1).
The quality of a schedule can be improved if we know the interaction between the resources of interest exactly. The more detailed the information is, the better the schedule can be computed. Bearing this knowledge in mind we can design improved resource management mechanisms and scheduling policies.
The coverage of all resources would go beyond the scope of this thesis. Therefore we are neglecting the field of I/O and are focussing on the area of cache and memory related issues.

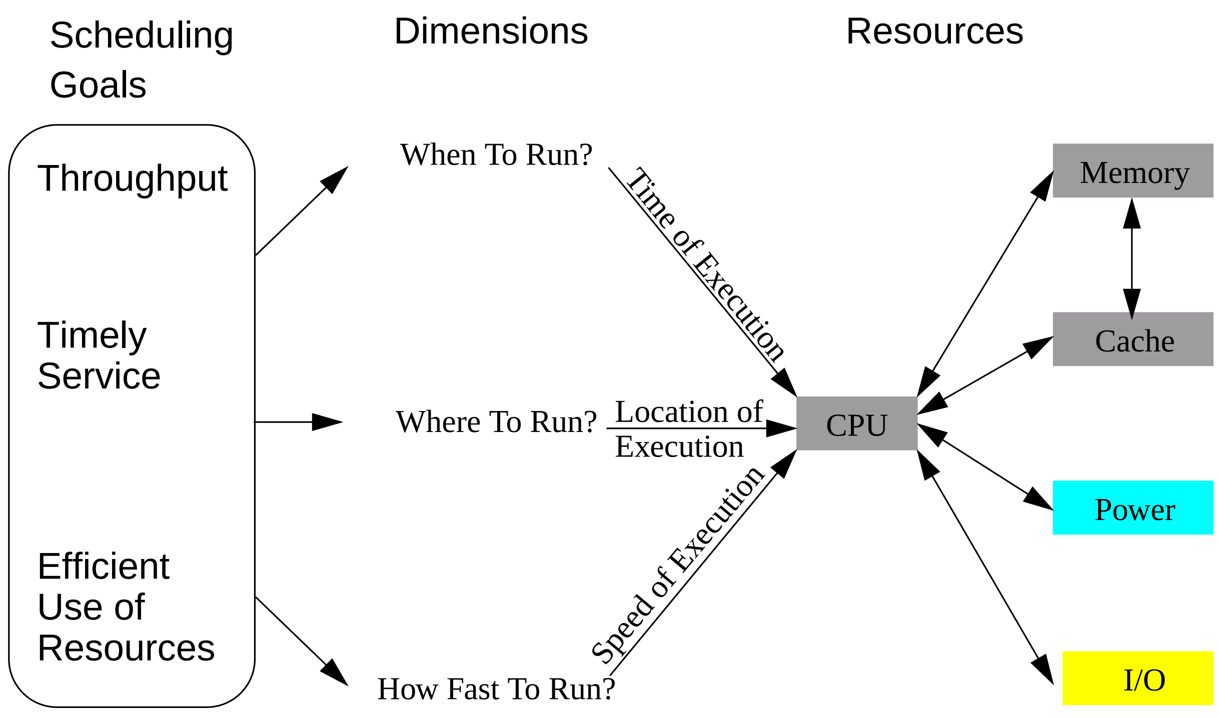


Fig. 1.1: The Three Dimensions of Scheduling

Two types of events build the internal runtime information for scheduling in contemporary kernels: timing information in the form of timer interrupts or clock counters and I/O related interrupts. Timer interrupts make time-sharing scheduling possible whereas clock information is the basis of processor affinity-scheduling and soft real-time frame- or deadline-scheduling [BB95]. Interaction with slow I/O devices leads to blocking of processes in the kernel. Deblocked processes due to finished I/O operations normally are executed in privileged mode to improve the interactive performance and to save buffer resources. Furthermore, the importance of specific types of I/O operations influences the priority boost for deblocked processes. Interaction with slow devices is therefore an indication for priority adjustment managed by the scheduler.

Memory speed is often a major component of the perceived execution speed of the computer since the processor can only execute as fast as the memory system provides data. As processors and cache memories increase in speed, memory system performance has become increasingly sensitive to the usage patterns and policies of operating systems [GZH93]. Consequently, memory access operations, the reading and writing of data to relatively slow main memory, have to be considered analogous to I/O operations. High speed processor caches are the counterpart to I/O buffers. The main memory access of high priority processes via interconnection networks is equivalent to high priority I/O operations. If it is accepted to influence execution priorities by slow I/O events, why should scheduling neglect events related to other slow devices like main memory and memory data paths?

We introduce an additional class of scheduling parameters: *memory access patterns*. If the operating system knows about the memory access characteristics of all tasks in the system it can manage the execution of tasks, such that the effects of memory access are predictable and therefore an optimization toward increased efficiency and throughput is possible.

1.3 Sources of Memory Access Information

In order to investigate potential sources of memory-access information we take a look at contemporary 64 bit cache architectures.

Processor	First Level	Second Level	Third Level
Ultra SPARC I	16 KB virtual data 16 KB physical instr.	512 KB - 4 MB physcial	-----------
MIPS R10000	32 KB virtual data 32 KB virtual instr.	512 KB - 16 MB physical	-----------
PowerPC 620	32 KB physical data 32 KB physical instr.	1 MB - 128 MB physical	-----------
Alpha AXP21164	8 KB physical data 8 KB virtual instr.	96 KB physical	1 MB - 64 MB physical
HP PA-8000	1 MB - 4 MB physical data 1 MB - 4 MB physical instr.	-------------	------------

Tab. 1.1: Cache hierarchy of contemporary 64 bit processors

They use primary virtually or physically indexed caches with physical tags and secondary physical caches. In typical computer architectures with a CPU that uses first- and second-level caches we can identify six locations (see figure 1.2) where information related to memory access can be registered.

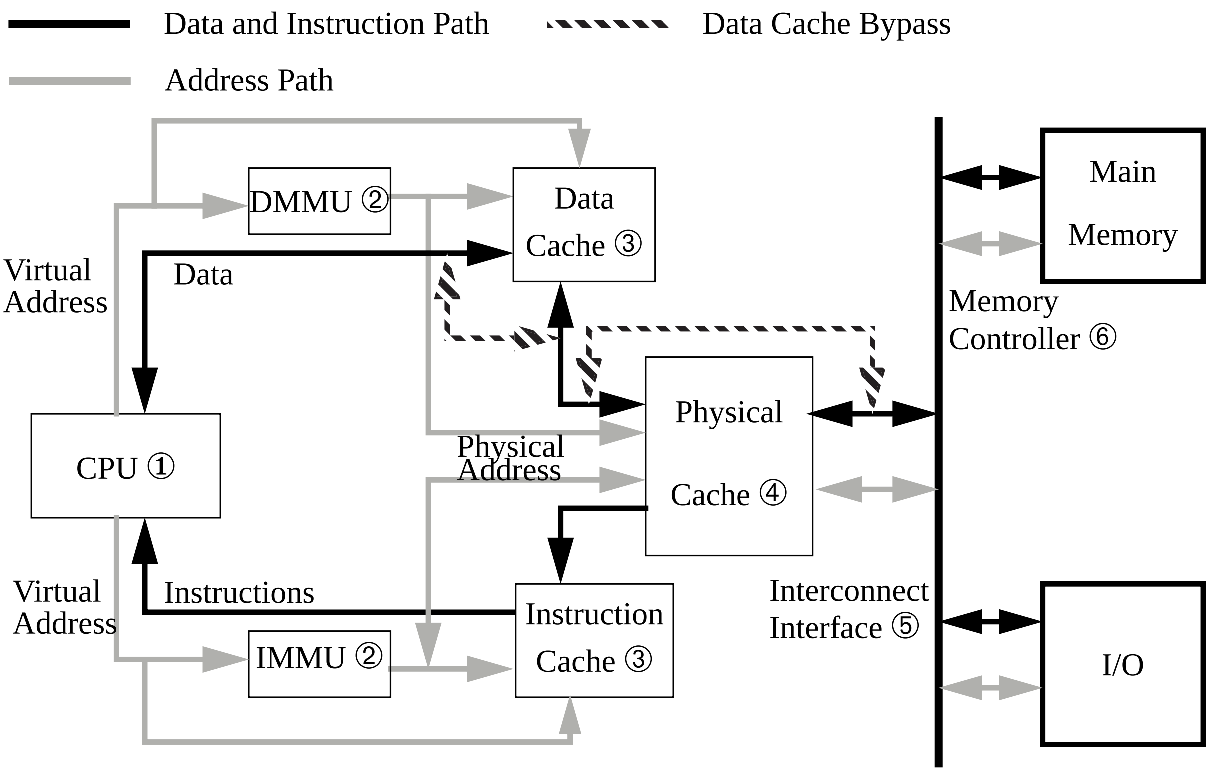


Fig. 1.2: Two-level cache architecture

- Counters in the processing unit ① can register
  - the number of cycles.
  - the number of memory references.
  - the number of stall cycles while waiting for data or code.
- The memory management unit ② can register
  - the number of TLB misses for data and instructions.
  - the mapping from virtual to physical pages.
- The cache controller for the first level cache(s) ③ and second level cache(s) ④ can register
  - the number of cache references.
  - the number of cache hits.
- The interconnect interface (e.g., bus connector in a bus-based system) ⑤ can register
  - the number of transferred data and address packets.
- The memory controller ⑥ can register
  - the number of memory requests which have been serviced by each memory bank.
  - the number of bus cycles that memory requests had to wait for to be serviced.

System designers have always observed the behavior of computer systems with hardware monitors to detect bottlenecks and to check the performance gain of architectural improvements. These points of measurement include the locations in the memory-access path mentioned above. In the past these sources of information could not be exploited for online usage in an operating or runtime system because the measurement was done with external hardware monitors. Because of increased clock speed reaching up to several hundred megahertz, external hardware monitors became more and more expensive due to electrical prerequisites. Today internal event monitors that count events with 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.
Today information from internal event counters is usually used only during the phase of system development and sometimes for off-line profiling [ZLT+96][ABD+97][Con97]. But why should we ignore information for optimizing the behavior of a computer system at runtime?

In this section, we have identified the locations where information related to the memory access can be gathered. This information will be used in the next chapters to improve the quality of kernel scheduling (see chapter 2), user-level scheduling (see chapter 3), and real-time scheduling (see chapter 4). Into the bargain we will expand the freedom of execution to the three dimensions location, time, and speed. A discussion about further research in memory-conscious operating system design (see chapter 5) will complete this dissertation. The final conclusion is drawn in chapter 6.

2 Kernel Scheduling

The heart of an operating system is the CPU scheduler. It controls the activity of the processing units and keeps the sequence of execution up to date. It might be sensible to choose carefully the sequence of processes in advanced computer architectures in which the sustained performance is critically dependent on cache performance. Cache performance depends on locality of reference. When an operating system switches contexts, the assumption of reference locality may be violated because the instruction and data of the newly-scheduled process may no longer be in the cache(s). Therefore we have developed a new approach for evaluating the cache-related performance impact of virtual memory and in particular a dynamic scheduling policy to alleviate the cache effects directly following a context switch.

2.1 Criteria of Traditional Kernel Scheduling

Scheduling algorithms can be characterized by the criteria they apply to determine the sequence of execution [SG94].

- CPU utilization:
  The scheduler tries to keep the CPU busy. In a system dispatching threads according to priorities this goal can be achieved by assigning different priorities to different classes of threads if they become runnable. The scheduler assigns a high priority to a thread which becomes deblocked. Threads which are CPU bound are assigned a low priority. This keeps the CPU busy during I/O waits and gives short running threads the opportunity to issue as many blocking calls as possible.
- Throughput:
  The goal is to do as much work as possible by using all resources in the system in the best manner. Sometimes there has to be a trade-off between CPU- memory- and I/O-utilization. E.g., to keep the I/O system busy and to guarantee a high throughput of I/O bound threads, memory has to be used for I/O buffers so that there is not enough memory in the

system to run memory-bound threads efficiently. If there are many runnable memory-bound threads, the throughput can be improved by executing them sequentially to avoid all paging or swapping effects.

- Response Time:
  In an interactive system the time span between the submission of a request till the first response is produced should be minimized. The priority of a thread in an information server (e.g., in a web server) has a high CPU and I/O priority to transfer the first packets of a request with a low latency. After the first burst of packets the priority decreases to a low level. This is motivated by the observation that a transfer is frequently aborted shortly after the first response packets. In order to support a fast overview with a continuous improvement of information, some formats for the description of pictures were designed so that the picture can be displayed with a low resolution after the first packets and is improved with further packets. Reducing the response time enables more interactive connections while reducing unnecessary transferred data, as the user has the chance to abort a transfer of uninteresting data. Again, this is a kind of throughput because it helps users to make faster progress towards a solution.
- Real-time criteria:
  In a real-time environment a special quality of service (QoS) has to be guaranteed. Quality of service can have many different flavors like
  - a cyclic thread has to be executed once per time-period
  - a thread has to come to completion before a specified deadline
  - the response-time is not allowed to exceed a threshold.

2.2 Follow-On Scheduling

All contemporary scheduling policies which try to satisfy some of the above mentioned criteria assume a fixed speed of execution. They assume that the execution of one task does not affect the speed of an other, provided that there are no effects from the virtual memory subsystem (paging, swapping). But this assumption does not hold for advanced RISC architectures in which the sustained performance is critically dependent on cache- and MMU performance.

In the next subsection we investigate the impact of scheduling on caches and propose a scheduling policy using the memory access information gathered in the memory management unit (see figure 1.2 ②) to avoid context-switch-related cache misses.

2.2.1 The Impact of Scheduling on Caches

The sustained performance of a computer is considerably influenced by the fast supply of data and instructions to the processing unit. Processor caches are designed to store the most recently used subset of the main memory and to provide this subset with low latency in order to narrow the gap between the slow main memory and the fast processing unit. Cache performance depends on locality of reference; when the sequence of addresses referenced by software cannot all be stored in the cache, cache misses result. In modern computers, the penalty for a single cache miss might be tens of cycles (e.g., 50 cycles on a Sun E3000 architecture [SUN95B]). Most contemporary cache architectures (e.g., UltraSPARC [SUN95A], MIPS R10000 [MIPS95], DEC AXP 21164[DEC95], PPC620[MOT94]) use second and third level physical caches to facilitate cache coherency and to avoid cache flushing during a context switch (see table 1.1). Memory of multiple contexts can simultaneously co-reside with portions of the operating system in physical caches. If two threads with a different working set follow upon each other, the cache content is displaced. In this situation in which the cache is effectively cold-started, the processor stalls due to cache misses in the instruction-fetch and load/store pipeline stage. The cost to refill all of the 8192 cache-lines of an UltraSPARC-I processor (167 MHz, 512KB E-Cache, 50 cycles latency see [SUN95B]) is about 2.4 milliseconds. With about 300 context-switches per second, the system is indeed busy with reloading the cache, and produces almost useless work. Therefore it might pay to choose carefully the order in which to schedule processes.

2.2.2 Promotion of Cache Reuse by Shared Pages

While small virtual caches are common as primary caches because they permit fast access, physical caches dominate the area of second level caches. Physical caches ease cache consistency for I/O operations and multiprocessing and require no additional support to eliminate aliases or ambiguities. Physically indexed caches store a small subset of physical memory without any consideration concerning context, address space and memory mapping. Therefore, physically indexed caches can store the working sets of multiple threads, even if these threads belong to different contexts.

If the working sets of threads differ, large portions of the second level cache have to be loaded after a switch because multiple divergent working sets do not fit into the cache. On the other hand cache entries of shared memory regions can be reused after a switch. Switching between threads that share large parts of their working set results in few cache misses after the re-start, and thus in good system performance [BEL96B]. Our approach to improve cache reuse is to exploit the property of shared pages exhibited by many applications.

To investigate the property of page sharing we have to analyze three aspects:

- the type of memory which is shared among threads
- the sets of threads sharing memory
- the degree of sharing between threads

Multithreaded applications and those using shared memory segments for inter-process communication (IPC) share physical pages filled with data. Shared code pages can be found if a shared library is mapped into multiple address spaces or if multiple instances of the same executable are running. Threads which trigger similar kernel activities (networking, filesystem, synchronization) also share pages related to the kernel (see figure 2.1).

Our analysis of typical application- and information-servers shows that a large fraction of the physical memory pages is shared by multiple contexts. In a highly loaded server system, there

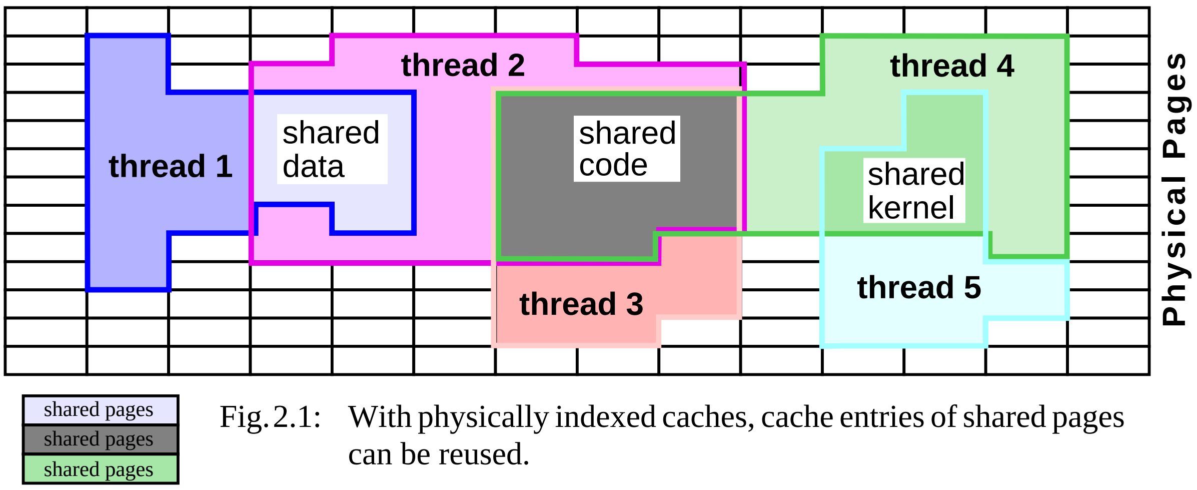


Fig. 2.1: With physically indexed caches, cache entries of shared pages can be reused.

are normally many runnable threads in each runqueue sharing a lot of memory. It is desirable for threads with the same working set to follow upon each other to reuse the content of the cache, but in contemporary operating systems the sequence of execution is independent of working-set aspects.

Those threads which share pages have to be identified. If two threads share a large portion of their working set they are declared as *relatives*. The scheduler should influence the dispatcher in a way that related threads follow upon each other. We introduce this technique called ***Follow-On Scheduling*** [BEL97B].
The benefit of Follow-On Scheduling lies in the reduction of the number of cache misses a thread experiences after the switch, if a related thread has run on the same CPU before. If *related threads* reside in the same dispatch queue, the order of the dispatch queue is carefully changed so that related threads follow upon each other. Especially on highly populated run-queues, the number of cache misses due to changed working sets can be considerably reduced. A good example of a highly populated queue is the fixed-priority dispatch queue dedicated to threads returning from sleep in a Unix System V Release 4 (see figure 2.2).

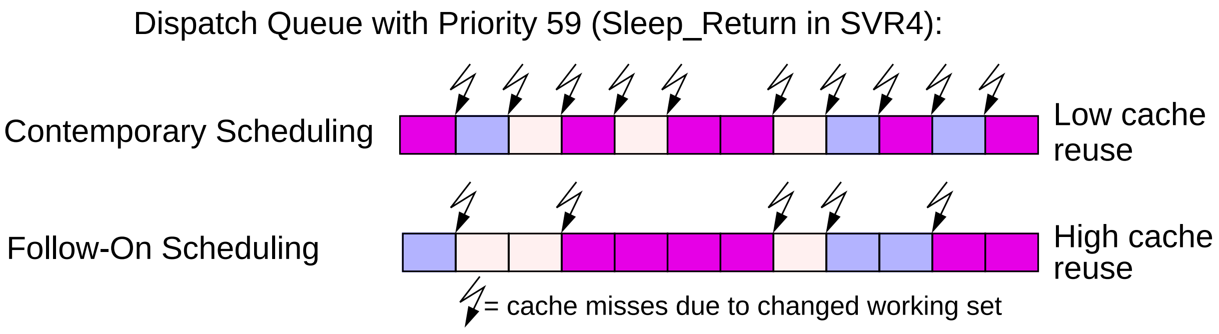

Dispatch Queue with Priority 59 (Sleep_Return in SVR4):
Contemporary Scheduling
Low cache reuse
Follow-On Scheduling
High cache reuse
= cache misses due to changed working set

Fig. 2.2: Reordering of run-queues by Follow-On Scheduling decisions

On a highly loaded server we observed more than 50 runnable threads in this queue. The longer the queue, the more effectively the sequence of threads can be optimized.The determination of related threads is a compute-intensive procedure because there exists no trivial mapping from protection domains (physical pages – contexts) to active entities (threads). We were therefore forced to implement a reverse lookup mechanism to coalesce threads and pages.

2.2.3 TLB-Miss Guided Enqueueing

Scheduling decisions are based on information. Follow-On Scheduling extends the deciding factors of contemporary schedulers (timing, priority, event-type, swap-state of process, reason for blocking) by adding information concerning the relationship between threads. To gather this additional information we considered three data structures which store memory-access information.

- The page table provides the complete mapping from virtual to physical address space. The detection of heavily used pages is difficult due to the low scan rate of the clock algorithms used to provide information for paging decisions. The page table is not a CPU-local structure, so we cannot distinguish the page access of multiple processors. Traversing the page table is a complex task and the access bits of the page table entries do not provide information to distinguish the page access of multiple threads assigned to the same process. Finally, the page table does not allow the detection of shared kernel resources.
- Small memory regions are frequently used to cache the page table. These page table caches narrow the gap in access speed between the tiny full-associative TLB and the complex page table. The translation storage buffer (TSB) of the SPARC V9 architecture [SUN95A] is one example of such caches managed by software. The design objective of the TSB is to store the hot spots of the page table in the CPU caches, speeding up the resolution of TLB misses. The TSB is a CPU-local structure which is easy to access. The typical size of the TSB allows the analysis of the relationship between processes as the number of TSB entries (usually 32768 pages) is quite high. It can therefore be used to identify shared pages. The TSB gives no hints to enable the detection of related threads because it contains context information but no thread-specific information.
- The TLB is a hardware structure which is hard to read. But the TLB-miss handler can be modified so that it makes a note of TLB misses in a trace buffer local to each CPU. Because there is no kernel virtual memory while executing the TLB-miss handler, the thread-ID of the currently running thread has to be stored in a CPU-local memory region in physical memory. This task is done in the switch routine. Now we can trace the thread-ID causing a TLB miss and the page frame number loaded into the TLB (see ① in figure 2.3). Because the trace is not influenced by the MMU context, the page access of a thread can be observed independent of whether the thread is running in user- or in kernel-space. By periodically analyzing the trace buffer, we know exactly which pages are accessed by a specific thread and the degree of relation between all threads of execution.
  The idea of using TLB-information was proposed in [SW95] to detect *correspondent* processes on kernel level to influence co-scheduling of parallel jobs. The aim of co-scheduling lies in the reduction of coordination-overhead in a parallel computer. However, the aim of Follow-On Scheduling is the optimization of the sequential execution on a single CPU.

Despite the complex modification of trap handling, switching routines and CPU-specific data structures, we chose the third approach for our implementation of Follow-On Scheduling. A

SUN Enterprise server running the Solaris 2.5.1 operating system is the platform of our prototype.

The analyses of the trace data are done by the scheduler which updates a table representing the relation between threads surviving multiple scheduling cycles (short-living interrupt threads are filtered out). This table, called *Thread Relative Table* (see ② in figure 2.3), gives hints to the

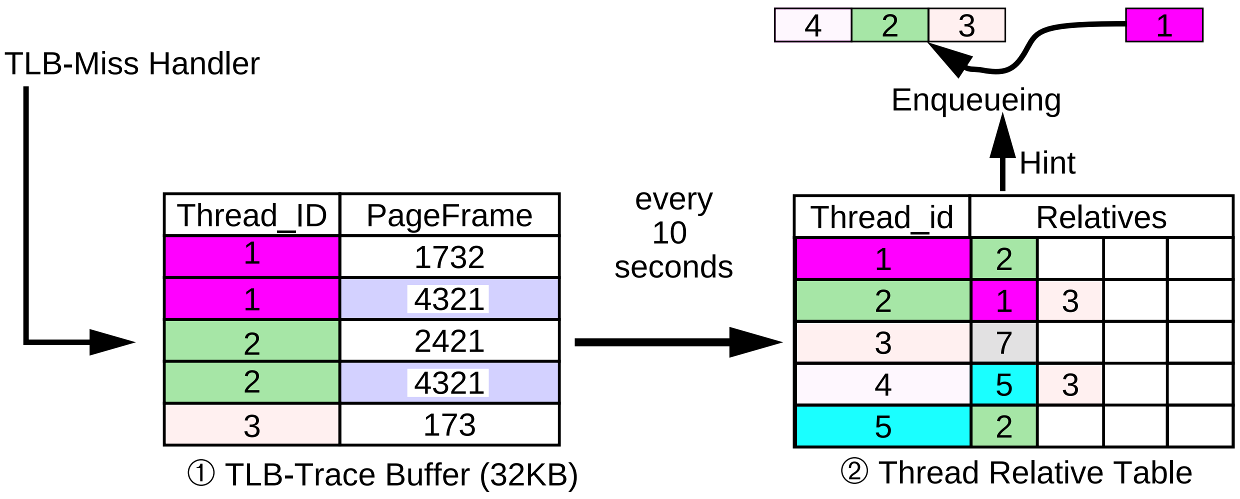


Fig. 2.3: TLB-miss guided enqueueing

enqueueing functions. If a related thread is found near the location where a thread is designated to be enqueued (normally at the front or back of the queue), other threads can be bypassed. To prevent starvation, the scope of the lookup function should be limited to two or three queue positions. Furthermore a thread can only be bypassed a few times.

Contemporary Scheduling	Follow-On Scheduling
elis	elis
elis	elis
httpd-1.2.1	elis
httpd-1.2.1	elis
elis	httpd-1.2.1
elis	httpd-1.2.1
httpd-1.2.1	httpd-1.2.1
httpd-1.2.1	httpd-1.2.1
elis	httpd-1.2.1
httpd-1.2.1	elis
elis	httpd-1.2.1
httpd-1.2.1	httpd-1.2.1
httpd-1.2.1	httpd-1.2.1
elis	elis

Tab. 2.1: Snapshot of runqueue 59

The results are very promising. On our server we could clearly identify classes of threads sharing a lot of pages. For example we ran a highly loaded Apache WWW-server and the **E**rlangen **Li**brary **S**ystem (ELiS) with 30 active users searching the libraries of the university. Both applications imply frequent blocking due to I/O events. Snapshots of the runqueues (see table 2.1) of a highly loaded server demonstrate the effects of Follow-On Scheduling. Without any application-specific knowledge, the scheduler is able to identify threads sharing physical pages and to enqueue them so that they follow upon each other.

2.2.4 Measurements

Follow-On Scheduling is running in a production environment on SUN Ultra-I workstations and on a SUN Enterprise 3000 server. As we intended to demonstrate our results in a production environment, we had to use the available hardware for an analysis and could not use a simulation of new hardware enhancements like those proposed in [HMMS96] or [JH97].
In order to determine the number of cache misses more precisely, we have established virtual event counters which register events like cache misses or clock ticks. While handling traps we update virtual 64-bit counters located in the CPU- and thread-structures using the values of the physical 32-bit counters (see figure 2.4).

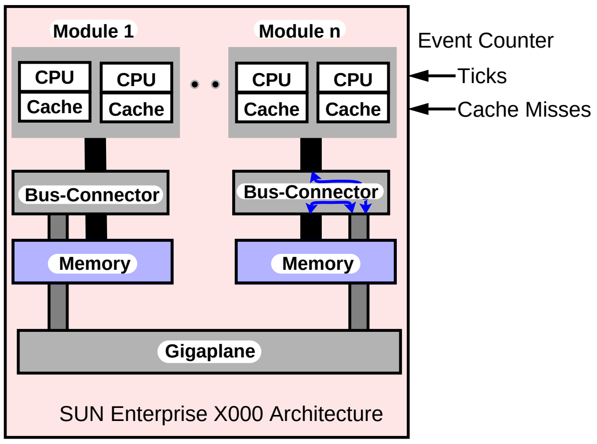


Fig. 2.4: Enhancing the thread- and CPU-context by cache misses and clock ticks

It was our intention to access the virtual counters from user-space without any system call because we do not want to influence the operating system by additional calls. Our approach is an enhancement of the `/proc`-filesystem which provides kernel-virtual addresses of those regions where virtual counters are 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. An unmodified application can now be observed from a dedicated CPU without any influence on the application's execution. To observe the behavior of individual threads we developed the CacheExplorer97, a combination of a sampling process, pinned on a free CPU and a visualization tool, running on a remote host (see figure 2.5).The explorer visualizes the location

(CPU) of execution of a thread and the thread- as well as the CPU-local number of cache references and cache misses.

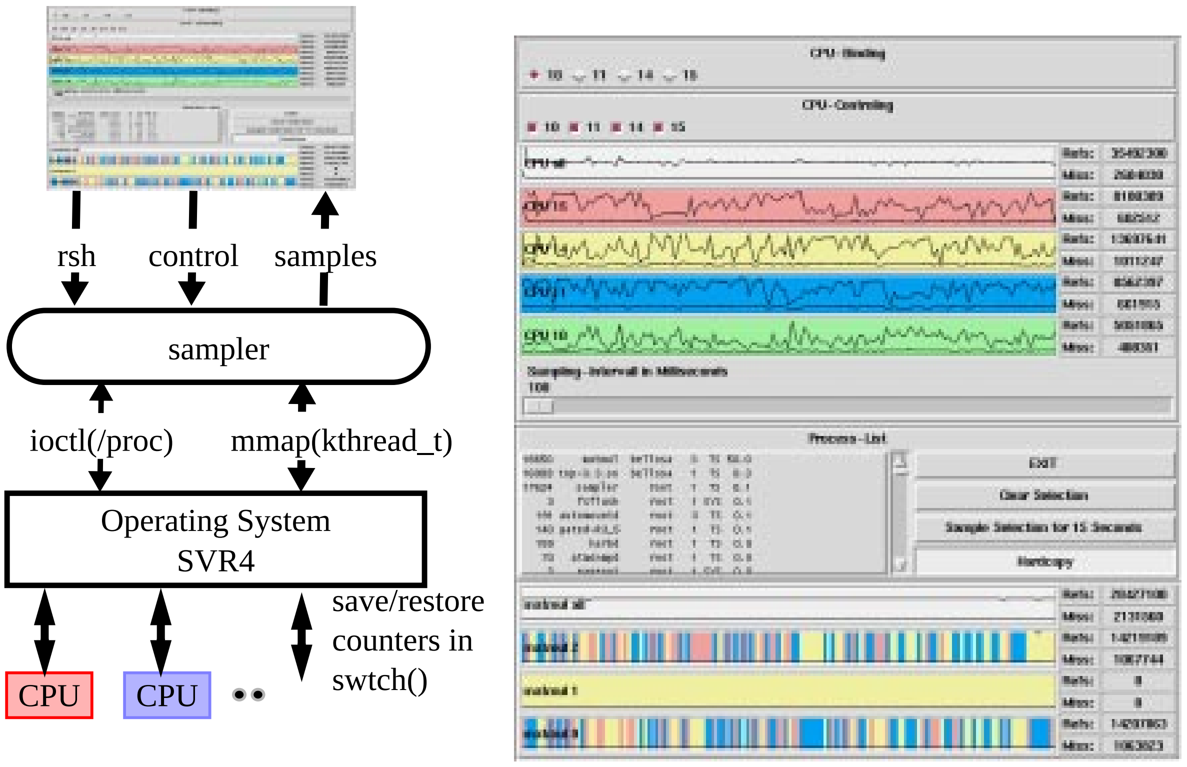


Fig. 2.5: CacheExplorer97

With approximately 20,000 TLB misses per second the overhead of TLB tracing amounts to about 5000 second-level cache misses per second. This can be neglected on a server where half a million cache misses per second occur. The analysis of the trace buffer every 2-10 seconds adds 10,000 cache misses per second and does not severely influence the cache performance. Having obtained detailed information about the cache- and TLB-behavior of our system, we realized that there is no reproducible performance improvement on our production servers. The reasons are:

- The CPUs browse through large code and data regions inside the scheduler and dispatcher (see also section 5.1 *Memory-Conscious Data Structures*). This exceeds the capacity of the caches found in our machines (512 KB 2nd-level caches).
- The shared libraries used in our applications (libc, libsocket, etc.) do not use functionality provided by the hardware (Visual Instruction Set of the UltraSPARC-I CPU) to bypass the cache while copying memory. Polluting the cache with load/store operations of one-way data displaces reusable cache entries.
- The physical pages frequently have the wrong page color. Erratic page coloring implies a large number of conflict misses.

Therefore the second-level cache is frequently cleared of shared data and code. The few remaining cache lines available for reuse are not sufficient to demonstrate that the number of cache misses can be reduced and thus the overall performance improved.

2.2.5 Prerequisites of Follow-On Scheduling

2.2.5.1 Cache Bypassing

Processor caches are designed to store the most recently used subset of the main memory and to provide this subset with low latency. But a large fraction of computational work is done with data that will never be reused. Examples are multimedia data streams, network streams, or huge data structures traversed in search operations. Processing of this types of one-way data draws no benefit of caches. On the contrary, one-way data displaces valuable cache contents.

Processor designers have recognized the problem of one-way data in the field of multimedia. Most advanced processor architectures (e.g., UltraSPARC-I [SUN95A], Pentium MMX [INTEL96A]) offer instructions to support operations on multimedia data streams. Beside arithmetic operations they offer load/store operations that transfer data between the memory and the floating point registers without storing them in the caches. These non-caching load/store operations can be used to zero and copy memory blocks as well.

But in the field of integer and logical operations on one-way data those multimedia instructions cannot be applied because they do not support the usage of the integer registers. For further research we recommend to investigate load/store instructions that:

(1) bypass the first and second level cache.
Transfers from memory directly into the registers would not pollute the caches when the CPU has to browse in search functions through huge data fields.

(2) bypass the second level cache but store data in the first level cache.
Reuse of data in the first level cache without polluting the second level cache could support operations on network streams where packets have to be assembled and disassembled. This task requires for a short period a working set that does not fit into the registers, but that is not worth to be stored in the second level caches because it will not be reused extensively.
Allocating cache lines in the first-level cache without a backup in the second-level cache breaks the law of caching that each level of the memory hierarchy stores a subset of memory blocks stored in the level below. If we do not have a backup in the second level cache, we run into trouble in the presence of multiprocessing because there is no mechanism to invalidate first-level cache lines which are not stored in the second-level cache. Consequently the proposed operations may not be applied to data that might be shared by multiple processors.

If one-way data or reduced reusable data can bypass the second-level cache, the effective footprint of a thread in the cache is reduced and therefore more footprints find room in the caches. This paves the way for scheduling policies which aim to improve the reuse of cache contents.

Our investigations of real-world applications have shown that there are enough shared pages and corresponding threads so that Follow-On Scheduling should draw remarkable benefit from reordering the sequence of executions according to the relationship of the threads. If we do not pollute the caches, we can increase their efficiency by advanced scheduling policies and improve the performance of the system.

2.2.5.2 Page Assignment and Page Re-Coloring

Large physically-indexed caches can provide memory efficiently, only if the mapping from virtual to physical pages takes the working set of processes into consideration. With operating systems neglecting the page mapping of working sets, memory blocks in different physical pages with the same prefix in their physical address (same page color) can compete for the same cache region (cache bin), even if large parts of the cache are not used. This results in bad processor performance and unpredictable execution time.

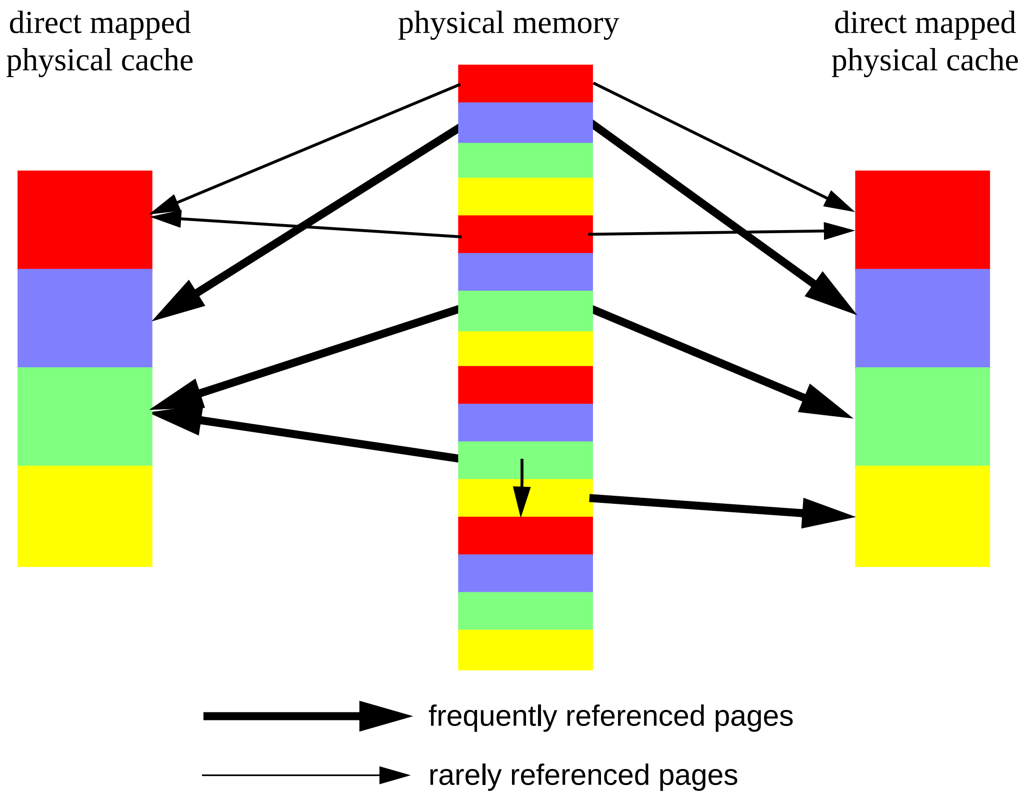

direct mapped physical cache
physical memory
direct mapped physical cache
frequently referenced pages
rarely referenced pages

Fig. 2.6: Page placement

The left placement in figure 2.6 is poor because it maps frequently referenced pages on the same cache bin where they compete for the same cache lines and cause many conflict misses. In a mapping like the one in the right side of figure 2.6 the cache bins are evenly utilized and many conflict misses can be avoided. If the number of cache misses is dominated by conflict misses, we cannot draw any profit from advanced scheduling strategies which try to minimize the number of compulsory and start-up misses.

As we could measure no benefit from Follow-On Scheduling implemented in the Solaris operating system, we took a look into the page mapping to identify it as a k.o. criterion for our scheduling approach. We investigated the page mapping in a single processor workstation with a direct mapped physical cache of 512 KB with 8 KB page size.
If we assume a random placement, we have a binomial distribution if the number of pages in the system is large compared to the number of allocated pages. The probability X that we have p pages mapped to the same cache bin when allocating P pages on a cache with C colors is:

$$X(\text{p in bin}) = \binom{P}{p}\left(\frac{1}{C}\right)^{p}\left(1-\frac{1}{C}\right)^{P-p} \tag{2.1}$$

A page conflict exists if more than one page is mapped on the same cache bin. If p pages are mapped to the same cache bin we have (p-1) page conflicts. The number of expected conflicts can be calculated according to equation 2.2:

$$RandomConflicts_{AVG} = C\sum_{p=2}^{P}(p-1)\binom{P}{p}\left(\frac{1}{C}\right)^{p}\left(1-\frac{1}{C}\right)^{P-p} \tag{2.2}$$

Using equation 2.2 we can plot the number of additional page conflicts when allocating a working set of P pages compared to an optimal page mapping.

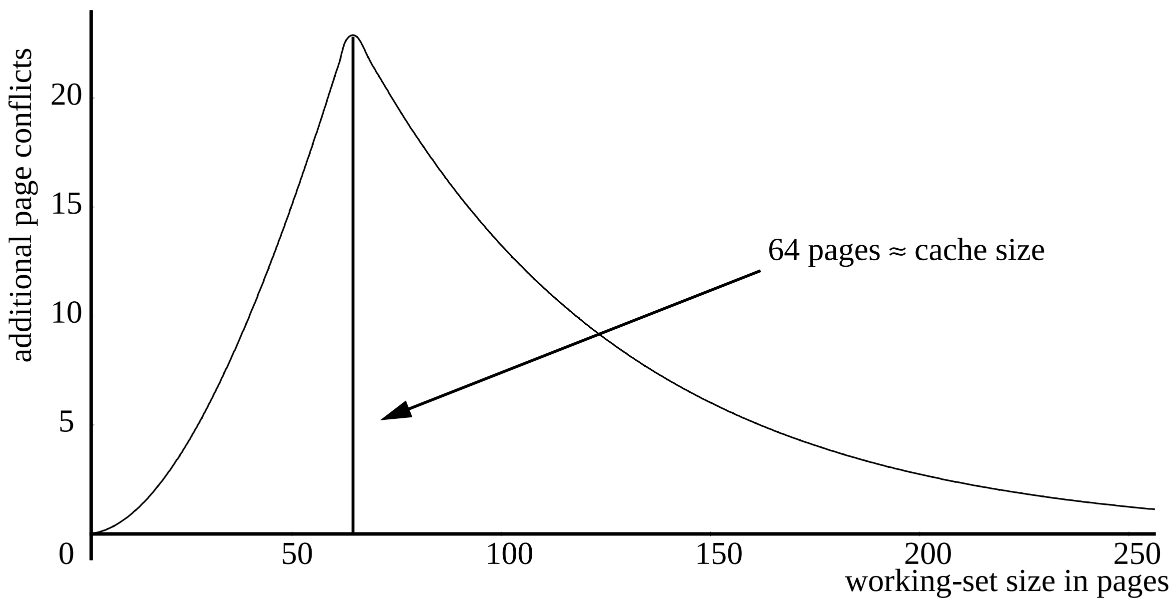


Fig. 2.7: Additional page conflicts in a direct mapped physical cache assumed random placement (512 KB cache size, 8 KB page size)

We see in figure 2.7 that the number of cache conflicts reaches its climax of 23 conflicts when the working set has the same size as the cache (64 pages). If we allocate 32 pages, we still can expect more than 7 page conflicts. Assuming a random mapping we have to realize that a remarkable number of page conflicts can be notified if the working set has a size of 25%-100% of the cache size. But does the assumption of random mapping apply to systems used for the implementation and measurement of Follow-On Scheduling?

We investigate the page mapping for memory segments in Solaris 2.5.1 Unix System V.4.

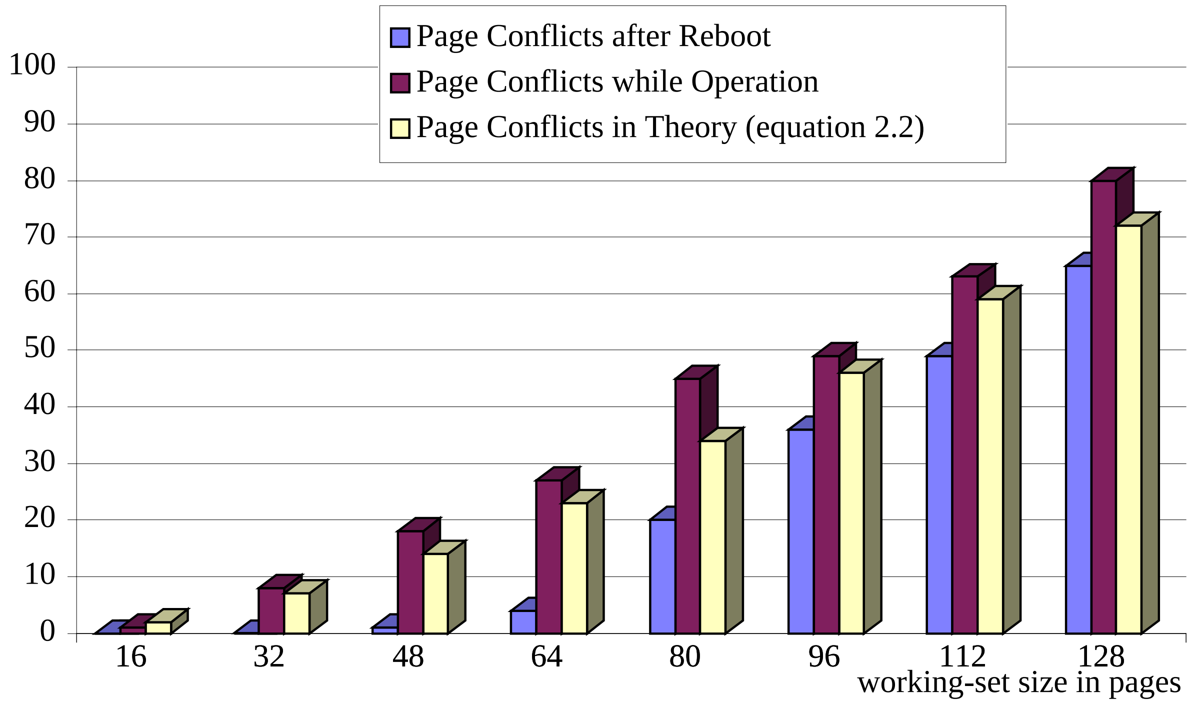


Fig. 2.8: Page conflicts in a direct mapped physical cache (512 KB cache size, 8 KB page size)

After the reboot of the system, the allocation of physical memory is almost optimal. Having this page mapping in normal operation would greatly support memory conscious scheduling strategies like Follow-On Scheduling. Unfortunately, the mapping tends to a random mapping if the system is running and many pages have been paged out and paged in.

The impact of page mapping on caches has been investigated in detail in the last years, but only a few results of the research seem to have been incorporated into today's operating systems. In [KES91] and [LF91] the behavior of multi-megabyte secondary caches has been examined and many page mapping algorithms are proposed. To deal with the problem of erratic page mapping in dynamic operation, a hardware approach [BLRC94] and a software approach [RCLB95] have been developed to dynamically copy and re-map pages to resolve cache conflicts.

As long as the cache utilization for uninterrupted threads is low due to an erratic page coloring, the scheduling can't improve the utilization for systems with frequent context switches. If some of the well known page placement and re-coloring techniques will be applied, operating systems can benefit from Follow-On Scheduling.

2.2.5.3 Spinning vs. blocking

Today's operating systems assume that I/O operations are relatively slow compared to context switches. Therefore they favor to block a thread for each I/O event and switch to another context.

Today some I/O operations are that fast that it is not worth to switch for a short moment to another context and with it to displace valuable cache contents. An example is the exchange of a

pair of minimal-length network packets in a FDDI network, which takes less than 13 microseconds [MB90]. Sometimes it might be better to busy-wait for a limited period instead of blocking to avoid the costs of a context switch.
The benefit of spinning versus blocking for short I/O operations goes beyond the avoided overhead of context switches. Reducing the frequency of context switches lowers the number of working sets which have to be stored in the caches while increasing the size of the working sets. A reasonable footprint in the cache is a prerequisite to evaluate the degree of relationship between threads. The more precise the relationship can be determined, the more benefit can a system gain from Follow-On Scheduling.

2.3 Conclusions

Sharing of memory pages is a common case in multiprogrammed computers. A design principle in computer architecture but also in software development is to make the common case fast. Therefore we should aim to reuse shared memory pages stored in high-speed caches. The virtual memory subsystem can provide the necessary information about the existence and intensity of sharing. This information can be used in the scheduler to let related threads follow upon each other when being executed on the same processing unit.
We have proved that this information can be extracted from online-traces of TLB-miss handler activities. In an environment with correct page coloring and cache-conscious system software, Follow-On Scheduling represents an essential step towards the efficient use of caches.

3 User-Level Scheduling

Parallel processing systems embedding commodity processors have a significant price/performance advantage compared to traditional vector computers. Therefore many computer centers replace their vector machines by parallel processing systems. While a high degree of vectorization was essential to exploit the full performance of traditional supercomputers, a high locality of reference and a minimal interaction between the processing units are the fundamental requirements for high performance computing on parallel processing systems. In this chapter we work out the scheduling demands on high performance scientific computing, propose advanced scheduling strategies using memory access information, and analyze their performance benefit.

3.1 Scheduling Demands of High Performance Computing

Cache-coherent multiprocessors with **n**on **u**niform **m**emory **a**ccess (NUMA architectures) like SGI/Cray Origin [SGI96] or Convex/HP Exemplar [CON95] have become quite attractive as compute servers for parallel applications in the field of high-performance scientific computing. They combine scalability and the shared- memory programming model, relieving the application designer of data distribution and coherency maintenance. But there is a potential conflict between the goals of achieving the full performance of the hardware and providing a parallel programming environment that makes effective use of programmer effort.

The parallelism expressed by “UNIX-like” heavy-weight processes and shared-memory segments is coarse-grained and too inefficient for general purpose parallel programming, because all operations on processes like creation, deletion and context switching invoke complex kernel activities and imply costs associated with cache and TLB misses due to address space changes. Threads have become a common abstraction in the field of programming languages and operating systems. Contemporary operating systems (like Solaris, IRIX or MACH) offer middle-weight kernel-level threads decoupling address space and execution entities. Multiple kernel threads mapped to multiple processors can speed up a parallel application. But kernel threads only offer a middle-grained programming model because thread management implies expensive protected system calls.

On the one hand, a moderate parallelism may appear to be necessary, both to achieve good cache performance and to limit the amount of overhead due to thread management. On the other hand, it may be more expedient to use a more natural and finer-grained programming style using a high number of threads. Most scheduling decisions are a result of synchronization conditions among the threads of an application. By moving thread management and synchronization to the user level, the cost of thread management operations can be drastically reduced to one order of magnitude more than a procedure call [ALL89]. Some advantages of user-level threads are:

- All scheduling operations belonging to a single application are handled inside the same address space. Cache and TLB misses are reduced to a minimum.
- The scheduling algorithm and its interface can be designed with respect to the needs of a specific application, thus offering the optimum in performance and functionality. For example, preemptive or priority-based scheduling of threads can be omitted to achieve low thread management overhead, if a lean scheduler is sufficient for an application.
- Data structures for processes and threads are deeply rooted in most kernels. Only the user level offers the necessary flexibility in adapting data structures to the degree of parallelism inherent in an application ranging from several to thousands of threads.

In general, light-weight user-level threads, managed by a runtime library, are executed by kernel threads which again are mapped on the available physical processors by the kernel. Efficient user-level threads are predestined for fine-grained parallel applications because they make frequent context switches affordable.
Problems with this two-level scheduling arise from the interference of scheduling policies on different levels of control without any coordination or communication. A loss of parallelism and the occurrence of a deadlock situation is possible due to blocking system calls invoked by user-level threads. Solutions to these problems are discussed in section 3.4.

The decision to design a user-level runtime system for NUMA architectures was motivated by trends in hardware technology. Powerful, modular, and scalable NUMA systems are built today by using inexpensive, small shared memory multiprocessors (SMPs) coupled with high-speed interconnection networks (SGI Origin, Convex SPP). There is also a trend in computer architecture to move from a CPU-centric to a memory-centric design philosophy. By embedding memory and CPU on the same chip the latency can be diminished. The integrated processor/ memory chip can be regarded as the building block for a NUMA architecture with a scalable interconnection fabric [SPN96].
NUMA systems use multiple stages of caches to hide latency. Locality of reference is nowhere more critical but when high performance processors rely on the effective use of caches in a highly parallel multithreaded environment. We claim that the locality issue in fine-grained parallel programs can be addressed effectively by a scheduling architecture that reflects the various levels of the memory hierarchy and that uses memory-access information derived from event counters which are buried in the memory hierarchy of NUMA systems.

Follow-On Scheduling, proposed in chapter 2, aims on the reuse of shared pages, stored in cache partitions, by exploiting page mapping information on the level of middle-grained kernel-level scheduling. However, *affinity scheduling*, discussed in this chapter, aims on the reuse of cache lines by exploiting information from event counters on the level of fine-grained user-level scheduling.
The rest of this chapter is organized as follows. Section 3.2 describes the architecture of the **E**rlangen **Li**ghtweight **T**hread **E**nvironment (**ELiTE**), a scheduling architecture for cache-coherent NUMA multiprocessors developed and implemented at the University of Erlangen. Several affinity policies are evaluated in section 3.3. We describe the interaction between kernel- and user-level scheduling in section 3.4. Finally, we conclude in section 3.5.

3.2 **E**rlangen **Li**ghtweight **T**hread **E**nvironment **(ELiTE)**

In NUMA architectures with their discrepancy between computing and communication performance, memory-conscious scheduling is essential to minimize the total completion time of an application by reducing inter-processor communication. Cache affinity scheduling for bus-based multiprocessors has been investigated in detail [SL93][Tuc93] because cache architectures become more and more dominant. The decisions within this type of scheduling are made on the basis of CPU utilization and information about the processor where a specific thread was most recently executed. Additionally state timing information from each process is used e.g. in SGI's IRIX operating system [BB95]. Our approach to memory-conscious scheduling goes beyond the use of information about timing and execution location by using cache miss information for each level of the memory hierarchy.

Most thread schedulers attempt to optimize load balancing while reducing the costs for thread management including queue locking. This strategy is reasonable for bus-based shared-memory architectures with uniform memory access. The most valuable resource of these architectures is the computing power of the processor and the bandwidth of the bus system. Thus, these scheduling policies focus on a high processor utilization while reducing bus contention [ALL89].
The focus of thread scheduling has to move when we look at scalable shared-memory architectures with non-uniform memory access. Modern superscalar RISC-based processors are able to perform multiple operations per clock cycle while simultaneously performing a load/store operation to the processor cache. A multiprocessor system can only take advantage of this immense computing power if the processors can be supplied with data in time. The bandwidth of interconnection networks is no longer a bottleneck for today's scalable parallel processors (e.g. the Scalable Coherent Interface (SCI) of the Convex SPP has a bandwidth of 2.8 GBytes/s). But switches as well as affordable dynamic memory cause a latency of about a hundred nanoseconds, while processor cycles only need a few nanoseconds. The consequence of this discrepancy is that scheduling policies for NUMA architectures have to satisfy three essential design goals:

(3) **Distributed Scheduling:** Data structures of the scheduler (run queues, synchronization objects and pools for reusable memory regions) are distributed. There are no global structures with the potential risk of contention. Concerning the allocation and disposal of data

structures like stacks, synchronization objects and thread control blocks there has to be found a trade-off between fast allocation and fast access. A fast allocator can be of limited value if the allocated memory can only be accessed with a high latency. A complex allocator might need so many cycles that the benefit of a fast access does not pay.

(4) **Locality Scheduling:** Threads are assigned to the processor which is close to the data accessed by the thread. This policy aims to reduce processor waiting time due to cache misses. Fairness among threads of the same application is not necessary as each optimally used processor cycle within an application helps to increase throughput.

(5) **Latency Hiding:** Prefetch operations cause overlapping of computation and communication.

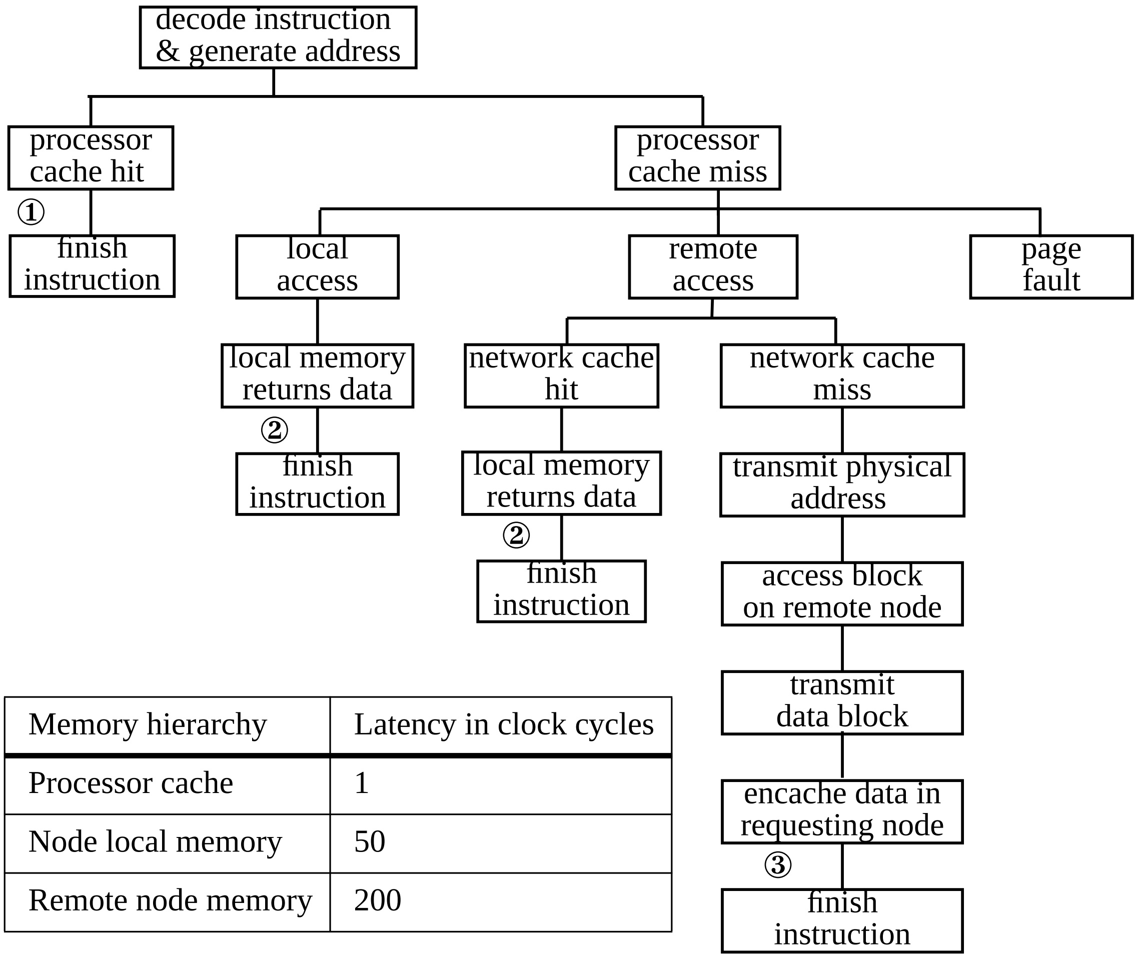
As contemporary threads packages, developed for use on shared-memory multiprocessors with a modest number of processors, have design goals which cannot be applied to scalable NUMA multiprocessors with a high number of processors, novel scheduling architectures have to be designed. After presenting the architecture of the Convex SPP, a cache-coherent NUMA multiprocessor, we describe the architecture and implementation details of the ELiTE runtime system which is a non-preemptive user-level threads package.

3.2.1 Architecture of the CONVEX SPP

The Convex Exemplar Architecture [Con95] implemented in the Convex SPP 1000 multiprocessor is a representative of cache-coherent NUMA architectures. A symmetric multiprocessor called hypernode is the building block of the SPP architecture. Multiple hypernodes share a low-latency interconnect responsible for memory-address-based cache coherency. Each hypernode consists of two to eight HPPA 7100 processors, each having 1 MB direct mapped instruction and data cache with a cache line size of 32 bytes.
The processors on a single hypernode can access up to two GBytes of main memory over a non-blocking crossbar switch. The memory in remote hypernodes can be accessed via the interconnect. To reduce network traffic, part of the memory is configured as a network cache with 64-byte cache lines. Load/store operations step through various stages depending on the locality of the referenced memory region (see figure 3.1).

There are non-blocking prefetch operations to concurrently fetch data regions from a remote node into the local network cache. These operations can be used to overlap computation and network traffic in order to hide latency.

Performance-relevant events can be recorded by a performance monitor attached to each CPU. The performance monitor registers cache misses satisfied by the local or a remote hypernode and the time the processor waits for a cache miss to be served. For high resolution time stamps, several timers with various resolutions are available. There is also a system-wide clock with a precision of 1µs.



	Memory hierarchy	Latency in clock cycles
①	Processor cache	1
②	Node local memory	50
③	Remote node memory	200

Fig. 3.1: Stages required to access various levels of the memory hierarchy

The operating system is a MACH 3.0 microkernel with a HP/UX-compatible Unix server on top. It provides the system call interface from Hewlett-Packard's Unix and an additional system interface to create and control kernel threads.

3.2.2 Scheduling Architecture of ELiTE

The overhead associated with lightweight processes goes beyond the cost of thread management due to memory transfers between the various levels of the memory hierarchy. We present a scheduling architecture outlined in [BEL95], refined in [BEL96A][BEL96B], and implemented in [STE95].

The following architectural features characterize the ELiTE runtime system:

– Division of thread control block (TCB) and stack allocation (see figure 3.2):
Each processor manages its own pool of free TCBs ① and stacks ③. If a new thread is created, the creating processor allocates and initializes a free TCB. After initialization the TCB is enqueued in a startqueue ②. A processor with an empty runqueue ④ fetches a TCB from a startqueue and can run the thread after allocating and initializing a stack. By sep-

arating TCB and stack allocation, memory objects of a thread which have to be modified, will be allocated from memory pools managed and touched by the modifying processor. The consequence is a high cache reuse and a low cache miss rate.

– Pool and startqueue hierarchy correspond to memory hierarchy:
The number of startqueue entries is limited on the first level (processor level) and the second level (node level). If an overflow occurs, the TCB becomes enqueued in the next level. The consequence is a high degree of locality with an implicit load distribution.
When the stack- and TCB-pools of the first level (processor level) are empty, new memory objects will be enqueued from the second level. Likewise, memory objects will be moved from the first level to the second level when an overflow occurs. If a pool on the second level is empty, new memory will be allocated from global memory. Therefore, global pools are not useful. The consequence of this strategy is high reuse of local memory while keeping memory allocations to a minimum.

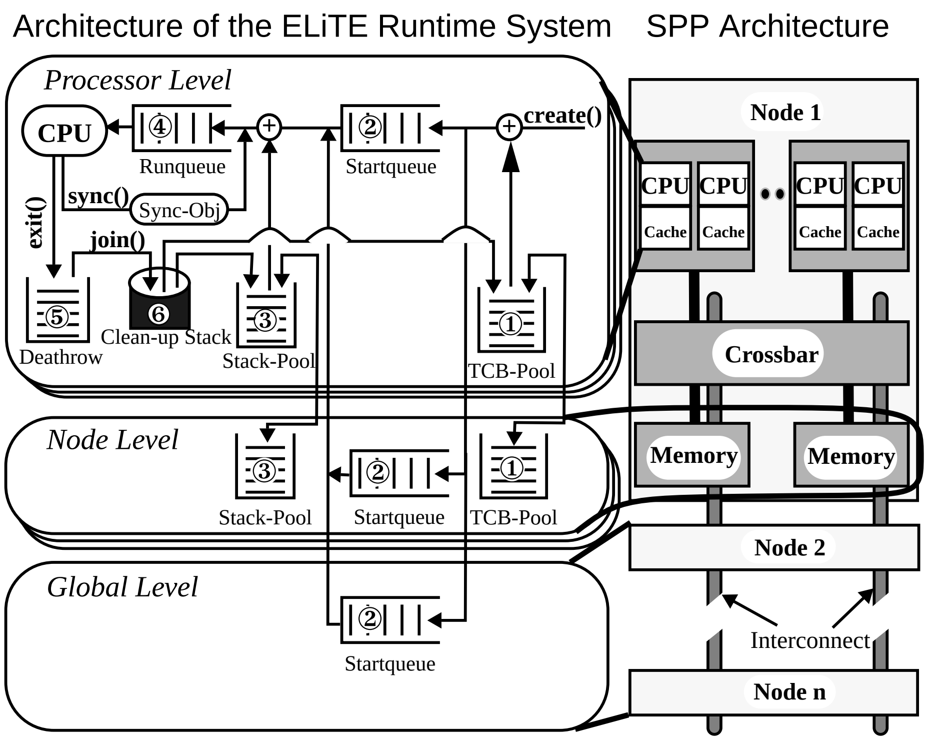


Fig. 3.2: Scheduling Architecture ≈ Memory Architecture

– Local runqueues with load balancing:
Each processor manages its own priority runqueue (see figure 3.2 ④). The priority of a thread depends on its affinity. The scheduler prefers threads with high affinity. A processor with an empty runqueue, finding no threads in the startqueues, scans the runqueues of the processors in the same node and finally the runqueues of all other processors for runnable processes. The advantages of local runqueues are high cache reuse, low data cache invalidation and minimal contention for queue locks.

– Local deathrow with local clean-up stack:
When a thread exits, its context is stored in the deathrow (see figure 3.2 ⑤) of the processor. The processor executing the join() reads the exit status of the joined thread and pushes

an entry on the clean-up stack (see figure 3.2 ⑥) of the processor which executed the exit(). The processors periodically scan their local clean-up stacks and remove the contexts of joined threads from the deathrow and push memory objects (stacks and TCBs) into the local memory pools. As processors executing a join never modify the deathrow and memory pools, cache invalidations can be avoided and the memory locality will be preserved. Cache misses are reduced to a minimum, because the push onto the clean-up stack concerns only a single cache line.

We measured the number of threads a single thread can fork and join if *n* CPUs can start and run the created threads. We compared an approach with central pools and another with distributed pools (see figure 3.3). The results show that the fork-join-rate of the centralized approach

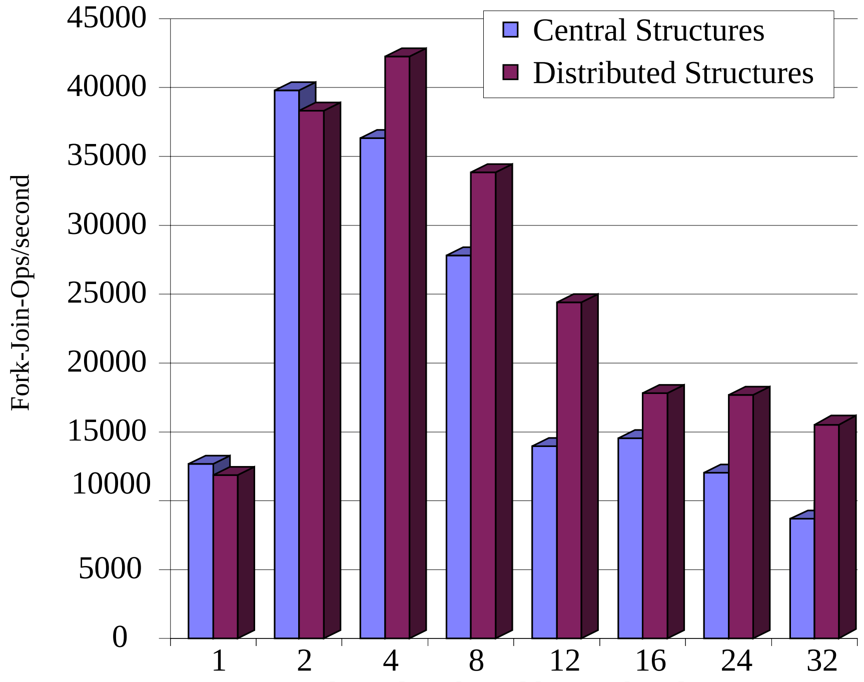


Fig. 3.3: Central vs. distributed hierarchical structures

drops when using 4-8 processors due to lock contention and dramatically drops when using more than 8 processors (more than one hypernode) due to allocation of stacks not cached on the local node. The performance degradation of the distributed approach is only moderate, because all stacks are allocated from local pools with encached memory. Furthermore, locking of central structures (pools and queues) and remote memory access can be reduced to a minimum by the mechanisms of local deathrow and clean-up stack. The distributed approach cannot scale because of the limited fork rate of the single forking thread and because of the transfer of TCBs from the forking to the executing processor. But it degrades only moderate with increasing processor numbers.

– Distributed synchronization objects with local wake-up stack:
  Unlike common UNIX sleep queues with hashed entries [LEF90][GC94], the ELiTE runtime system binds blocked threads to synchronization objects (see figure 3.4 ②). If a process becomes unblocked, a reference to its TCB will be pushed on the wake-up stack (see figure 3.4 ③).

Likewise the deathrow management, the processors scan their local wake-up stacks periodically and enqueue unblocked threads in the local runqueue (see figure 3.4 ①).

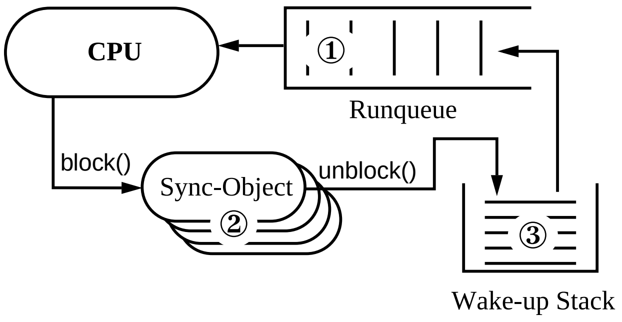


Fig. 3.4: Distributed lock structures

3.2.3 Implementation Details

3.2.3.1 Fast context switch

A fast context switch free of race conditions is the basis of most synchronization mechanisms inside a runtime system.
Context switching is delicate for race conditions on multiprocessor systems, because one processor could resume an enqueued thread while its stack is not yet completely frozen by the processor of its last run. To implement context switching, we have investigated two models:

- **Scheduler Threads**: During a switch, control is returned to a scheduler thread local to each processor. The scheduler thread enqueues a thread from the run queue and performs an additional switch to it. Races cannot occur because the freezing of a thread is performed on the stack of the scheduler. However, this simple and secure switching model is very time consuming, as two context switches are necessary per thread switch.
- **Preswitch**: After saving the state of the old thread, the stack of the new thread is used to enqueue the TCB of the old thread without the danger of a race condition. This mechanism assumes that the next thread is known and existent before the switch occurs and that the next thread already owns a stack, which makes lazy stack allocation difficult.

As switching efficiency is essential for a fast runtime system, preswitching is used in ELiTE. Based on the QuickThreads package of the University of Washington [Kep93], which provides the preswitch model for various processor architectures, we have ported QuickThreads to the HPPA-RISC processor architecture [Red95]. On a CONVEX SPP/1000 using a PA-RISC 7100 processor, the following times for a context switch can be reported:

Operation	Clock Cycles
Context switch between threads with all data in cache	153
Context switch between threads in the same node	1122
Context switch between threads in different nodes	1805

Tab. 3.1: Context switch latency in ELiTE

The proportion for a context switch with thread control blocks in the three levels of the memory hierarchy is 153/1122/1805 = 1/7/12. These are almost exactly the proportions expected to result from a memory latency of 1/50/200 cycles and 32/(64) Bytes (network-) cache lines. Most of the time is spent saving and restoring the callee-saves registers. The consequence is that switching can only be optimized by reducing the number of registers to be saved. These are the callee-saves registers, regulated by the calling conventions (e.g. by the HP PA-RISC calling conventions). As context saving and restoring for most contemporary RISC processors (an exception is the SUN SPARC processor with its register windows) is a sequence of machine instructions and not part of the instruction set, a change in the calling conventions could make context switching much more efficient by increasing the caller-saves registers and reducing the callee-saves registers.

3.2.3.2 Fast synchronization

Lim and Agarwal [LA93] have investigated waiting algorithms for synchronization in large-scale multiprocessors. With increasing CPU numbers, the type of synchronization has a significant influence on the performance of fine-grained parallel applications. As proposed we use two-phase locking with a fixed number of spin cycles in the ELiTE runtime system.
The proposed two-phase waiting algorithm combines the advantage of polling and signalling. A thread blocks after a default polling interval. The polling threshold depends on the overhead of blocking.
We count the number of clock cycles as long as a lock is held and calculate the average duration (in clock cycles) of the last 8 times each specific lock was acquired. If the average of lock-holding cycles exceeds a proposed value (default is 50% of the cycles for a context switch), we block at once. Otherwise we spin the default number of cycles. To reduce memory accesses while spinning we use exponential backoff. For details refer to [STE95].
We have measured the peak performance for synchronization by starting 4096 micro-threads (8 kBytes stack and no workload) doing nothing but synchronizing. The total amount of memory is about 4096*8192 KBytes = 32 MBytes.
With central queues we see a severe performance loss due to lock contention and data cache corruption as a consequence of non local memory accesses (see figure 3.5).

Using a distributed approach, the time for a synchronization depends on the time to save/restore the stacks into the processor/network-caches and to access the synchronization objects. We reach the peak performance of about 97000 synchronizations/second per processor (1-8 processors in figure 3.5), if all synchronization objects reside in the same hypernode. Therefore the latency to access a single synchronization object can be neglected. Because the total amount of memory (32 MB) exceeds the size of the caches, the number of synchronizations per second is determined by the time to save/restore the thread contexts to/from main memory.
If the stacks do not completely fit into the processor-caches (12 and 16 processors), they have to be stored in the network caches (= local memory) in part. Because some of the synchronizations objects reside in remote hypernodes, the performance (20373 synchronizations/second for 12 processors and 27390 synchronizations/second for 12 processors) is determined by the share of cacheable memory and the share of remote synchronization object requests.

If the stacks can be stored in the processor caches, but the synchronization objects have to be touched in part from multiple nodes, the synchronization performance deteriorates to about 23300 synchronizations/second per processor (32 processors in figure 3.5).

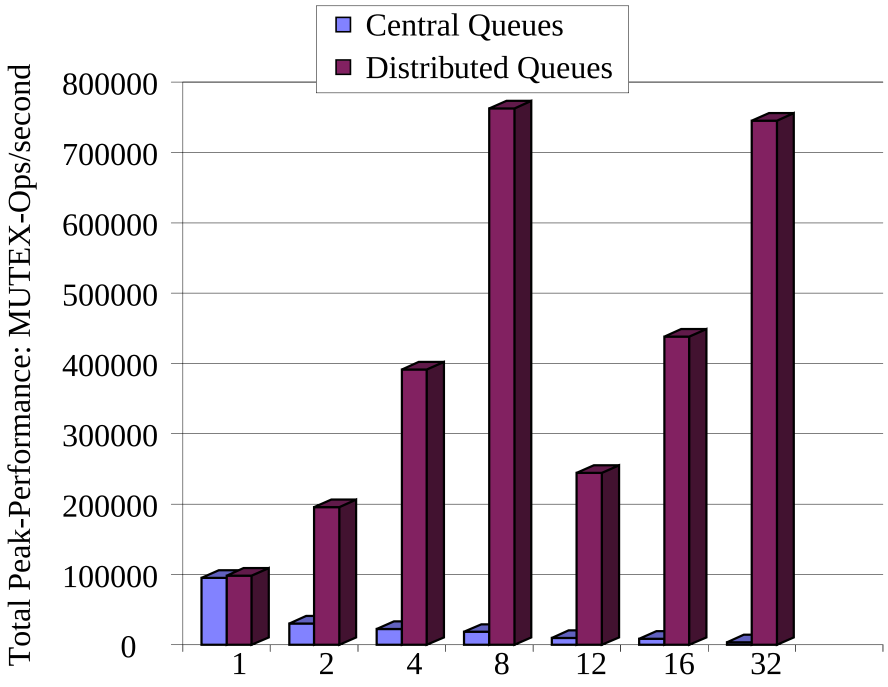


Fig. 3.5: Fast synchronization and context switch with 4096 threads

With about 57µs to access a remote memory object and 11µs to save/restore the context to/from the main memory (see the 1122 clock cycles in Table 3.1) the number of synchronizations per second can roughly be guessed according to the following equation:

$$syncs/second = \frac{\text{\# of CPUs}}{\text{\% remote syncs} \cdot 57us + \text{\% uncached memory} \cdot 11us} \tag{3.1}$$

Using the shares in Table 3.2, we came close to the number of synchronizations measured and plotted in figure 3.5)

Number of CPUs	Remote sync objects	Uncached memory
1 - 8	0 %	100%
12	66 %	62 %
16	50 %	50 %
32	75 %	0 %

Tab. 3.2: Share of remote sync objects and uncached memory

3.2.3.3 Queue Structures

The decisions of a memory-conscious scheduler depend on the affinity of the threads to a specific memory region, e.g. cache or node local memory. Consequently, threads have to be enqueued according to their affinity. Several data structures for priority queues exist [Knu73], where Fibonacci heaps and relaxed heaps [DGST88] only need O(log *#threads*) operations for the

time-critical ‘find_and_remove_maximum’-operation, which is necessary to identify and extract the processes with maximum locality from the priority queue. But heap-structures are not suitable for runqueues, because heaps cannot be partitioned fast enough in the case of load imbalance.
Priority queues implemented as binary trees enable fast de- and enqueueing and can be divided very easily into partitions with entries of high or low locality.

3.2.3.4 Application Interface

Contemporary NUMA architectures like Convex SPP or KSR1/2 have non-blocking prefetch operations in their instruction set to concurrently fetch data regions from a remote node into the local network cache, overlapping computation and network traffic and thus hiding latency. Furthermore advanced processor architectures like PA-RISC 8000 [SGH97] or Ultra-SPARC [SUN95A] offer prefetch operations to fetch data regions into the processor cache. If thread-specific data can be stored in a single block, a pointer to this block and its length can be stored in the thread control block. If there is an interface to the scheduler, a currently running thread can ask the run-time system to prefetch the data of the thread which will run in the near future. This idea was motivated by implementations of adaptive numerical methods [BEL94][RUE94], where thousands of threads, each corresponding to a point of an adaptive grid, resume the threads representing the grid points in the neighborhood after calculating the local grid point before they suspend themselves. This numerical method, called *active threads strategy* can only run with high efficiency on NUMA architectures if all thread-specific data is resident in the cache before the context switch occurs (see also the measurements of irregular applications in section 3.3.2).

3.3 Affinity Scheduling

The performance of a computer is considerably influenced by the fast supply of data to the available processing units. Out of order execution and the toleration of outstanding loads in advanced processor architectures (see HPPA 8000 [SGH97]) can delay the impact of insufficient locality, but they don’t change the fundamental. Only if the data essential for operation is cached in fast memory the processor can work without latency and contention. Affinity scheduling tries to prefer processes with a high amount of cached data in order to increase throughput.

Besides information about processor number and time behavior, we use information about data locality in our scheduling architecture. Locality information about each process/thread like the cache miss rate, the processor stall time, and the processor of last execution can be used to calculate an affinity value. A prerequisite is a computer architecture providing information about cache misses and CPU stall cycles due to memory access. Contemporary processor architectures like HPPA 8000, MIPS R10000 or Ultra SPARC gather this information on chip. As the processor (HPPA 7100) used in the Convex SPP 1000 does not offer event counters on the processor chip, these counters have been embedded in the memory interface of each CPU. The information derived from event counters is normally used for off-line profiling. Many theoretical studies involve off-line analysis [MCN+90][CL93][ZLT+96][ABD+97] and search for optimal solutions given that everything is known up front and nothing changes. However, real systems operate in a dynamic environment with unpredictable page mapping and memory allocation effects. Why should we therefore ignore information for optimizing the behavior of multithreaded applications at runtime to analyze and influence the execution?

In the next subsections we describe several affinity strategies and the prospect of the proposed technique.

3.3.1 Scheduling Strategies

We have designed and implemented a user-level runtime system, offering the possibility of easily importing new affinity strategies. The strategies examined are listed in the table below:

Scheduling strategy	Basis of decision	Policy
No Affinity	Processor location	LIFO
Virtual Time	Sequence numbers	Most recently run
Minimum Misses	Cache misses	Thread with the minimal number of cache misses
Cache Miss Sum	Cache misses	Thread with the minimal sum of cache misses (with aging of values)
Reload Transient	Cache misses	Minimal reload transient (Markov chain model)

Tab. 3.3: Scheduling strategies

- Using *virtual time* stamps, each thread is assigned a sequence number after its run. This strategy does not need cache miss information and can be used on every type of hardware. Threads with the highest time stamp given by the same processor will be preferred. If a fast global time source with high resolution is included in the hardware, more precise timing strategies can be used [BB95].
  This strategy does not take into consideration the memory access behavior of a thread, neither the locality of reference nor the size of the working set.
- The *Minimum Misses* strategy compares the number of cache misses during the last run. The thread with the lowest number of cache misses is preferred. This strategy favors threads that block frequently and those with a high locality of reference.
  Minimum Misses does not take into consideration the size of the working set. Furthermore, a thread with a high number of start-up misses is assigned a low priority when it is deblocked the first time. Because this thread is never preferred, it can only seldom find his working set in the cache. Whenever it runs, it has to fill the cold cache and therefore it will never gain a high priority.
- The *Cache Miss Sum* strategy cumulates the number of cache misses of each thread. To prevent long running threads from starvation, an aging strategy continuously reduces the

sum of cache misses with each scheduling operation. The thread with the lowest sum of cache misses is preferred.

This strategy does not take into consideration the size of the working set. However, the start-up anomaly of *Minimum Misses* is avoided, because the history buffer of a thread is pre-set with zero misses. Consequently a thread will gain a high priority in the first scheduling cycles after the start-up and will continuously adopt his cache miss sum to the real value.

- The *reload transient* model is more complex, but offers some potential. We refer to the working set of a thread that is present in the cache as its *footprint* in the cache. The reload transient is defined as the cost to establish the footprint of a thread after restarting it. Our scheduling policy selects the thread with minimal reload transient.

  Thiebaut and Stone [TS87] found a simple analytical model for the estimation of the influence of cache size on the reload transient. Their reload transient depends on the cache size and on the sizes of the footprints of competing threads. Simulations based on address traces show an excellent agreement between the model and the observations.

  Because it is impossible to gather and evaluate address traces for scheduling decisions online, we had to find a fast and easy way to guess the sizes of the footprints of all threads having some data stored in a cache. Our solution is the evaluation of cache-miss information gathered in the event-counters. Just like Thiebaut and Stone we assume that any cache line is equally likely to be the destination of a memory reference.

  We developed a Markov model to calculate the footprint of each thread. In the state transition diagram in figure 3.6. each node *V* represents a state with *v* valid cache lines of a thread residing in the cache. Our direct mapped cache consists of *N* cache lines.

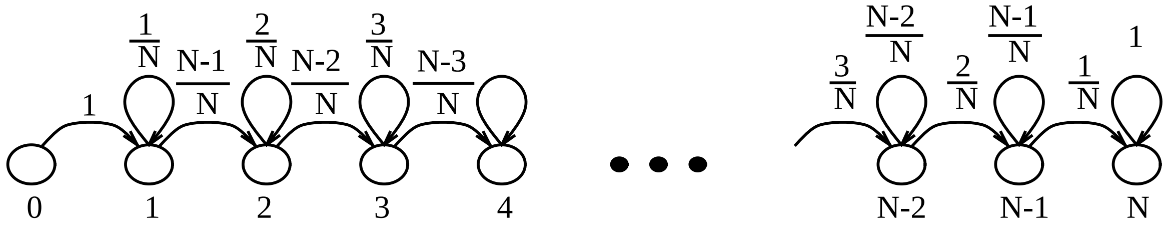


Fig. 3.6: Running threads increase the number of valid cache lines

The probability to increment the number of valid cache lines as the consequence of a cache miss during the run of a thread is $(N - v)/N$. The probability that a cache miss hits a valid cache line is v/N. We can generate the transition probability matrix *P*.

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & \dots & & 0 \\ 0 & \frac{1}{N} & \frac{N-1}{N} & 0 & 0 & \dots & & 0 \\ 0 & 0 & \frac{2}{N} & \frac{N-2}{N} & 0 & \dots & & 0 \\ \vdots & & & \ddots & \ddots & & & \vdots \\ & & & & & & & 0 \\ 0 & & & & \dots & 0 & \frac{N-1}{N} & \frac{1}{N} \\ 0 & & & & & \dots & 0 & 1 \end{bmatrix} \tag{3.2}$$

Each element $p_{i,\,j}$ is the probability that a cache miss increases a thread's cache state from *i* to *j* cache lines. Raising the matrix to the n-th power gives the probability for an increase from *i* to *j* after *n* cache misses. Using these probabilities we calculate the expected footprint size *F* after each run. The size of the footprint depends on the number of cache lines *v* still valid at startup and the number of cache misses *n* which occurred during this run $E[F|V = v, M = n]$.

$$F^n{}_v = \sum_{j=0}^{N} j \cdot p^n{}_{v,\,j} \tag{3.3}$$

Figure 3.7 demonstrates the growth of the footprint depending on the number of cache misses and the number of valid cache lines for a cache with 64 cache lines. The plot looks similar to other cache dimensions. With increasing number of cache misses the footprint converges to the size of the cache. If there are many valid cache lines (>32 ≈ 50% cache size), the footprint increases very slowly. If there are only a few valid cache lines (<16 ≈ 25% cache size), there is a linear growth of the footprint after the first cache misses (see the zoomed plot in figure 3.7). A thread with a small working set can establish its footprint with a modest number of cache misses, whereas a thread with a large working set needs many cache misses to establish a reasonable fraction of its working set as its footprint.

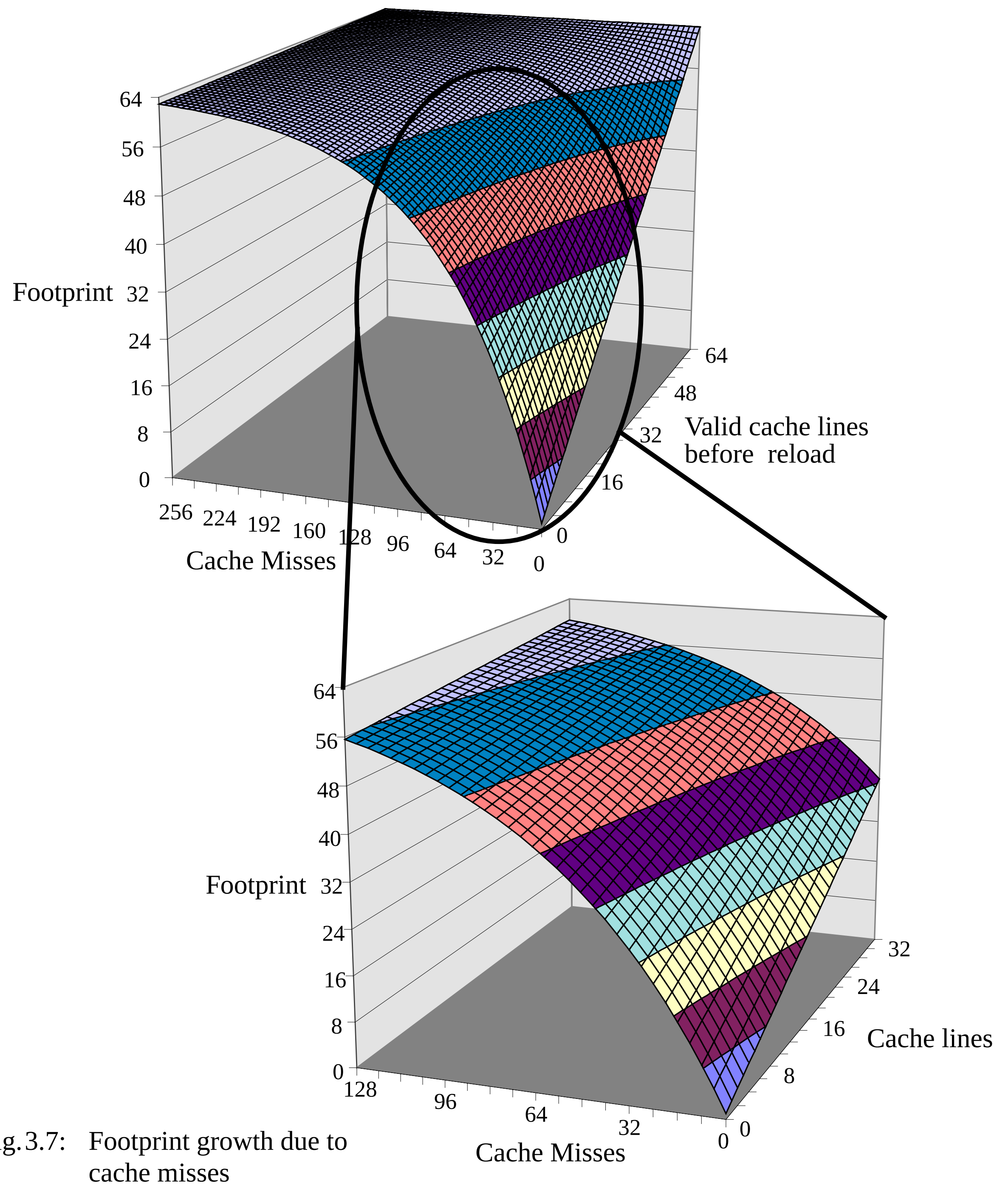


Fig. 3.7: Footprint growth due to cache misses

Equivalently, we can calculate the transition probability matrix for a blocked thread. The number of resident cache lines is decremented with a certain probability for each cache miss caused by the intermediate run of an other thread. The probability to decrement the number of valid cache lines as a consequence of a cache miss during the run of an other thread is v/N. The probability that a cache miss does not hit a valid cache line is $(N-v)/N$ (see figure 3.8).

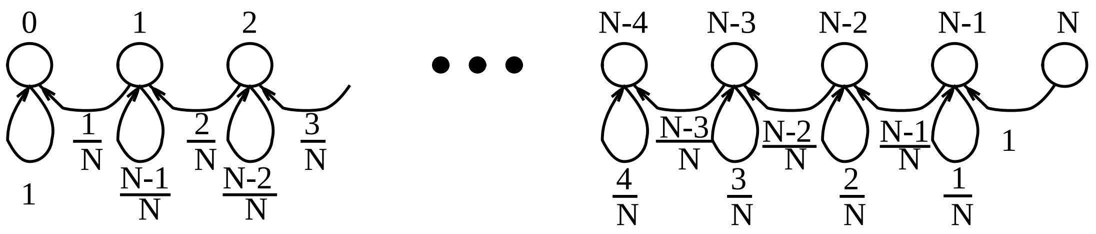


Fig. 3.8: Stopped threads lose cache lines due to cache misses of running threads

Analogous to the calculation of the footprint we can also compute the expected number of remaining lines *V* which are still valid depending on the footprint *f* of a specific thread and the number of cache misses *n* caused by other threads $E[V|F = f, M = n]$.
Figure 3.9 demonstrates the displacement of the footprint depending on the number of

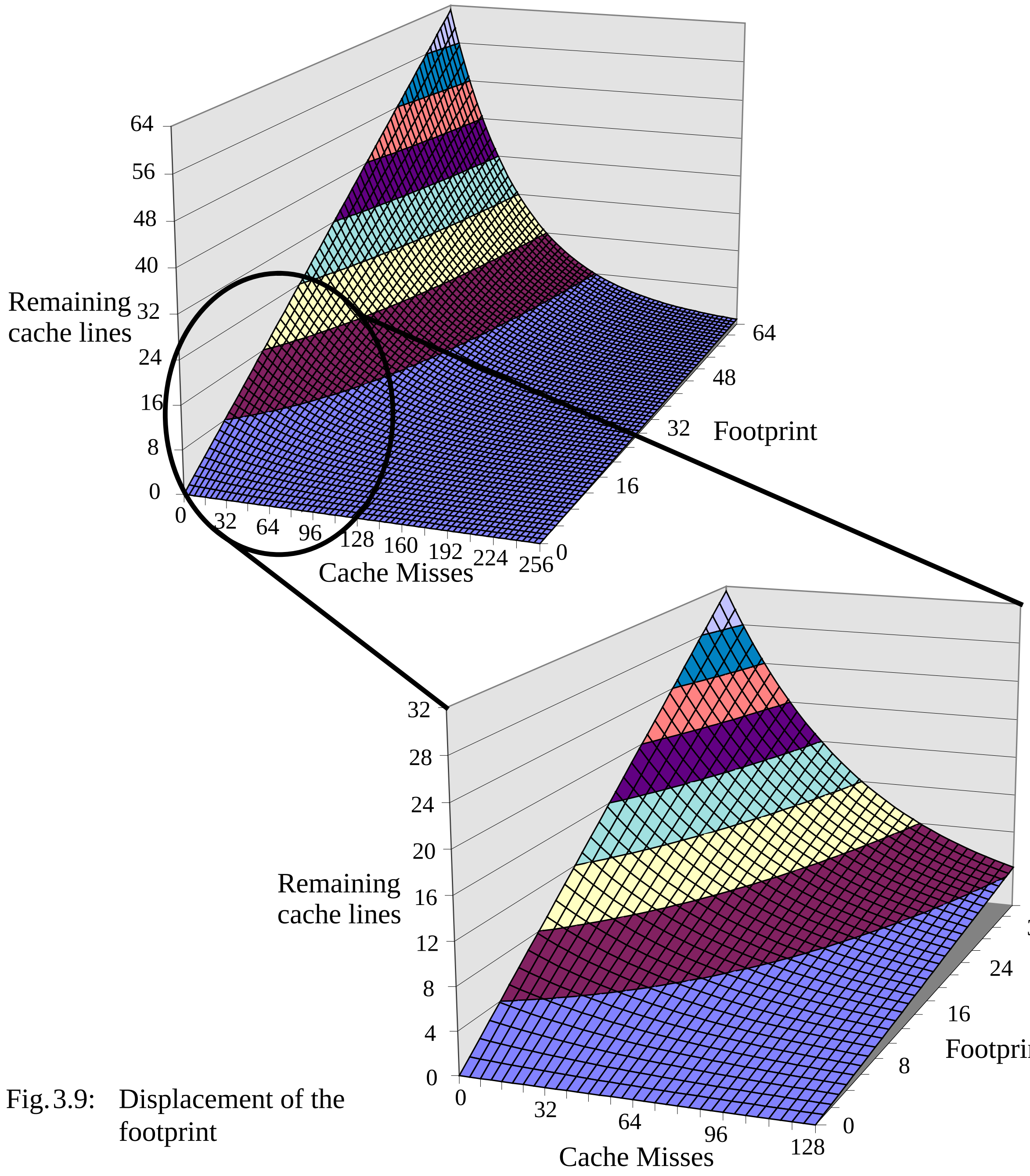


Fig. 3.9: Displacement of the footprint

cache misses of other threads and the number of valid cache lines before the thread blocked. With increasing number of cache misses the footprint is totally displaced. Threads with a large footprint ($>$ 32 cache lines $\approx$ 50% cache size) are heavily displaced, whereas a small footprint ($<$ 8 lines $\approx$ 12% cache size) has a good chance to keep a large fraction of its footprint (see the zoomed plot in figure 3.9). As we focus on a high cache reuse in the presence of multithreaded applications, we have to restrict our investigations to user-level threads with a working-set size of less than 10% of the cache size.

Basis of our cache affinity calculation is the expected footprint size F of a thread in its last run and the expected number of valid cache lines V before its potential run. The reload transient is defined as the expected number of cache misses $E[M|F = f, V = v]$ when rebuilding the working set of a rescheduled thread. First we determine the remaining cache lines due to displacement. The reload transient is the number of cache misses n in equation 3.3 so that the footprint is re-established from the remaining cache lines.
The plot of the reload transient (see figure 3.10) demonstrates the advantages of the strategy. Because they only need a few cache misses to re-establish their footprint after a certain time of blocking, threads with a small footprint run with higher priority than those with a large footprint. Otherwise, threads with a larger footprint are preferred, if they only blocked for a short period of time (see the zoomed plot in figure 3.10).
Our scheduling policy selects the thread with minimal reload transient. A characteristic of our approach is that the order among runnable threads remains the same even if the scheduler enqueues new runnable threads. Consequently the reload transient has only to be determined when a thread is deblocked and enqueued. By pre-computing expected reload transients for a set of footprints and a set of different cache miss numbers the priority calculation of the scheduler can be reduced to a simple table lookup [TTG95].
Two assumptions can cause a difference between the predicted reload transient and the reload transient a thread experiences:

- Our model assumes that each cache line of a thread that was displaced has to be reloaded when the thread resumes. This is not always true. A cache line that was displaced can very well be a *dead* line which will never be accessed for the remainder of the threads execution. Beside the *live* lines our approach takes the *dead* lines into consideration to calculate the reload transient. Consequently the reload transient in reality should be smaller than our prediction.

- We assume that any cache line is equally likely to be the destination of a memory reference. In reality many references are clustered. Clustered references reduce the size of the footprint because the probability of a conflict miss is increased. Otherwise clustered references may reduce the loss of valid cache lines because of fewer collisions between the footprints. Both effects of clustering can influence the reload transient. They should be analyzed in detail in future work.

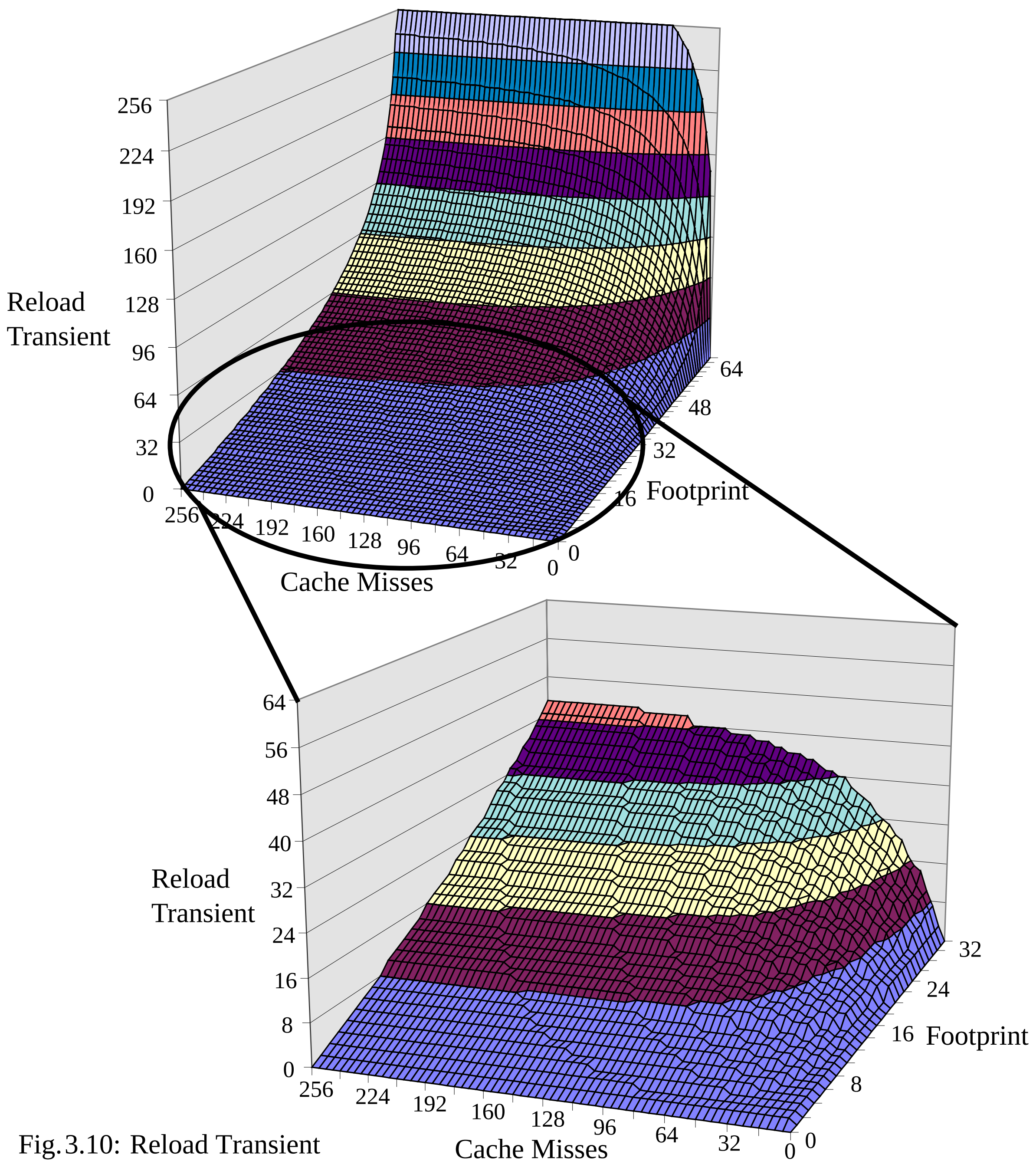


Fig. 3.10: Reload Transient

3.3.2 Interpretation of Measurements

We have implemented and tested the proposed strategies as part of the ELiTE scheduling architecture on a Convex SPP 1000/XA in a range from 1 to 32 processors. The test environment includes synthetic tests with artificial workloads as well as real-world numerical kernels like Gauss elimination, LU-decomposition and adaptive solvers on irregular grids [BEL94][RUE94]. All measurements show that a fine-grained parallelization does not inherently imply low performance. On the contrary, a fine-grained numerical efficient algorithm outperforms most conventional methods, because fine-grained parallelism implies a high data locality in most cases. This locality can be used by a sophisticated scheduler to achieve good overall performance even if we register some overhead for complex strategies. Consequently the impact of scheduling on a fine-grained environment affects performance much more than with a coarse-grained approach. With a synthetic workload of 1024 threads, each snooping through a working set of 64 kBytes (≈ 6 % cache size) between synchronizations, we can get a good impression of affinity strategies. We measure how many synchronizations can be executed in one second with different numbers of processors (see figure 3.11). This example resembles parallel numerical applications with regular execution order like iterative solvers on block-structured grids.

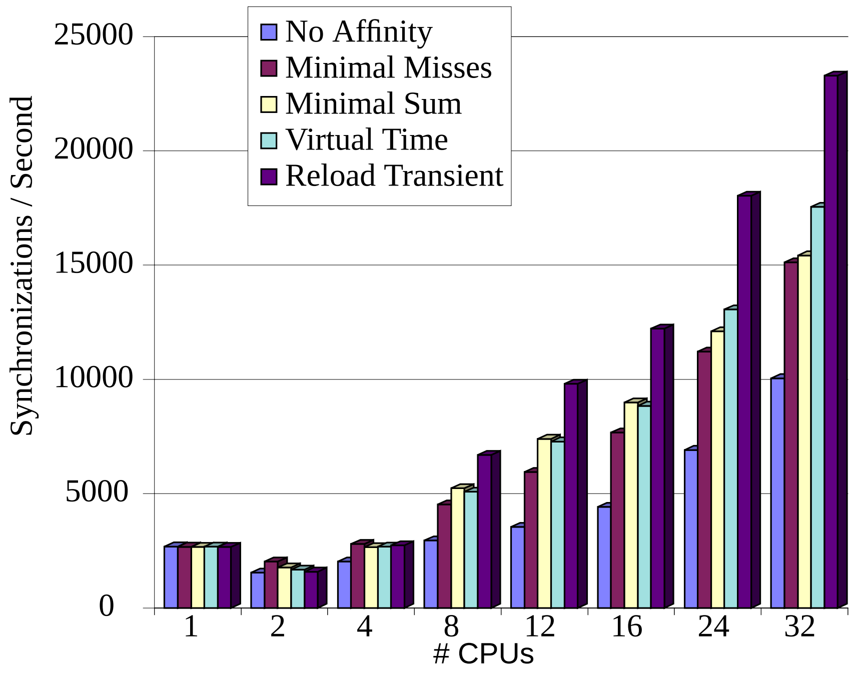


Fig. 3.11: Performance of applications with regular execution order

- *No Affinity* will be outperformed by all strategies using locality- or time-information in configurations with more than 2 CPUs.
- The simple *Minimal Misses* and the *Minimal Sum* strategy perform quite well in cases with homogeneous execution behavior. As these strategies favor threads that block frequently, anomalous behavior is possible if some threads acquire locks during the polling interval whereas other threads block. *Minimal Sum* performs slightly better than *Minimal Misses* because the start-up anomaly is avoided and the effect of lock acquirements during polling phases is smoothed.
- *Virtual Time* performs equivalent to *Minimal Misses* and the *Minimal Sum* for single or double hypernode configurations (1-16 CPUs). The more CPUs, the higher is the probability that two threads willing to synchronize run in parallel. Threads frequently acquire locks during the polling interval in these situations. *Virtual Time* does not lower the affinity value of those threads. If the memory reference patterns of all threads are regular, time informations corresponds to cache miss information. Therefore *Virtual Time* performs better for multinode configurations than the simple strategies using cache misses.

- The *Reload Transient* strategy shows the best performance for applications with regular execution order. Because the execution order as well as the data structures are very regular, the Markov approach calculates the overhead very precisely to re-establish the working-set independent of blocking or non-blocking synchronizations. Selecting the thread with minimal reload transient seems to be the best policy to keep the number of cache misses as low as possible.

In order to investigate the relation between cache related events and the speed of execution we have measured the number of cache misses per synchronization event (see figure 3.12). There is a correspondence between the speed of execution and the number of cache misses. The more the gap between CPU speed and memory latency widens, the more the speed of execution depends on the occurrence of cache misses.

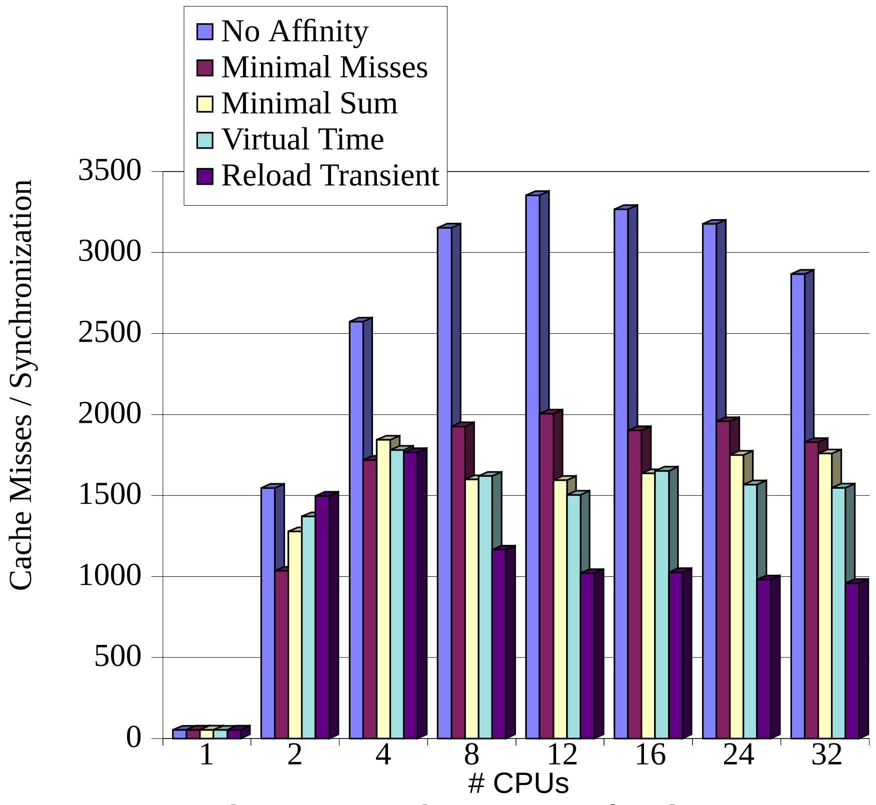


Fig. 3.12: Cache misses/synchronizations of applications with regular execution order

To definitely prove that the speed of execution can better be predicted and influenced by looking at cache miss information than by looking at time information, we investigate an application with irregular execution order. Several thousand threads (5000 in the example in figure 3.13), each corresponding to a point of an adaptive grid, resume the threads representing the grid points in the neighborhood after calculating the local grid point before they suspend themselves. If a grid partition has to be smoothed, the corresponding threads become runnable (The black spots in figure 3.13 represent the active threads in several stages of the calculation). There is no activity in a relaxed region of the grid. This parallel smoothing algorithm called *active threads smoother* terminates if there are no more runnable threads. The implementation of the *active*

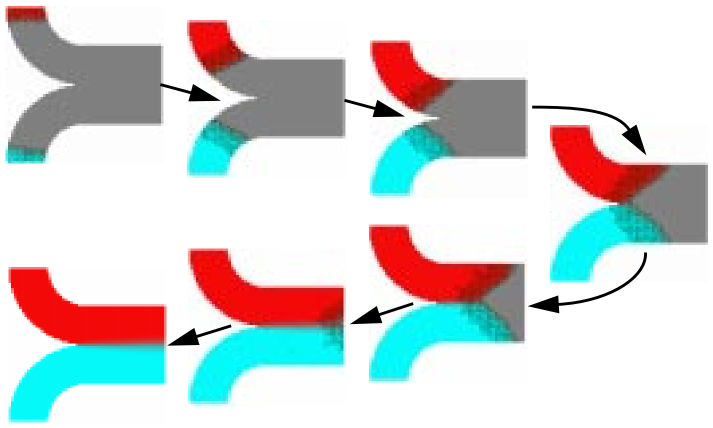


Fig. 3.13: Active Threads Smoother

threads smoother is simple and offers high parallelism. But it can only run efficiently if a high locality of reference is guaranteed. The challenge of *active threads* to the scheduling lies in the irregular execution order, the frequent synchronization- and suspend/resume-operations, and the irregular data types of grid points linked by pointers.

We measure the number of smoothing operations per second on an unstructured grid executed by a full-adaptive iterative solver [BEL94][RUE94]. To demonstrate the influence of the scheduling strategy on the application performance, we compare the smoothing rate of the proposed affinity strategies with the *No Affinity* strategy. A relative smoothing performance of 2 means that the adaptive solver executes twice the number of smoothing operation under the affinity scheduler compared to an execution under the *No Affinity* scheduler (see figure 3.14)

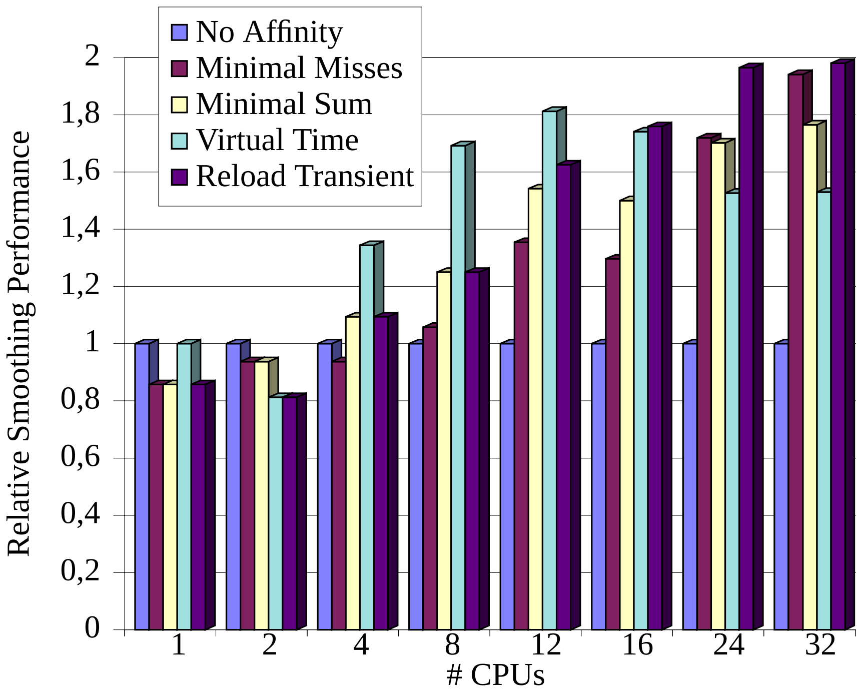


Fig. 3.14: Performance of adaptive applications with irregular execution order

- *No Affinity* has no additional overhead for the calculation of affinity values. With one or two processors under high load, complex affinity considerations offer no benefit. Therefore the simple LIFO strategy is sufficient.
- The *Virtual time* approach only uses timing behavior and the processor number of the last run. It never shows anomalous behavior and performs very efficiently with moderate processor numbers (1-16 processors) because it offers the minimal overhead compared to the other strategies. It should be the policy of choice for UMA architectures not offering cache miss information (see [BB95]).
- The overhead of the *Minimal Misses* and *Minimal Sum* strategy does not outweigh the benefit of saving some cache misses for moderate parallel configurations (< 20 CPUs). While *Minimal Misses* suffers from the anomaly in the presence of blocking and non-blocking synchronization operations, *Minimal Sum* is not able to adopt the affinity values in a high dynamic environment due to its sluggish aging strategy. Both strategies perform better than the timing based strategy in a multinode configurations (>20 CPUs).
- The *Reload Transient* strategy shows the best performance in multinode architectures with non-uniform memory access. The overhead of gathering cache miss information and computing the expected working sets is only justified when memory latency really strikes. This is the case on all contemporary and future scalable parallel processors like Convex SPP, KSR 1/2, SGI Origin, Sequent NUMA-Q and multiprocessors coupled with SCI-Hardware (SCI = Scalable Coherent Interface).

A final look at the speedup-curve (see figure 3.15) demonstrates that an efficient parallel execu-

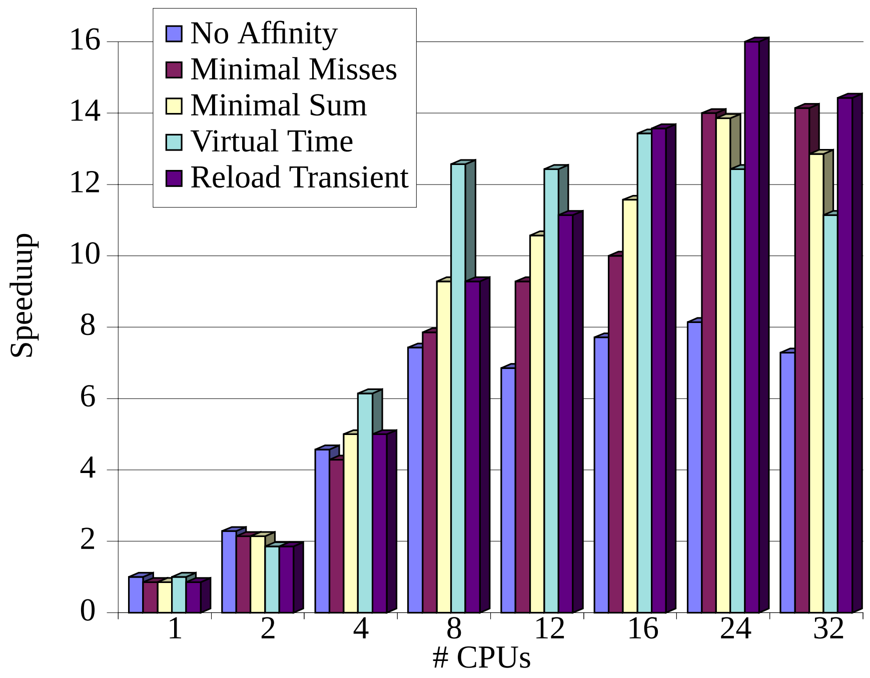


Fig. 3.15: Speedup of adaptive applications with irregular execution order

tion is only possible if the scheduler has the choice between multiple threads to select those with the best cache affinity. In the example above only 5% of the grid points are smoothed concurrently (see the the black spots in figure 3.13). If some of the active threads stick in blocking synchronization calls, the number of runnable threads can decline to a few hundred. With 32 CPUs, we have only a hand's full of threads per CPU. This is not enough for efficient affinity decisions. In this situation an advanced scheduler should detect that the system has gone beyond the optimal operation point and should lower the number of running kernel threads. A sophisticated processor allocation strategy to deal with the optimal operation point for UMA architecture has been discussed in [Tuc93]. It could be applied to NUMA architectures as well.

3.4 User-Kernel Interaction

A lightweight thread generally executes in the context of a middleweight or heavyweight thread. Specifically, the threads library schedules lightweight threads on top of middleweight or heavyweight threads which in turn are scheduled by the kernel on the available physical processors. Such a two-level scheduling policy has some inherent problems:

– User-level runtime systems using kernel threads as virtual processors assume an equivalence of physical and virtual processor. This assumption does not hold, because events like page faults, I/O and system calls block the virtual processors. As a result, the application library cannot schedule a thread on a "just idle" processor.

– When the number of runnable kernel-level threads in a single address space is exceeding the number of available processors, user-level threads built on top of kernel-level threads are actually scheduled by the kernel's thread scheduler which has little or no knowledge of the application's scheduling requirements or current state.

The problems with multi-level scheduling arise from the lack of information flow between different scheduling levels.

– Anderson et al. in [ABLL92] attempt to solve these problems for two-level scheduling *by* explicit vectoring of kernel events to the user-level scheduler using upcalls called *Scheduler Activations* to notify the kernel of user-level events affecting processor allocation. *Scheduler Activations* use kernel threads to upcall the runtime system. This strategy suffers from the fact that a free processor is needed to run the kernel thread upcalling the user level. But there is no free processor in the case of a request for suspension of a virtual processor. The consequence is an expensive context switch on kernel level causing TLB misses and data cache corruption.

– In [LMMS91] communication mechanisms between the kernel and a user-level thread library are proposed to reduce the performance losses when threads block in the kernel or are preempted in critical sections. This set of kernel mechanisms (incorporated in the Psyche operating system) to implement *first-class user-level threads* includes shared kernel/user data structures (for asynchronous communication between the kernel and the user), software interrupts (for events that might require action on the part of the use-level scheduler), and a scheduler interface convention that facilitates interaction in user space between dissimilar kinds of threads. The kernel and the threads package communicate using shared memory whenever possible to avoid the need for synchronization interaction. Software interrupts signal to the thread package whenever a scheduling decision may be required. For example, polling of shared memory in a safe suspension point is used to instruct the runtime system to suspend a thread, while signalling is used to inform the runtime system that a thread can be resumed or a new kernel thread can be created. Signalling is used to prevent idling of a processor while information exchange over shared memory is used whenever quick response to events is not so important.

– A strategy offering fast response to blocking events is proposed in [Kop95] and is used in ELiTE. The runtime system parks spare kernel threads in the kernel. In case of a blocking call, the kernel deblocks a parked thread to maintain a fixed number of running kernel threads. When the blocking request is resolved, the kernel informs the runtime system of the deblocking via a shared page or shared-memory segment. If this deblocked user-level thread is selected for execution, the corresponding kernel thread initiates a system call to park in kernel again and to release the blocked kernel thread. The system is in the same state as before the blocking call.

3.5 Conclusions

Algorithmic optimizations of the application and scheduling mechanisms for the management of parallelism determine the overall throughput. The applications designer cannot be relieved of algorithmic considerations concerning memory locality, but he can benefit from a scheduling strategy which makes a fine-grained architecture-independent programming style possible thanks to its efficient memory-conscious thread management.

As maximum throughput is the goal of our efforts, we have presented the architecture of the Erlangen Lightweight Thread Environment (ELiTE). The focus of this scheduling architecture lies on the reduction of cache misses. Distributed data structures like those proposed in the ELiTE architecture are an absolute necessity. Scheduling strategies using locality information improve cache locality and therefore throughput. Strategies based on Markov chains offer the best process reordering in scalable architectures with non-uniform memory access despite their algorithmic overhead. The hardware of the Convex SPP 1000 used for our implementation cannot distinguish between processor- and network-cache misses, but these two types of cache miss differ by a factor of 4 in their penalty. Coming SPP generations offer information about both types of cache fault. This feature permits a much better calculation of the working set and allows strategies which will clearly outperform all other proposed affinity techniques.
The trade-off between scheduling overhead and performance gain due to better locality will favor complex strategies using cache miss information particularly in architectures with high memory latency and large caches. Consequently, the proposed scheduling techniques can be used from the high-end desktop workstation to the supercomputer.

4 Soft Real-Time Scheduling

One of the most challenging problems in modern operating systems is to design allocation strategies for shared resources which are critical for applications with predefined service rate objectives (e.g. interactive-, multimedia- or hard real-time demands). To meet the needs of those applications, the resource management has to allocate a share of a resource to an application and has to ensure that the allocated share does not drop below a specified bound.
In a *hard real-time system* all delays are bounded, from access-time of data and code up to the execution time and the frequency of requests. Any layer of abstraction reduces the predictability of a system concerning timing behavior. Thus hard real-time systems do not exhibit the same functionality as advanced operating systems.
Soft real-time systems are less restrictive. Applications can request a quality of service (QoS). If such a request is granted, the operating system tries to provide a certain QoS [MST93][MST94]. Those systems can be found in the area of multimedia, hand writing recognition or robotics. They need operating-system features which cannot be supported by hard real-time systems. But they have time-critical tasks which have to be executed with a high priority.

This section is dedicated to resource allocation strategies in the field of soft real-time scheduling. We describe the limits of current systems and sketch solutions for common problems. Special focus lies on the effect of memory preemption in shared-memory multiprocessors. We present a solution for this problem which is rapidly emerging in advanced computer architectures.

4.1 The Neglected Resources of Real-Time Scheduling

Resource management is a core functionality of each operating system. Beside the virtualization of resources it is essential to provide a sufficient share of critical resources to applications that need guaranteed performance. This issue has become more popular with the upcoming of multimedia applications (video-conference systems, internet phones, movie players or speech recognition) on affordable multi-purpose workstations and personal computers. These applications need flexible algorithms which adopt the resource allocation to the dynamic requests of competing clients in an environment in which real-time and normal time-sharing applications can run concurrently. The functionality of a system supporting real-time demands depends on the limitations of the hardware platform and the capabilities of the resource management algorithms.

If we take a look into the technical specification of a typical workstation we can register the following components:

(1) Processor(s)

(2) Main memory

(3) Disk devices

(4) I/O devices like keyboard, mouse, display, sound processor, and interfaces

(5) System interconnect (e.g., motherboard with active components including memory- and cache-controller)

(6) Caches

(7) Power-supply and cooling system

All components have an essential impact on the system performance, but only the components 1-4 have been managed by real-time operating systems in the past.

Typical resources that contemporary real-time operating systems know to deal with are

- processing units:
  The operating system can partition the time, a processing unit is assigned to a specific task. Timer interruption and preemption assures a kind of protection so that a single task cannot occupy the processing unit requested by other tasks. Scheduling policies like fixed priority scheduling [GC94], earliest deadline scheduling [JM74], rate monotonic scheduling [LSD89], or proportional share scheduling [SAW95][WW95] assign runnable threads to the processing units to provide a specific quality of timely services.

- main memory:
  Usually memory is shared in space. Contemporary real-time operating systems can ensure that the allocated share in memory related to a performance critical task cannot be displaced and that pages or segments cannot be moved to persistent storage in the presence of virtual memory [GC94]. Without locking of memory the latency to access memory is unpredictable because the memory region has to be fetched from an external device when a page- or segment-fault occurs.

- Disk devices:
  Disk devices serve device requests in turn. The time to service a request depends on the availability of disk blocks in the device cache, the time to transfer a block between the disk and the controller and the load of the disk channel. There is an upper bound for the time to service a request. Applications normally submit requests to the device driver which tries to optimize the sequence of requests in order to draw the best performance out of the

disk device [GC94][SG94]. In today's operating systems the device driver does not have any knowledge of the priority of the submitting task. Consequently the requests submitted by low- and high-priority threads can interfere, thus resulting in an unpredictable service time.

The device driver can reorder the requests in priority queues, if each request for a disk operation is coupled with the priority of the submitting task. A high priority task has an upper bound for the service time of its requests. But the rapid execution of a high priority request has to be paid with a decrease in efficiency as the full set of disk optimization and buffering strategies cannot be applied.
Disk devices are a typical example that an increase in predictability has to be paid with a loss of efficiency.

- Input/Output Devices:
  Likewise the CPU, input/output devices that cannot be shared in space (e.g., display, sound processor or network interface) can be shared in time by real-time threads. Because of the complex internal state of some devices (e.g., display controllers) low priority tasks can only be preempted at secure points. The consequence is a reduced predictability for the service time.
  Some devices like network interfaces offer the advantage that packets of low-priority tasks can be dropped to allow high priority requests to be serviced by the network interface. Threads that suffer from dropped packets have to retransmit the packets. Typical scheduling strategies for network devices are weighted fair queuing or priority queuing [Cis97].
  Other devices like frame grabber cards are normally assigned exclusively to a specific task and therefore have no real-time demands.

From the beginning of operating system design the resources mentioned above determined the performance of computer systems. But today's computer architectures have far more critical resources that are not covered by contemporary operating systems. But these forgotten resources have an impact on system performance that will increase continuously. In the next future the operating system community should therefore pay more attention to the management of these resources which were neglected in the past.

The management of the chip set, the caches and the power-supply were more a topic of overall system setup than of individual control related to the currently running task. In the next subsections we prove the necessity for dynamic management of those resources and demonstrate policies and mechanisms for resource management in a real-time environment. A special section is dedicated to the management of the shared resource of main memory with its limited bandwidth.

4.1.1 Caches

Most contemporary processor architectures arise from the von Neumann approach [BGN46][Gol80] predominantly invented by Eckert and Mauchly [Mac94][HP96]. A characteristic of these architectures is the feature that code as well as data is stored in the main memory of the computer. From there arise bottlenecks which have an emerging impact on the performance. As code and data have to be transferred from the main memory into the central processor, the latency of memory access determines the maximum speed of the processing unit. To bridge the increasing gap between fast processors and slower main memory, a memory hierarchy can be established, where the most frequently used memory contents are stored in faster memory. Each level stores a subset of memory that is stored in the levels below. The caches that we use for our investigations are placed between the main memory and the registers of the CPU. Caches are partitioned in cache lines of fixed size. The mapping and replacement strategy determines which cache line is displaced and when a new line has to be stored.

For uninterrupted threads, the speed of execution is reduced by capacity misses and conflict misses. *Capacity misses* occur if the requested data has been accessed before, but the CPUs working set size exceeds the cache size. *Conflict misses* occur if the requested data had been in the cache but was displaced by an intervening reference to another address. For a system specific mapping the number of conflict misses varies, if the memory management can allocate memory from various locations (see Chapter 2.2.5.2 *Page Assignment and Page Re-Coloring*). Variant conflict misses impose unpredictable execution time behavior and therefore are a handicap in soft real-time systems.

In a real-time environment the system has to respond quickly to external events. In order to respond to an event that was normally signalled by an interrupt, the system has to switch very fast from one task to another. Immediately after the switch, the system has to work on the event with maximum speed to provide a low interrupt-service latency. It is important to know what the costs of a context switch are in order to predict the interrupt latency of a system. But the costs of a context switch are above those associated with operations performed in the switching routines of the operating system. *Compulsory misses* occur if the instructions and data of the interrupt service routines may no longer be in the cache (see also Chapter 2.2.1 *The Impact of Scheduling on Caches*). The number of compulsory misses depends on the sequence of execution in the history. Compulsory misses can be reduced by software-initiated prefetch operations [Che93][Bia95][SGH97] if the following thread is known in advance. But prefetching has only a limited benefit because only a subset of the working set like the stack or control blocks can be prefetched. Most elements of the working set are widely spread over the address space and cannot be transferred with a few prefetch operations. Furthermore the sequence and the resulting number of compulsory misses is unpredictable with unforeseen interrupt threads that are launched by I/O controllers.

To reduce the unpredictability of cache memory systems while exploiting the speedup of caches, two approaches have been considered:

- The speed of execution is measured while the caches are switched off:
  This kind of worst case analysis works fine for write-through caches, as the number of read/write memory-operations for a code segment that was analyzed without caches cannot be exceeded when using the caches.
  But in the case of write-back caches, a read or write operation related to one task might

imply a write-back operation related to a previous task. This introduces another point of unpredictability and therefore reduces the value of measurements without caches.

- The cache is partitioned among competing classes of tasks:
  Cache-partitioning techniques have been invented to draw the best benefit out of large caches used in advanced computer architectures. Depending on the mapping of memory-addresses to cache lines the memory is partitioned in a way that the partitions are never mapped onto the same cache area. A memory partition is dedicated to a class of threads. Consequently, different classes of threads cannot interfere when using the caches. On systems with physically indexed caches, the cache partitioning can even be hidden from the application, as the physical address-space is partitioned, but not the virtual address space. The operating system makes this feasible by assigning physical pages of disjunct colors to disjunct classes of threads. Recent results show that the speedup of caches can be exploited without significantly sacrificing predictability and performance [MUE95][LHH97].

In conclusion we have to state that there is no universal approach that can hide the runtime effects of caches while preserving their benefits. The motivation of caching lies in the hope to speedup the execution by exploiting information of the past. As the speedup cannot be predicted, caches have to be considered of reduced value in real-time operating systems. Approaches remain that reduce the unpredictability of caching systems running in uncritical environments.

4.1.2 Power

The objective of real-time scheduling is to provide timely services. This includes that a service can be fulfilled at some point in the future. At this point environmental conditions have to be established such that the service can be provided. Power supply and cooling are an environmental condition that was assumed to be available independent of the device operations. But this assumption does not hold for portable devices with a limited battery-power and passive cooling-capacity. Therefore power is an essential resource which becomes an emerging impact with the upcoming of power sensitive devices requesting timely service rate objectives (e.g. in hand writing recognition on personal digital assistants).

The power that integrated circuits need for operation is proportional to the number of the gates and the frequency of the clock. From the high power consumption arises the problem of power supply, power dissipation, and cooling. A high performance processor consumes between 26W (UltraSPARC-II at 250Mhz) and 60W (alpha 21264 at 300 MHz). Large external caches need power in the same order as the corresponding CPU. The chips of a typical single processor system (CPU, caches, chip set, memory) consume more than 100 Watts. Facing the trend of rising clock frequency and chip complexity, two questions come up. Can future systems be supplied with enough power (e.g., in notebook computers or PDAs) and can the power be dissipated by cooling elements and fans? Is it possible to reduce the power consumption under realtime constraints? In the next paragraphs we sketch some scenarios and describe first solutions for arising problems.

- Running fast processors with high power demands in portable devices is impossible today, because the components cannot be supplied with enough energy and cannot be kept cold for a reasonable period of time. Some of today's processor architectures offer the feature of reducable clock speed to save power [INTEL96B]. Reducing the clock speed causes a linear reduction in energy consumption, but a similar reduction in performance. So the measure of the energy performance defined in millions of instructions per joule (MIPJ) is unchanged. However a reduced clock speed creates the opportunity for quadratic energy savings as the energy is proportional to the square of the voltage. The voltage level within a range of 1.4 Volts to 2.8 Volts again can be reduced with the clock rate. [WWDS95]. First processors are rated for multiple voltage values and clock rates. E.g., the Digital/ARM SA110 processor [DEC97] is rated for two different power/clock modes:

Clock Frequency	Voltage	Power	Performance
160 MHz	1.65 V	0.45 W	185 MIPS
215 MHz	2.0 V	0.9 W	245 MIPS

Tab. 4.1: Strong ARM SA 110 power/clock modes

Note that the values of clock rate, voltage and power satisfy the equation 4.1.

$$clock \cdot voltage^2 \propto power \tag{4.1}$$

$$\frac{160Mhz \cdot 1.65V^2}{215Mhz \cdot 2.0V^2} = \frac{0.45W}{0.90W} \tag{4.2}$$

If the manufacturers tested and rated their chips across a smooth range of voltage, the real-time operating system could reduce the voltage and slow down the CPU if the next deadline was far away. If an interrupt occurs, the CPU is powered up and clocked at high frequency to achieve a low interrupt latency. This offers high performance during bursts and reduces the power dissipation of the system which make passive cooling and battery power savings possible.

Having the possibility of integrating several million gates on a chip, we must make the design decision between a complex processing unit and multiple simple processing units on the same chip. From the point of power consumption, the single chip multiprocessor [HNO97] is quite attractive because some of the processors can be powered down in periods of low load to save energy. It is easier and more efficient to power up a processing unit than to preempt a running thread if a request has to be processed. Furthermore single chip multiprocessors help to reduce the number of context switches causing malicious side effects (see subsection 4.1.1 *Caches*) and simplify the power-management.

Power conscious real-time scheduling has more freedom in making decisions. Beside the question when something has to be executed, the operating system can decide where and how fast it should be executed.

4.1.3 System interconnection

Processors, memory and I/O devices are connected by the system interconnect. This includes passive and active components making a communication between the basic blocks (processor(s), memory and I/O) feasible. The number of transactions which can take place on the system interconnect (e.g., a bus system) is limited. Especially memory access operations, cache coherency transactions and DMA transfers impose a high load. There are two possibilities of avoiding congestion:

- The requests are processed according to priorities.
  An arbiter has to decide the order of requests which are allowed to be serviced. Normally requests are scheduled in a FIFO order. In a multiprocessor system low priority threads running on some of the CPUs can saturate the interconnect by memory, cache-coherency and I/O requests, so that a high priority thread can only submit a limited number of requests. In this situation low priority threads slow down high priority threads.
  A hardware approach to solve the problem of priority inversion due to interconnection contention is an arbitration policy preferring requests from CPU- and I/O modules working for high priority tasks. In the context switch routine the priority of a thread is written to a register. The value of this register is the priority tag of each request submitted by the CPU-module. If such a request is serviced by a memory- or I/O-module the priority tag of the request is used for the reply.
  From the beginning of networking, messages of different priority had to be transmitted from source to destination. With the rising performance of network hardware, the architecture of network routers and network switches has a great similarity to the system interconnect in a computer system. Architectural support and system software for prioritized routing and congestion avoidance is available in today's high performance routers [Cis97]. Therefore system designers can adopt router technology for real-time systems to solve the same class of problems.

- The requesting unit is asked/forced to reduce the number of requests per time frame.
  The load that an active component imposes to the interconnect can be throttled by reducing the speed of execution of the component in question. E.g., a CPU module running a low priority thread can perform idle cycles or reduce its clock frequency to slow down voluntarily. A disk controller working on requests for low priority threads can slow down its work and delay the service of requests.

Arbitration according to priority tags is a passive strategy, while request reduction is an active strategy. Both strategies have in common that the clue to the fast interconnect-access of high-priority real-time tasks lies in the throttled access of low-priority tasks.

4.1.4 Main Memory Bandwidth

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 and DMA devices can interfere and slow each other down when accessing memory. Up to now, this effect called *memory preemption* is not covered by contemporary real-time operating systems which cannot guarantee a share of the memory in time.
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.
In the next section we describe a new mechanism, called *Process Cruise Control* that maintains the execution speed of soft real-time applications in a multiprocessor environment [BEL97C]. 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. 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.

4.2 Process Cruise Control

In the last two decades memory technology has advanced concerning the volume of data that 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 is limited 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. 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 [McC95] benchmark run on a SUN E3000 server in parallel (see table 4.2).

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

Tab. 4.2: Copy rate of the STREAMS-benchmark (stream_d.c)
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 applications running on other CPUs. This effect which we named *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 MB/sec (see figure 4.1).

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.

In the next subsections 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.

4.2.1 Memory-Bandwidth Reservation and Throttling Model

To manage memory-bandwidth, which is the topic of our research, 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 [MR95][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 re-

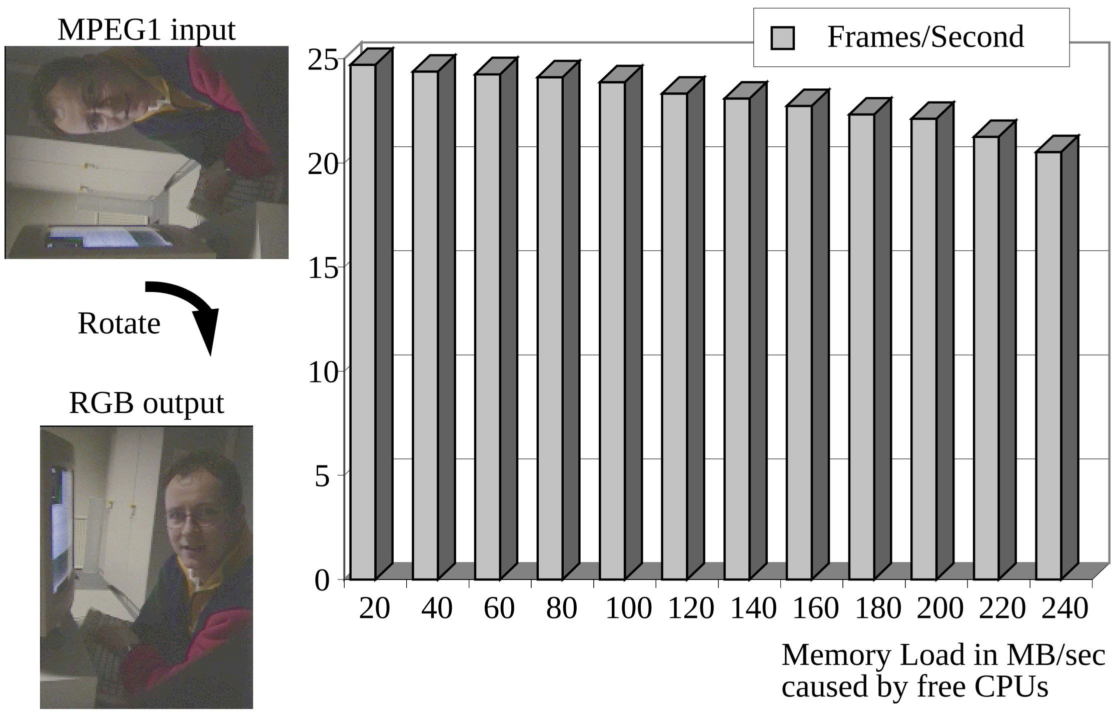


Fig. 4.1: Reduced execution speed of the video tool caused by memory preemption

sponsible 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 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 subsections, we propose concepts and implementation details to include memory bandwidth as an additional resource in a capacity reservation model.

4.2.1.1 Measurement of Memory Access

Memory access information is the total sum 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.

There are four potential regions in a shared-memory architecture suitable for the placement of memory-related event counters such that the information is usable in resource capacity reservation models:

- 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 [INTEL96A], UltraSPARC [SUN95A]) 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 [BEL97A]. Furthermore we propose architectural improvements which could overcome the limits imposed by the current hardware architecture.

4.2.1.2 Implementation of Memory Bandwidth Measurement

The SUN Enterprise X000 Server Architecture [SUN95B] is the hardware platform for our memory-bandwidth reservation model using memory access information. Each Ultra-I CPU [SUN95A] 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 4.2).

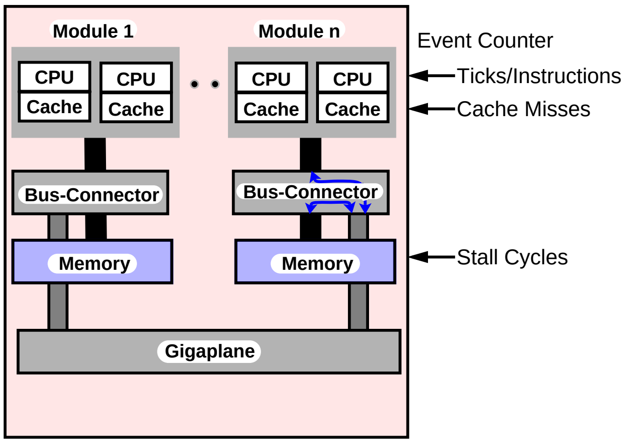


Fig. 4.2: Memory Bandwidth Measurement

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 MB/sec, 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 to 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.

4.2.1.3 Memory Access Patterns

To get an impression of the memory-access patterns of multiple classes of applications, we evaluated measurements from the following applications (see also table 4.3):

- A postscript interpreter (ghostscript) shows a high locality of reference and issues only a low rate of main memory references. The low rate of memory stall cycles indicates that ghostscript imposes only low load on the memory banks.
- A compiler (gcc) and a simple matrix multiply algorithm (mmult) imposes a medium load to the memory subsystem, yet showing a good locality of reference.
- The multithreaded numerical smoothing algorithm (active_threads) has an irregular execution order, but works on a grid of quite regular linked grid points (see figure 4.3). Because the code of all threads is shared and because of the “cache-friendly” structure of the grid, we notify an acceptable locality of reference.
- The situation changes if we look at a full adaptive numerical solver working on an unstructured adaptive grid (see figure 4.3). The locality of reference is low and the applications induces a high rate of main memory references. The high number of memory stall cycles indicates that the memory subsystem has reached its limits.
- The soft-realtime video-conferencing application relies on a continuous supply of data (1318 cache lines per millisecond). If the number of stall cycles exceeds the limit of 80 cycles per millisecond, the number of video-frames drops below the limit of 24 frame per second (see figure 4.1 on page 56).

	Clocks/Instruction	E-Cache Hit Rate (Hits/References)	Memory References/ms	Memory Stalls/ms
ghostscript	1.2	99 %	180	20
gcc	1.7	93 %	1360	74
mmult	1.4	92 %	1067	94
active_threads	2.6	90 %	1300	76
adaptive_grid	9.2	41 %	2448	360
video tool			1318	71

Tab. 4.3: Memory-Access Patterns (SUN E3000, 4 Proc., 128 MB 512 K E-Cache)

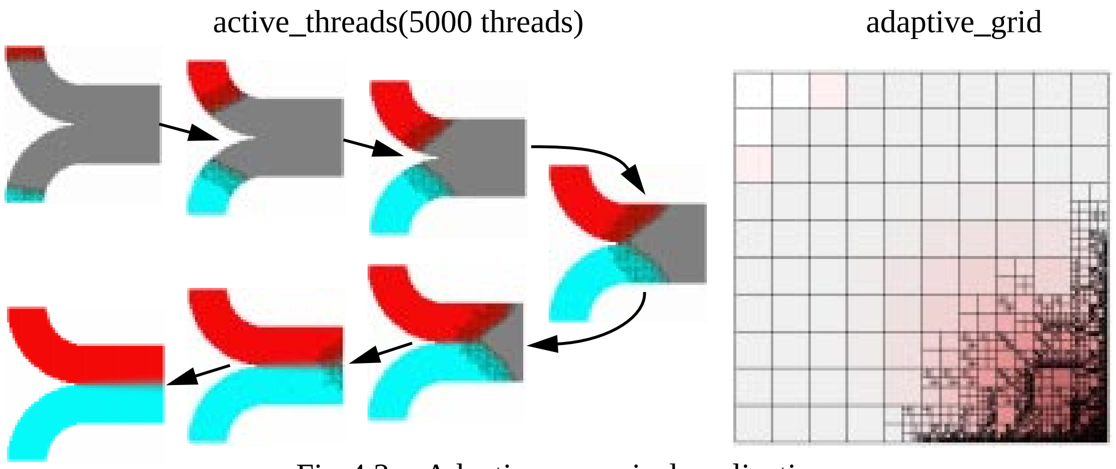


Fig. 4.3: Adaptive numerical applications

The memory access pattern of different classes of applications can vary extremely. Consequently the interference between two applications running in parallel on multiple CPUs can vary. The postscript interpreter will not influence the execution of the video application, while the adaptive_grid application will invalidate all assumptions concerning the execution speed of the real-time applications running on a dedicated CPU.

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

4.2.1.4 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 (MB/sec) - 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.

4.2.1.5 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 KB (Solaris 2.5.1 uses 8 KB page size) and the size of the external caches ranges from 256 KB to 1 MB. 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 so they do not exceed the imposed limits run the TLB-miss handler without any delay. This approach can be applied to all processor architecture that rely on a software implementation of the TLB-miss handler (e.g., all common RISC architectures).

– 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.

(3) Slowing down a process is also possible by reducing the clock frequency of the CPU. Some of today's processor architectures offer the feature of reducible clock speed to save power [INTEL96B][DEC97]. In a synchronous architecture, the CPU clock speed is a multiple of the clock speed of the system interconnect. Therefore the CPU clock speed can only be adjusted in large steps which is not sufficient for precise memory-access control. Asynchronous architectures allow any CPU clock speed but they impose high efforts for the interconnect interface.
Because all contemporary multiprocessor architectures are synchronous designs, a reduction of the clock speed is not applicable as the only mechanism for memory-control. But in conjunction with the insertion of idle cycles it would combine the advantages of memory-access control, power-management (see subsection 4.1.2), and interconnect-access control (see subsection 4.1.3).

As the Ultra-I processor architecture used for our implementation offers neither event counters generating interrupts nor any support for clock-tuning nor architectural support in the form of throttle registers, we have chosen the approach which throttles the memory-access by inserting an idle-loop into the TLB-miss handler. Because we do not have virtual memory available inside

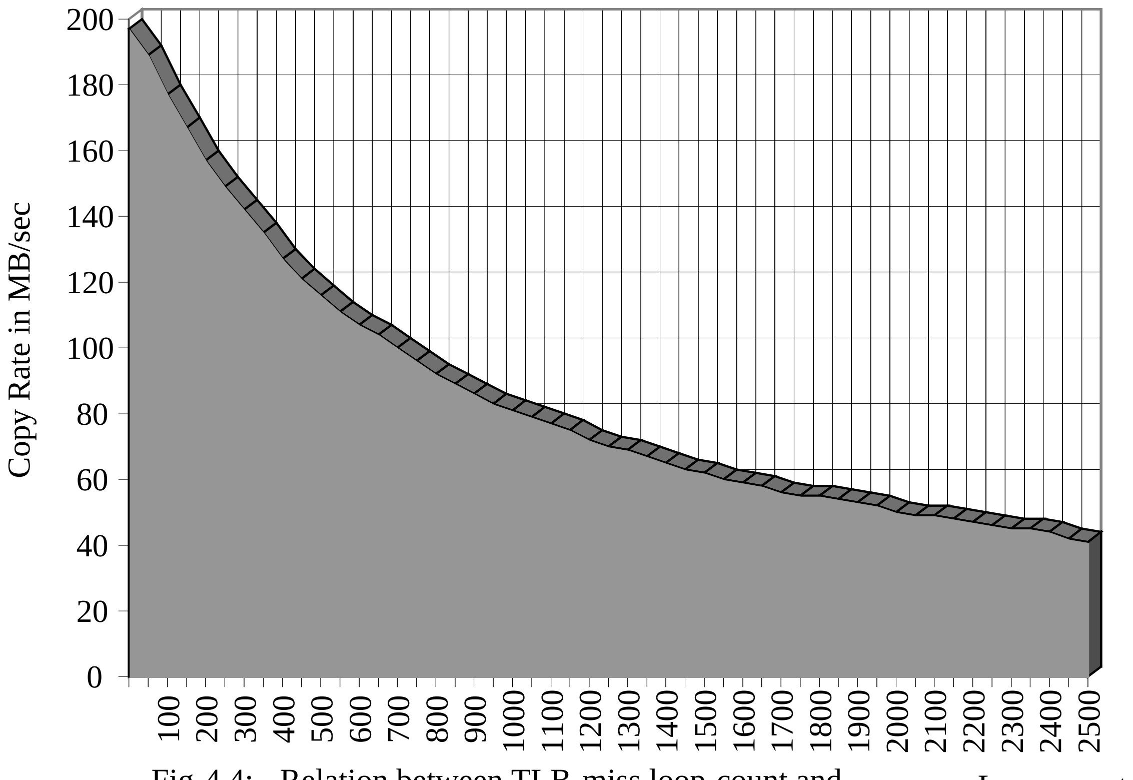


Fig. 4.4: Relation between TLB-miss loop-count and STREAMS copy-rate

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 (see figure 4.4). We can see that the memory-bandwidth a process is using can be exactly tuned by the idle-count in the TLB-miss handler.

4.2.2 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 as well as 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

4.2.2.1 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 to 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.

  $$MaxRequest = 0.9 \cdot StreamCopyRate \tag{4.3}$$

  Processes requesting the limit in memory bandwidth are very sensitive to any memory operations issued by other CPUs. Due to 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:

  $$FreeBandwidth' = (FreeBandwidth - RequBandwidth) \cdot \left(1 - \frac{RequBandwidth}{StreamCopyRate}\right) \tag{4.4}$$

  This rule is motivated by the observation that the application which wants to request some memory needs a certain fraction

$$\frac{RequBandwidth}{StreamCopyRate} \tag{4.5}$$

of the available memory cycles. Other threads are allowed to use the remaining fraction

$$1 - \frac{RequBandwidth}{StreamCopyRate} \tag{4.6}$$

of the remaining bandwidth

$$(FreeBandwidth - RequBandwidth)\,. \tag{4.7}$$

- The free bandwidth has to be greater than zero. If the free bandwidth would become negative as a result of a request, the request is denied.

$$FreeBandwidth > 0 \tag{4.8}$$

- Each thread with a guarantee is assigned its own processor

$$NumGuarantees + 1 < NumCPUs \tag{4.9}$$

$$FreeCPUs = NumCPUs - NumGurantees \tag{4.10}$$

- The free bandwidth is equally shared among the free processors

$$BandwithLimit = \frac{FreeBandwidth}{FreeCPUs} \tag{4.11}$$

Based on the measurements of the STREAMS-benchmark with 2, 3 and 4 CPUs and 2 memory

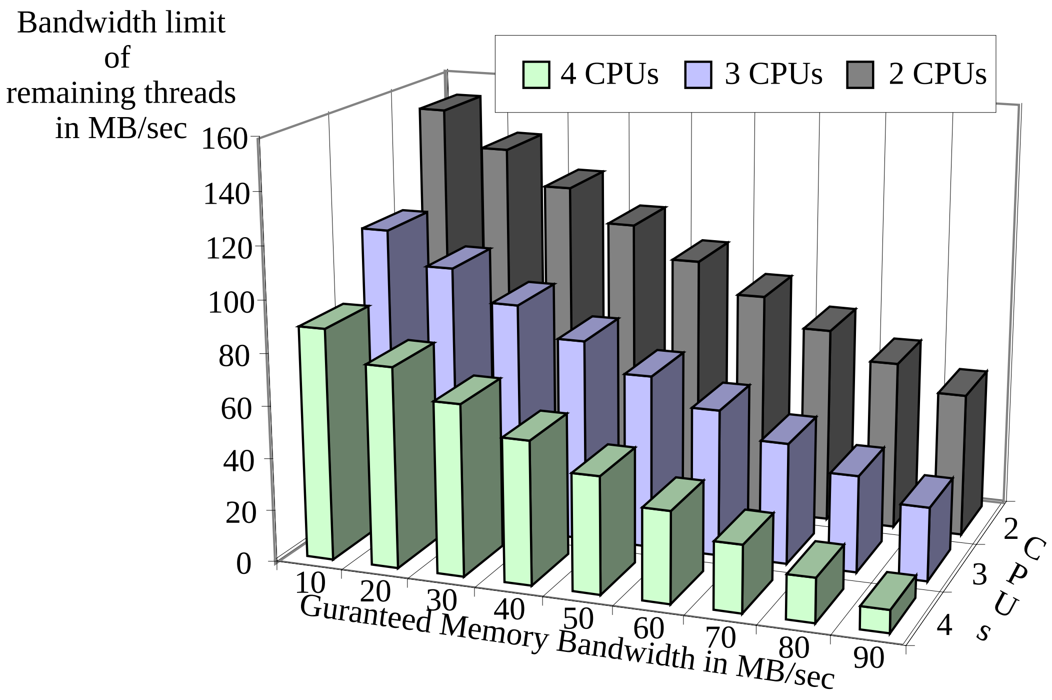


Fig. 4.5: Memory-bandwidth capacity reservation

banks (see table 4.2) 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 figure 4.5). For example, a request for 60 MB/sec limits the bandwidth of all other threads to 35 MB/sec in a machine with 4 CPUs, to 56 MB/sec when 3 CPUs are available, and to 87 MB/sec in a dual-processor configuration.

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 measurements 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. In order to smooth bursts in the memory access, we added a smoothing function in the calculation of the used bandwidth.

4.2.3 Proof of Concept

To demonstrate the benefit of process cruise control, we ran the video-image application used in section 4.2 to demonstrate the negative effect of memory preemption. As this application has a demand for 61 MB/sec, 108 MB/sec of free bandwidth can be used by the remaining 3 CPUs. This implies a bandwidth limit of 36 MB/sec for all remaining processes. Our measurement (see figure 4.6) 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 MB/sec.

4.3 Conclusions

The mutual influence of tasks in a real-time environment goes far beyond the sharing of processing units. Several resources which cannot be shared in space like memory bandwidth, interconnect bandwidth, battery power, or cooling capacity are critical for applications with predefined service rate objectives.

We have to bear in mind that we cannot conclude from the behavior of single tasks to the behavior of a system with those tasks running concurrently. The predictability of a system is low, if the execution of one task is influenced by the execution of other tasks in a time-multiplexed environment. Before we can control the interference, we have to register all events which can

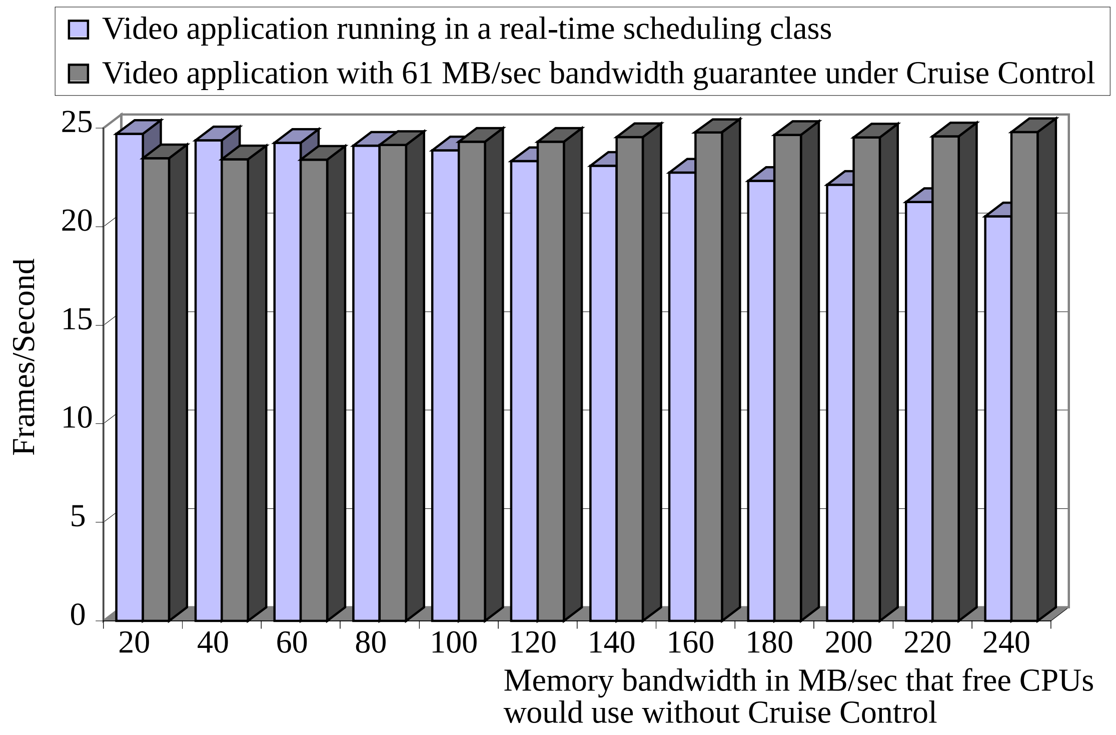


Fig. 4.6: Soft-realtime video-image processing with Process Cruise Control

cause a potential performance impact. The next step is the development of mechanisms and policies to enforce limits and to provide guarantees for critical resources.

To avoid the malicious effects of memory preemption, we have developed a complete memory-bandwidth reservation scheme. The basis of scheduling decisions is information derived from event counters in the hardware.

The prototype implementation of memory bandwidth reserves in Solaris 2.x demonstrates the feasibility of our design. We can effectively isolate real-time threads from the timing and memory-access characteristics of other threads running on different processing units. A 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.

Process Cruise Control has proved to be an effective reservation system for unmodified applications in a production environment, but it is not yet general enough to handle arbitrary interaction among applications and memory. So far, we have concentrated on reservations for memory access from CPUs. Other issues, such as the effects of DMA or cache-flushing/invalidation are not addressed in our current prototype. We expect that our design could be implemented in other operating systems with comparable effort.

5 Perspectives

Computation is more than modifying data. It comprises data modification as well as data movement. The more the gap between CPU speed and memory latency is widening, the more the focus of scheduling has to shift from the management of CPU-activity to the management of data. Data management has to concern spatial as well as temporal aspects on different levels of the memory hierarchy from caches down to the disks. In the next subsections we discuss examples of data structures, mechanisms and policies in the field of scheduling that are tailored to the special needs of advanced processor architectures. Neither claim the presented examples to be the final solution, nor are they proved by an implementation. But they should make operating system designers sensitive to the issue of memory consciousness and motivate them to revise their designs.

5.1 Memory-Conscious Data Structures

Over the past 40 years operating system structures have been established which can be found in almost all teaching books. These structures arose in a time where the speed of processing units was equivalent to the speed of memory, whereas the size of memory was very limited. As the speed of CPUs was slow, the operating system designers tried to minimize the number of instructions for operating system functions while at the same time the memory used for runtime information was minimized.
Today the situation has changed, but these old concepts are still in use and deeply buried in contemporary operating systems. In the next subsections we propose three criteria for memory-conscious data structures: *cacheability, prefetchability,* and *mapability.*

5.1.1 Cacheability

The memory layout of a data structure should facilitate the fast access of frequently referenced elements. The closer frequently referenced data are grouped in memory, the lower is the probability for conflict misses on all levels of the cache hierarchy. This design principle can be applied in the case of virtual as well as physical memory to efficiently use both, virtually indexed caches and physically indexed caches. An example is given in figure 5.1. Two records A and B of the

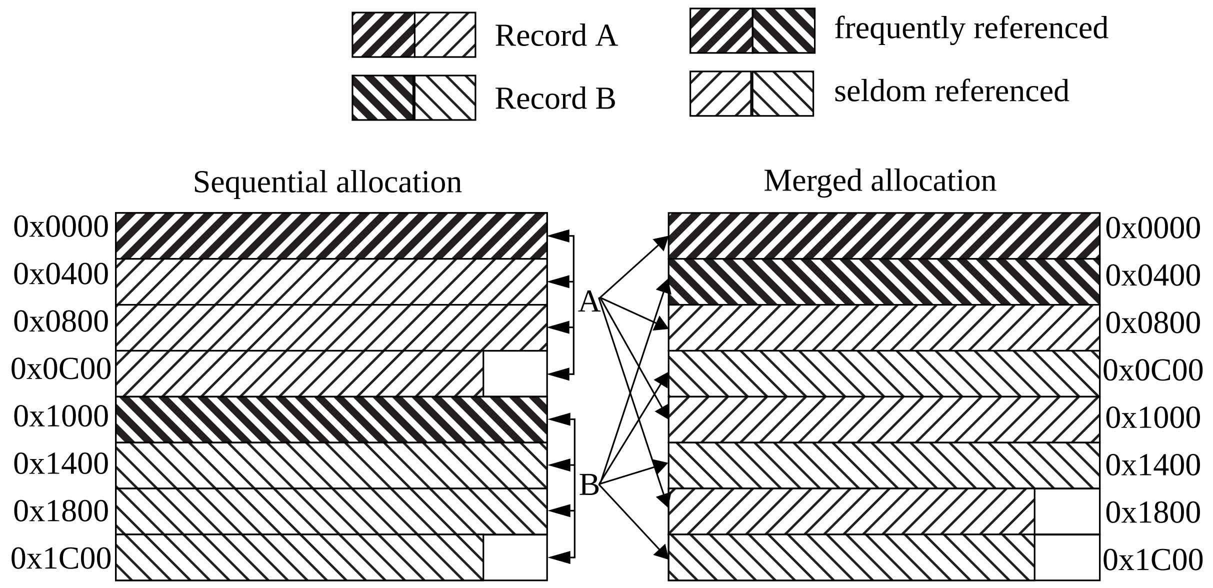


Fig. 5.1: Sequential and merged allocation of data records

same type have to be allocated in memory. The header of both records is frequently referenced. With sequential allocation both headers are mapped on the same cache lines in an architecture with direct-mapped 4KB first-level data cache. The consequence are many conflict misses. Furthermore, two TLB-entries are necessary to map the headers from virtual to physical address space if we assume pages of 4KB.
If the structures are merged so that the headers reside side by side in memory, mapping conflicts can be avoided and a single TLB-entry is sufficient because the spatial locality is improved. Reordering structures in memory is an common technique that is frequently found in the application development to better utilize caches (e.g., matrices that are referenced in the same dimension with the same indices at the same time are merged into a compound array). But a requirement of this technique is that all records of the same type reside in consecutive memory blocks, a property which is rarely found in operating systems of the past.

5.1.2 Prefetchability

Modern processor architectures offer the feature of non-faulting prefetch-operations transferring data from memory into a lockup-free cache. The goal of prefetching is to overlap execution with data movement. Prefetching operations are of high value if the processor has to browse over a large data set in search- and update functions. A prerequisite of prefetching is

- a knowledge of coming demands so that data can be fetched in time,
- a high spatial locality, and
- a TLB mapping for the data in question, because prefetch operations are non-faulting. This means that they do not throw an exception if there is no mapping.

5.1.3 Mapability

Records that are logically connected and that will be referenced in the same temporal context should be allocated in physical memory so that there are no mapping conflicts (see subsection 2.2.5.2 *Page Assignment and Page Re-Coloring*). If a good allocation is impossible due to the dynamic behavior of the operating system, only records that will be referenced within the same temporal context should reside on a page to promote page re-coloring. Furthermore, the number of allocated pages should be minimal to avoid TLB-misses.

5.1.4 Example

Because of address space limitations, data structures of today's operating systems are allocated on demand and linked by pointers. An example is the linkup of the kernel-thread structure `kthread` in the Solaris 2.x scheduling architecture (see figure 5.2). All kernel threads are linked

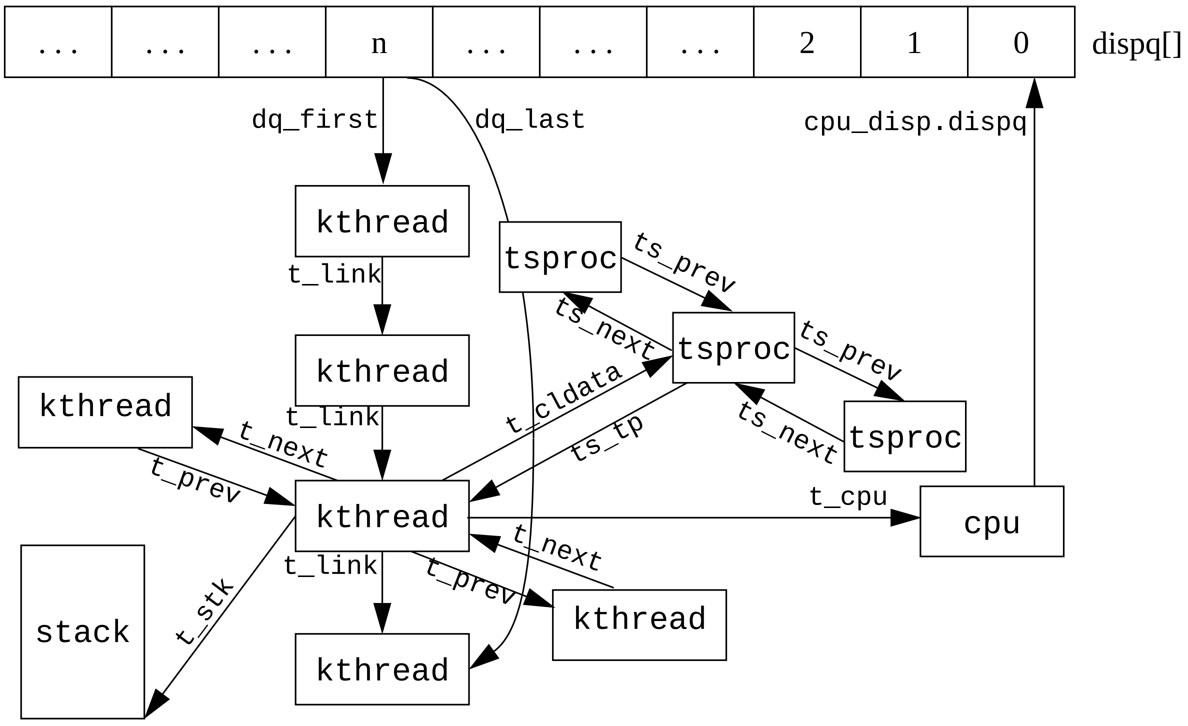


Fig. 5.2: Linkage of the `kthread` structure

by a double linked list (`t_next`, `t_prev`). Each thread belongs to a scheduling class. E.g., a thread that belongs to the time sharing scheduling class has a link (`t_cldata`) to its time-sharing class-specific thread structure (`tsproc`). All threads of this class are connected in a double linked list (`ts_next`, `ts_prev`) as well. The priority dispatch queues are established by a unidirectional linked chain (`dq_first`, `t_link`) and a pointer to the tail of the queue (`dq_last`). In a multiprocessor it is necessary to know the CPU on which the thread had last run on (`t_cpu`). Furthermore, each CPU owns an array of dispatch queues (`dispq[]`).
Each record `kthread` is allocated by a general-purpose memory allocator without any respect to the relationship to other records. The consequence of this design are many memory fragments. Memory fragmentation implies the following drawbacks:

- Lack of mapability:
  Because of the erratic memory allocation, most of the records reside on different pages and co-reside with records of a different context, although the size of the record is so small (280 bytes see table 5.1) that several records would fit into a single page. While updating all records, each record is responsible for an exception due to a TLB miss. Additionally the structure cannot benefit from page re-coloring.
- Lack of prefetchability:
  Periodically, the function `ts_update()` in Unix System V.4 has to update the scheduling parameters of all runnable threads in the time-sharing scheduling class. Despite the knowledge of the succeeding thread (`ts_next`) the record is not prefetchable because the access to the successor's data residing on another page will cause an exception.
- Lack of cacheability:
  The thread records and the corresponding class specific thread structures are distributed over the address space and therefore rise the probability for conflict misses. Despite of being referenced together entries of both records are not grouped thus they could share cache-lines.

Type of record	sizeof() in bytes
kthread	280
tsproc	32
cpu	592

Tab. 5.1: Size of kernel structures

According to the rules for memory conscious data structures we propose a scheme for the allocation and organization of the thread structures. The kernel-functions that should benefit most from the new design are:

- `swtch()+ resume()`: Election of a new thread and the switch to it
- `setfrontdq()` + `setbackdq()`: Enqueueing of a thread in the dispatch queue
- `ts_update()`: Update of scheduling parameters

We propose the following key-aspects of the new design:

(1) The kernel and the class-specific thread structure should be allocated together in a single block because they are referenced in the same temporal context. To allow cache-line alignment of both structures even for large cache-lines (128 bytes), we designate 384 bytes for the `kthread`-structure and 128 bytes for the class-specific structure (e.g., `tsproc`). Therefore we allocate 512 bytes for each thread. If this space should not be sufficient for an exotic scheduler, the scheduler could allocate additional memory and link it with the structure.

Additionally the dispatch-queue array and the cpu structure should be allocated together as well because they are referenced in every switching and queueing function (`swtch()`, `resume()`, `setfrontdq()`,`setbackdq()`).

(2) With 64-bit addressing there exists no caps for the address-space. Instead of dynamically allocated structures we recommend a static array (see figure 5.3) that has a size which can be configured according to the demands of the system. Overestimation of the demands is not a problem because only virtual memory has to be allocated at boot time. A special allocator is responsible for the proper assignment of physical pages so that a continuous virtual memory region is mapped on physical pages with continuously colored pages.

An additional improvement is the partitioning of the second level cache:
Stacks, thread-structures and the cpu/dispatcher records belong to different sets of page colors so they cannot interfere (see also subsection 4.1.1).

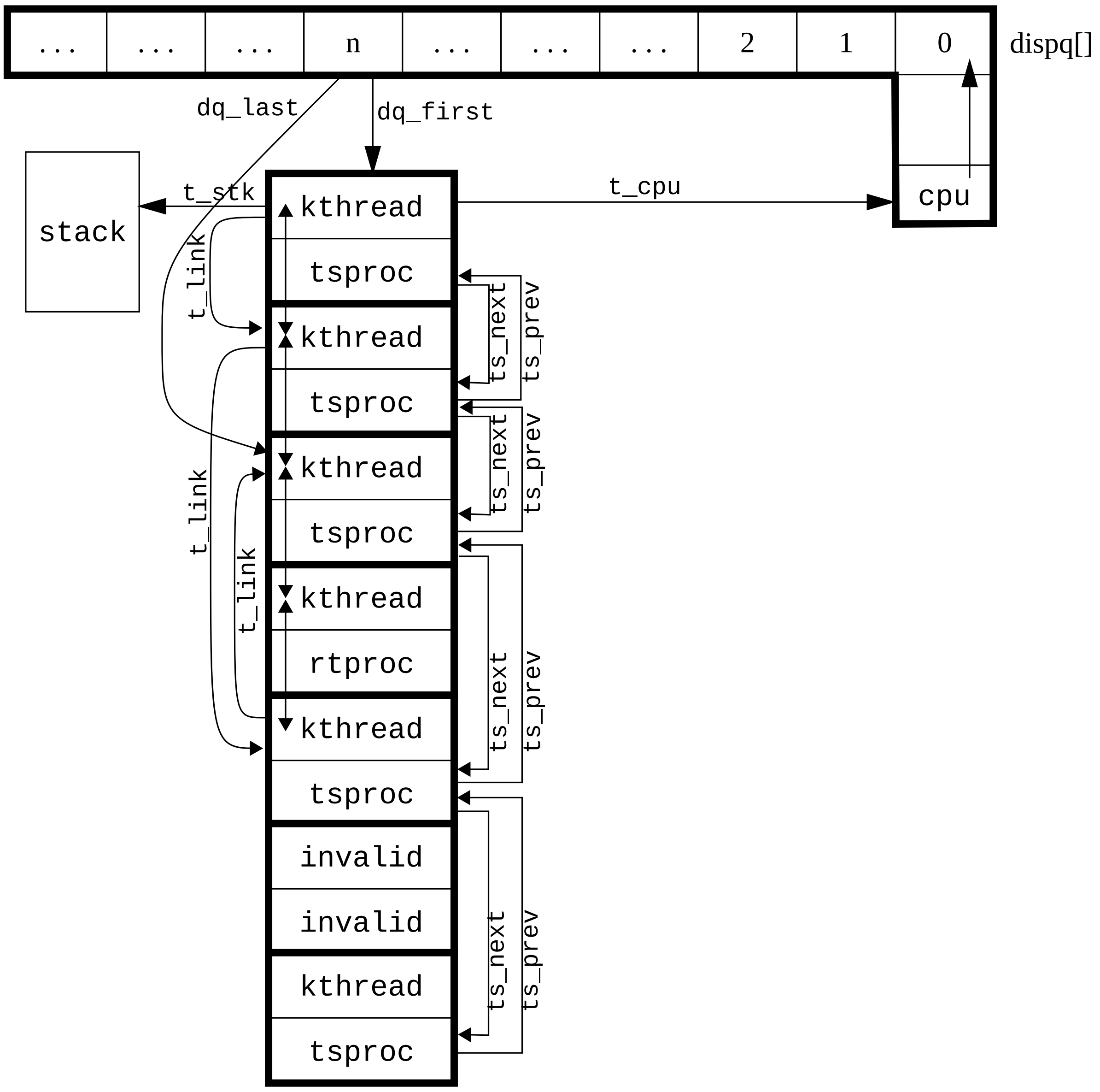


Fig. 5.3: Static array of compound thread records

(3) Instead of pointers to link up thread structures, we recommend thread identifiers which will be resolved to pointers by a hash function. Resolving identifiers offers the benefit that a structure can be moved to another address without the danger of stale references. A pointer that corresponds to an identifier may be cached, but the identifier has to match an entry of the referenced record. If the identifier does not match, it has to be resolved by the hash function. Because of the static array there is no danger of an illegal reference. However the check of the matching identifiers has to take place under the protection of a lock.

(4) Periodically the array should be rearranged so that the most frequently running threads have their records in only a few pages. With it we can avoid TLB-misses and we can prefetch the record of the thread that is designated to run on the CPU next. We even have the possibility of prefetching the top entries of a thread's stack to speed up the `resume()` function.

This example demonstrates that it is possible to design memory-conscious data structures with the property of cacheability, prefetchability, and mapability, if we need not take care for a small address space. The benefit of reduced TLB-overhead and of saved stall-cycles due to faster memory access makes up for additional operations for reference look-up, memory mapping, and record-movement.

5.2 Total Speed Control

The striving for speed is a driving force of computer science. But speed is nothing without control if we have crossing data paths causing congestion under high load.

One approach to reduce the load caused by competing components is the reduction of their speed of execution (see section 4.2 *Process Cruise Control*). However, most of the critical resources are fed by FIFO buffers to improve the throughput, to avoid synchronous data transfers, and to bridge different clock frequencies. Several throttled components can fill a buffer if they submit their requests within a small time window. If the buffer is filled, an unleashed component has no chance to bypass the other requests and will therefore not get the assigned share of the resource. To prevent filled buffers we have to throttle some components far below the value that is reasonable if we take the total capacity and the reserved capacity into consideration. E.g., in the example in subsection 4.2.2 we have to throttle down 3 CPUs to 35 MB/s if 60 MB/s are requested in a machine with more than 400 MB/s total capacity. In this example the reason for the buffering of memory requests lies in the bus system of the SUN X000 architecture supporting split transactions and several outstanding memory reads.

Another approach is to exchange the buffers by priority queues. Priorities alone are not sufficient to moderate the access to timely shared resources because priorities cannot consider the long- and middle-turn behavior of components submitting requests in bursts. But they are best suited to deal with bursts in the short term.

A combination of speed control and prioritized queueing seems to be a prospective solution to control and maintain the speed of execution without sacrificing system throughput. Now the

task of the scheduler is to determine the time, location, speed, and priority of a thread's execution using measurements from counters in all active components of the system so that a requested service can be provided. The concept of the priority queues has to be applied to all crossings on the data path: CPU–shared caches–system interconnect–memory–I/O. It also has to be applied to all crossings on the data path in the operating system, e.g., I/O buffering (buffer caches, streams buffers) and VM-subsystem.
Summing up this considerations we want clarify, that there should be no activity or data movement which is not assigned to an active entity under the control of the scheduler.

5.3 Towards Optimal Affinity Scheduling

The user-level scheduling approach presented in chapter 3 considers the locality behavior of individual threads at runtime. An additional optimization is possible if we look at the interaction of threads on the same memory regions by comparing their working set. In section 2.2.3 we used TLB information to find cooperating processes on kernel level. For fast interaction a user-level readable TLB would be necessary, as fast accessible as the processor cache and with a fine resolution in the range of 1 KB. This could be a feature of new processor generations.
The knowledge of synchronization events is an alternative way to identify cooperating threads. Cooperating threads can be collocated at the same processor, whereas a compromise between collocation and load balance has to be made. To decide an ideal point defined by the location and timing of execution, we define gravitational and repulsive forces between threads by the frequency and extent of information sharing.
The forces influence affinity in a metric space, so that threads migrate near to their (sub)optimal operation point. This affinity model resembles the computational field model for migrating objects proposed in [Tok90].
A thread *A* running on processor *X* with a great cache affinity exerts a gravitational force on thread *B* running on processor *Y* with just a small cache affinity. Thread *A* exerts the force by increasing the affinity of thread *B* to processor *X* each time *A* synchronizes with *B*. The value used for increasing the affinity depends on the size of the memory region shared by the two threads (see figure 5.4).

Likewise the approach to estimate a thread's footprint in the cache (see section 3.3.1) we use the number of cache misses occurring during the modification of the shared-memory region as the basis of the affinity adjustment.
Cooperating threads with a high compute load and rare synchronization events exert repulsive forces on each other (see figure 5.4). Threads showing this behavior can be recognized by a high number of cache misses and a long time-frame between blocking events. If these threads run on the same processor they decrease the affinity of their synchronization partner. Threads with a low affinity can be caught by an idle processor.

Threads with low affinity sharing a memory region which is large compared with the private working set exert aggregative forces on each other (see figure 5.4). Before unblocking, the affinity values of a synchronization partner will be modified in a way that the partner will run up

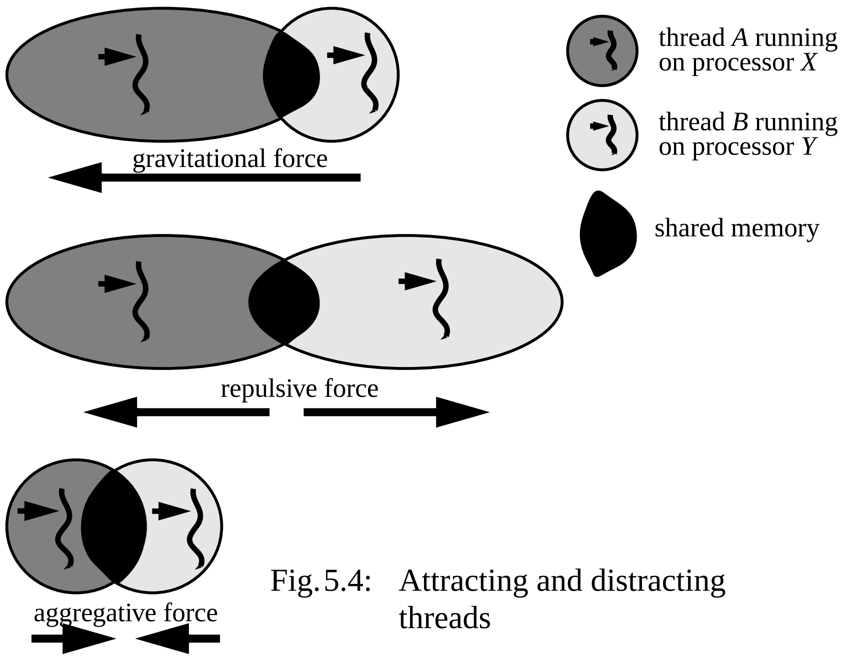

gravitational force
repulsive force
aggregative force
thread *A* running on processor *X*
thread *B* running on processor *Y*
shared memory

Fig. 5.4: Attracting and distracting threads

on the same processor as the unblocking thread. The aggregative force is influenced by the number of cache misses on shared and thread-private memory regions.
The proposed strategy has the potential to come closer to the optimum operation point of all threads concerning location and timing of execution. By introducing gravitational and aggregative forces, the number of cache misses of cooperating threads can be reduced, while compute-intensive threads with low synchronization rate exerting repulsive forces can be distributed very easily.

6 Conclusions

The nature of operating systems is changing as the focus of computing shifts from the processor to the memory. The research presented in this dissertation has concentrated on the gathering and usage of memory access patterns in the area of user-level as well as kernel-level scheduling, yet covering both time-sharing and real-time aspects.
Beside the questions when a task has to be executed and which CPU should be used, we have enlarged the freedom of scheduling to a third dimension, the speed of execution. The control over the speed of execution has proved to be an unrenounceable element to manage the access to many resources which are shared in time.

6.1 Contributions of this Research

The quality of a schedule can be improved if we know exactly about the interaction between the resources of interest. Apart from timing information and I/O related interrupts memory access information gathered in event counters and memory management units of advanced computer architectures provides a valuable source of information to improve the quality of scheduling decisions concerning efficiency and predictability.

On the coarse level of page access we have developed a new approach for evaluating the cache-related performance impact of virtual memory and in particular a dynamic scheduling policy to alleviate the cache effects directly following a context switch.
Sharing of memory pages is a common case in multi-programmed computers. The virtual memory subsystem provides the necessary information about the existence and intensity of sharing. *Follow-On Scheduling* extracts this information and schedules related threads sharing many pages so that they follow upon each other when being executed. In an environment with correct page coloring and cache-conscious system software, follow-on scheduling represents an essential step towards kernel support for the efficient use of caches.

Memory access patterns on the fine grained level of cache access can be gathered by evaluating counters of the processing unit and of the cache-controller. This research demonstrates how to use cache-miss information in different scheduling strategies to improve the cache affinity and to reduce the number of cache misses. In order to compare conventional strategies with our novel ones, we have designed and implemented the architecture of the *Erlangen Lightweight Thread Environment* (ELiTE), a user-level runtime system, offering the possibility of easily importing new affinity strategies. The conventional scheduling strategies using timing information offer the best trade-off between scheduling overhead and performance gain due to better locality in multiprocessor architectures with uniform memory access and low memory latency. One of our novel strategies using cache-miss information is based on a Markov model to estimate the cost to establish the footprint of a thread after restarting it (the reload transient). This strategy offers the best process reordering in scalable architectures with non-uniform memory access despite its algorithmic overhead. The higher the memory latency, the better the performance gain will be because of the usage of memory access patterns in novel affinity strategies making a fine-grained architecture-independent programming style possible.

The mutual influence of tasks in a real-time environment goes far beyond the sharing of processing units. Several resources which cannot be shared in space like memory bandwidth or interconnect bandwidth are critical for applications with predefined service rate objectives.
To avoid the malicious effects of memory preemption, we have developed a complete memory-bandwidth reservation scheme. The basis of scheduling decisions is information derived from memory-access counters in the hardware. A new mechanism called *Process Cruise Control* maintains the execution *speed* of soft real-time applications in a multiprocessor environment and throttles other applications with high memory demands in their speed of execution.
Process Cruise Control has proved itself as a viable approach to effectively isolate real-time threads from the timing and memory-access characteristics of other threads running on different processing units.

6.2 Future Directions

We have learned a number of lessons based on our experience with memory access patterns. The mutual influence of all components on the crossing data paths inside a multiprocessor architecture is much higher than expected at the beginning of our research efforts. We believe there is tremendous potential to be gained by exploiting information concerning the movement of data in all levels of the memory hierarchy. This thesis demonstrated the benefits to be gained in a few specific arenas. A long term research goal is to arrive at a holistic view of scheduling that integrates all flavors of scheduling to control and predict any activity in a computer system.

Bibliography

[ABD+97] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, C. Waldspurger and W. Weihl. "Continuous Profiling: Where Have All the Cylces Gone?". In *Proceedings of the Sixteenth Symposium on Operating Systems Principles,* Saint Malo, October 1997.

[ABLL92] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. "Scheduler activations: Effective kernel support for the user-level management of parallelism". *ACM Transactions on Computer Systems*, 10(1):53--79, February 1992.

[ALL89] T. Anderson, E. Lazowska, and H. Levy. "The performance implication of thread management alternatives for shared-memory multiprocessors". ACM Trans. on Computers, 38(12):1631--1644, December 1989.

[BB95] J. Barton and N. Bitar. "A scalable multi-discipline, multiple-processor scheduling framework for irix". In *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*.

[Bel94] F. Bellosa. "Implementierung adaptiver Verfahren auf komplexen Geometrien mit leichtgewichtigen Prozessen". Technical Report TR-I4-10-94, University of Erlangen-Nürnberg, IMMD IV, October 1994.

[Bel95] F.Bellosa. "Memory-Conscious Scheduling and Processor Allocation on NUMA Architectures". University Erlangen-Nürnberg, IMMD IV, TR-I4-95-06, May 1995.

[Bel96a] F. Bellosa. "Locality-information-based scheduling in shared-memory multiprocessors". In L. Rudolph D. Feitelson, editor, *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *LNCS*, pages 271--289, Honolulu, Hawaii, April 1996.

[Bel96b] F. Bellosa, M. Steckermeier.. "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, August 96.

[Bel97a] F. Bellosa. "Memory Access - The Third Dimension of Scheduling". Technical Report, Department of Computer Science IV (Operating Systems), TR-I4-97-01, January 97.

[Bel97b] F. Bellosa. "Follow-On Scheduling: Using TLB-Information to Reduce Cache Misses". In *Proceedings of the Sixteenth Symposium on Operating Systems Principles (Work in Progress Session)*, October 1997.

[Bel97c] F. Bellosa. "Process Cruise Control: Throttling memory access in a soft real-time environment". *Sixteenth Symposium on Operating Systems Principles (Poster Session)*, October 1997.

[BGN46] A. Burks and H. Goldstine and J. von Neumann. "Preliminary discussion of the logical design of an electronic computing instrument". Report to the U.S. Army Ordnance Department, 1946, also appears in “Papers of John von Neumann”, MIT Press, Los Angelos, 1987.

[Bia95] R. Bianchini. "Exploiting Bandwidth to Reduce Average Memory Access Time in Scalable Multiprocessors". Ph.D. thesis, Computer Science Department, University of Rochester, 1995.

[BLRC94] B. Bershad, D. Lee, T. Romer, and J. Chen. "Avoiding conflict misses dynamically in large direct-mapped caches". In *6th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 158--170, San Jose, CA, October 1994.

[BS97] Aaron Brown and Margo Seltzer. "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture". Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, June 1997.

[Cis97] "Cisco IOS (Internetwork Operating System) online Management Guide". Cisco Systems, 1997

[Con95] "Convex: Exemplar SPP1000/1200 Architecture". Convex Press, May 1995.

[Con97] "Convex Exemplar Profiling Manual". Convex Press, 1997.

[Che93] T. Chen. "Data Prefetching for High-Performance Processors". Ph.D. thesis, July 1993. also published as TR-93-07-01.

[CL93] M. Crovella and T. LeBlanc. "The search for lost cycles: A new approach to parallel program performance evaluation". Technical Report 479, Computer Science Department, University of Rochester, December 1993.

[DGST88] J. Driscoll; H. Gabow; R. Shrairman; R. Tarjan. "Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation". Communications of the ACM, 32:1343-1354, 1988.

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

[DEC97] "StrongARM SA-110 Power Dissipation and Thermal Characteristics". Digital Equipment Corporation, July 1997.

[Fei97] D. Feitelson. "Job Scheduling in Multiprogrammed Parallel Systems". IBM Research Report RC 87657, Second Revision, August, 1997.

[GC94] B. Goodheart and J. Cox. "The Magic Garden Explained: The Internals of UNIX System V Release 4, an Open Systems Design". Prentice-Hall, 1994.

[Gol80] H. Goldstine, "The Computer from Pascal to von Neumann". Princeton Univ. Press, 1980.

[GZH93] D. Grunwald, B. Zorn, and E. Henderson. "Improving the cache locality of memory allocation". In *SIGPLAN '93 Conference on Programming Languages Design and Implementation*, Albuquerque, NM, June 1993.

[HNO97] L. Hammond, B. Nayfeh and K. Olukotun. "A Single-Chip Multiprocessor". IEEE Computer Special Issue on "Billion-Transistor Processors", September 1997

[HMMS96] M. Horowitz, M. Martonosi, T. Mowry, and M. Smith. "Informing memory operations: Providing memory performance feedback in modern processors". In *23rd Annual International Symposium on Computer Architecture ISCA '96*, May 1996.

[HP96] J. Hennessy and D. Patterson. "Computer architecture, a quantitative approach". San Francisco, Morgan Kaufmann, 1996.

[Intel96a] "MMX Technology Developer's Guide". Intel Inc., 1996.

[Intel96b] "Advanced Power Management (APM) Inteface Specification R1.2". Intel, February 1996.

[JH97] T. Johnson and W. Hwu. "Run-time adaptive cache hierarchy management via reference analysis". In *24th Annual International Symposium on Computer Architecture ISCA '97*, June 1997.

[JM74] H. Johnson and M. Madison "Deadline Scheduling for a Real-Time Multiprocessor". In *Proceedings of European Computing Congress,* Uxbridge, 1974.

[Kep93] D. Keppel. "Tools and techniques for building fast portable threads packages". Technical Report TR UWCSE 93-05-06, University of Washington, Seattle, May 1993.

[Kes91] R. Kessler. "Analysis of Multi-Megabyte Secondary CPU Cache Memories". Ph.D. thesis, University of Wisconsin-Madison, July 1991.

[Knu73] D. Knuth. "The Art of Computer Programming, Vol. 3: Sorting and Searching". Addison-Wesley, Mass. , 1973.

[Kop95] C. Koppe. "Sleeping Threads: A Kernel Mechanism for Support of Efficient User-Level Threads". In *Proc. of International Conference of Parallel and Distributed Computing and Systems PDCS'95,* Washington D.C., October 95.

[LA93] Beng-Hong Lim and A. Agarwal. "Waiting algorithms for synchronizations in large-scale multiprocessors". *ACM Transactions on Computer Systems,* 11(1):253--297, August 1993.

[Lef90] S. Leffler. "The Design and Implementation of the 4.3BSD UNIX Operating System". Addison-Wesley, 1990.

[LHH97] J. Liedtke and H. Härtig and M. Hohmuth "OS-Controlled Cache Predictability for Real-Time Systems". In proceedings of the Third IEEE Real-time Technology and Applications Symposium (RTAS'97), Montreal, June 1997.

[LF91] W. Lynch and M. Flynn. "Paging performance with page coloring". Technical Report CSL-TR-91-492, Stanford University, October 1991.

[LM90] T. J. LeBlanc and E. P. Markatos. "Operating system support for adaptable real-time systems". *IEEE Real-Time Systems Newsletter,* May 1990.

[LMMS91] T. J. LeBlanc, E. P. Markatos, B. D. Marsh, and M. L. Scott. "First-class user-level threads". *Operating Systems Review*, 25(5):110--121, 1991.

[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.

[LSD89] I. Lehoczky and L. Sha and Y. Ding "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior". In *Proceedings of the Real-Time Systems Symposium* , pages 166--171, Santa Monica, December 1989.

[Mac94] N. Macrae "John von Neumann". Addison-Wesley, 1994.

[McC95] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". IEEE Computer Architecture newsletter, December 1995.

[Mue95] F. Mueller "Compiler Support for Software-Based Cache Partitioning". In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, La Jolla, June 1995

[MB90] J. Mogul and A. Borg. "The effect of context switches on cache performance". Technical Note WRL-TN-16, Digital Western Research Laboratory, December 1990 and In *4th Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 75--84, Santa Clara, April 1991.

[MCN+90] A. Mink, R. Carpenter, G. Nacht, and J. Roberts. "Multiprocessor performance measurement instrumentation". IEEE Computer, September 1990.

[MIPS95] "MIPS R10000 Microprocesspr User's Manual V1.1". MIPS Technologies, Inc. Mountain View, California, 1995.

[Mot94] "PowerPC 620 RISC Microprocessor, Technical Summary". Motorola Order Number MPC620/D, Motorola, 1994.

[MR95] C. Mercer and R. Rajkumar. "An interactive interface and rt-mach support for monitoring and controlling resource management". In *Proceedings of the Real-Time Technology and Applications Symposium,* May 1995.

[MST93] C. W. Mercer, S. Savage, and H. Tokuda. "Processor capacity reserves: An abstraction for managing processor usage". In *Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, October 1993.

[MST94] C. W. Mercer, S. Savage, and H. Tokuda. "Processor capacity reserves for multimedia operating systems". In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[RCLB95] T. Romer, J. Chen, D. Lee, and B. Bershad. "Dynamic page mapping policies for cache conflict resolution on standard hardware". In *Operating System Design and Implementation OSDI'94*, pages 255--266, 1995.

[Red95] U. Reder. "Implementierung eines effizienten Prozeßumschalters auf Benutzerebene". University of Erlangen-Nürnberg, IMMD IV, Studienarbeit, February 1995.

[Rue94] Ulrich Rüde. "On the multilevel adaptive iterative method". *SIAM Journal on Scientific and Statistical Computing*, Vol. 15, 1994.

[SAW95] I. Stoica and H. Abdel Wahab. "A new approach to implement proportional share resource allocation". Technical Report Technical Report 95-05, Department of Computer Science, Old Dominion University, April 1995.

[SFG+91] K. Schwan, H. Forbes, A. Gheith, B. Mukherjee, and Y. Samiotakis. "A Cthread library for multiprocessors". Technical report, College of Computing, Georgia Institute of Technology, January 1991.

[SG94] A. Silberschatz and P. Galvin. "Operating System Concepts". Forth Edition, Addison-Wesley, Reading, 1994.

[SGH97] V. Santhanam, E. Gornish, and W. Hsu. "Data prefetching in the hp pa-8000". In *24th Annual International Symposium on Computer Architecture ISCA '97*, June 1997.

[SGI96] "Technical Overview of the Origin Family". Silicon Graphics, Mountain View, 1996.

[SL93] M. Squillante and E. Lazowska. "Using processor-cache affinity information in shared-memory multiprocessor scheduling". *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131--143, 1993.

[SPN96] A. Saulsbury, F. Pong and A. Nowatzky. "Missing the Memory Wall: The Case for Processor/Memory Integration". In *23rd International Symposium on Computer Architecture ISCA'96*, June 1996.

[Ste95] M. Steckermeier. "Using Locality Information in User-Level Scheduling". University Erlangen-Nürnberg, IMMD IV, TR-I4-95-14, December 1995.

[SUN95a] "UltraSPARC Programmer Reference Manual". Revision 1.0, SUN Microsystems Inc., 1995.

[SUN95b] "Ultra Enterprise X000 Server Family: Architecture and Implementation". Technical White Paper, Sun Microsystems Inc., 1995.

[SW95] P. Sobalvarro and W. Weihl. "Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors". In *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*.

[Tok90] M. Tokoro. "Computational Field Model: Toward a New Computation Model/ Methodology for Open Distributed Environment". Sony Computer Science Laboratory Inc., SCSL-TR-90-006, June 1990.

[TS87] D. Thiebaut and H. Stone. "Footprints in the cache". *ACM Transactions on Computer Systems*, 5(4):305--329, November 1987.

[TTG95] J. Torellas, A. Tucker, and A. Gupta. "Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors". *Journal of Parallel and Distributed Computing*, 2(24):139--151, February 1995.

[Tuc93] A. Tucker. "Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors". Ph.D. thesis, Department of Computer Science, Stanford University, December 1993. also published as CS-TN-94-601 by Stanford University, Department of Computer Science.

[WW95] C. Waldspurger and W. Weihl. "Lottery scheduling: Flexible proportional-share resource management". In *Operating System Design and Implementation OSDI'94*, 1995.

[WWDS95] M. Weiser, B. Welch, A. Demers, and S. Shenker. "Scheduling for reduced cpu energy". In *Operating System Design and Implementation OSDI'94*, 1995.

[WMc94] W. Wulf and S. McKee. "Hitting the Memory Wall: Implications of the Obvious”". Computer Architecture News, 23(1):20-24, March 1995.

[ZLT+96] M. Zagha, B. Larson, S. Turner and M. Itzkowitz. "Performance Analysis Using the MIPS R10000 Performance Conters". In *Proceedings of Supercomputing ‘96*, November, 1996.

Ablaufsteuerung unter Berücksichtigung der drei Freiheitsgrade Zeit, Ort und Geschwindigkeit

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Frank Bellosa

Erlangen - 1998

Als Dissertation genehmigt von
der Technischen Fakultät der
Friedrich-Alexander Universität Erlangen-Nürnberg

Tag der Einreichung:	22. Juni 1998
Tag der Promotion:	27. November 1998
Dekan:	Prof. Dr. G. Herold
Berichterstatter:	Prof. Dr. F. Hofmann
	Prof. Dr. M. Dal Cin

Inhaltsverzeichnis

Verzeichnis der Abbildungen . . . . *iii*
Verzeichnis der Tabellen . . . . *v*

1. Einleitung 1

1.1 Ist die Ablaufsteuerung eine autonome Einheit? *1*
1.1.1 Prozessor Caches *2*
1.1.2 Hauptspeicher *2*
1.1.3 Energieversorgung *3*
1.1.4 Ein-/Ausgabe *4*

1.2 Ablaufsteuerung mit drei Freiheitsgraden *4*

1.3 Quellen von Speicherzugriffsinformation *6*

2. Prozessverwaltung auf Betriebssystemebene 9

2.1 Entscheidungskriterien in der klassischen Prozessverwaltung *9*

2.2 Follow-On Ablaufsteuerung *10*
2.2.1 Der Einfluß der Ablaufsteuerung auf die Caches *10*
2.2.2 Gemeinsam genutzte Seiten und Wiederverwendung *11*
2.2.3 Durch TLB-Zugriffsfehler gesteuertes Einordnen *13*
2.2.4 Messungen *15*
2.2.5 Voraussetzungen für die Follow-On Ablaufsteuerung *17*

2.3 Zusammenfassung *21*

3. Prozessverwaltung auf Benutzerebene 23

3.1 Anforderungen des Hochleistungsrechnens an die Ablaufsteuerung *23*

3.2 Erlangen Lightweight Thread Environment (ELiTE) *25*
3.2.1 Architektur des Parallelrechners CONVEX SPP *26*
3.2.2 Architektur der Ablaufsteuerung in ELiTE *27*
3.2.3 Details der Implementation *30*

3.3 Ablaufsteuerung nach Affinitätsmaßen *33*
3.3.1 Strategien der Ablaufsteuerung *34*
3.3.2 Beurteilung der Messungen *40*

3.4 Interaktion zwischen der Benutzer- und der Betriebsystemebene *45*

3.5 Zusammenfassung *46*

4. Ablaufsteuerung bei schwachen Echtzeitanforderungen 47

4.1 Vernachlässigte Ressourcen . . . *47*

4.1.1 Caches ... *49*
4.1.2 Energie ... *51*
4.1.3 Verbindungsnetzwerk ... *53*
4.1.4 Bandbreite des Hauptspeichers ... *53*

4.2 Regelung der Ablaufgeschwindigkeit (Process Cruise Control) . . . *54*

4.2.1 Reservierung und Drosselung der Hauptspeicherbandbreite ... *55*
4.2.2 Reservierungsmodell und Drosselungsstrategie ... *64*
4.2.3 Tragfähigkeit des Ansatzes ... *66*

4.3 Zusammenfassung . . . *66*

5. Ausblick 69

5.1 Anpassung der Datenstrukturen an die Speicherarchitektur . . . *69*

5.1.1 Eignung zur Cache-Einlagerung ... *69*
5.1.2 Eignung zur vorausschauenden Einlagerung ... *70*
5.1.3 Eignung zur Minimierung der Arbeitsmenge ... *71*
5.1.4 Beispiel ... *71*

5.2 Umfassende Geschwindigkeitskontrolle . . . *74*

5.3 Auf dem Weg zu einer perfekten Ablaufsteuerung nach Affinitätsmaßen . . . *75*

6. Abschließende Betrachtungen 77

6.1 Beitrag zur Wissenschaft . . . *77*

6.2 Zukünftige Forschungsrichtungen . . . *78*

Literatur . . . *79*

K. Kurzfassung 89

K.1 Einleitung . . . *89*

K.2 Zusammenfassung . . . *89*

Kurzfassung

K.1 Einleitung

Die Sichtweise bei der Entwicklung von Betriebssystemen ändert sich in dem Maß, wie sich in Rechensystemen die Leistungsengpässe von der Prozessorarithmetik hin zum Datentransfer mit dem Speicher verlagern.
Die Ablaufsteuerung ist das Herz der Ressourcenverwaltung. Sie steuert die Aktivität der Prozessoren und aktualisiert den Abarbeitungsplan. In der Vergangenheit besaß die Ablaufsteuerung zwei Freiheitsgrade. Auf Monoprozessoren konnte sie den Zeitpunkt entscheiden, zu dem ein Aktivitätsträger vom Prozessor abgearbeitet wurde. Bei Multiprozessorrechnern kam noch ein weiterer Freiheitsgrad hinzu: Nicht nur wann ein Aktivitätsträger ablaufen durfte wurde bestimmt, sondern auch, auf welchem Prozessor dies geschehen sollte.

Die Forschungsarbeiten, die in dieser Dissertation vorgestellt werden, konzentrieren sich auf die Erfassung und Auswertung von Speicherzugriffsmustern im Bereich der Ablaufsteuerung auf Benutzer- und Kernebene. Dabei wird sowohl der Bereich der Ablaufsteuerung unter Echtzeitanforderungen als auch das Gebiet des Mehrprogramm-Betriebs im Zeitscheibenverfahren abgedeckt.
Neben der Frage, wann ein Auftrag ausgeführt werden soll und auf welchem Prozessor dies stattfinden soll, erweitern wir die üblicherweise betrachteten Freiheitsgrade der Ablaufsteuerung um eine dritte Dimension: die *Geschwindigkeit der Ausführung*. Die Geschwindigkeitssteuerung der Ausführung hat sich als unumgänglicher Mechanismus herausgestellt, um den Zugriff auf Ressourcen zu regeln, die im zeitlichen Multiplex betrieben werden.

K.2 Zusammenfassung

Die Qualität eines Ablaufplans für Prozesse kann verbessert werden, wenn die Wechselwirkungen von allen entscheidenden Ressourcen bekannt sind. Neben genauer Zeitinformation und dem Wissen über Unterbrechungen, die von der Ein-/Ausgabe herrühren, kann Information über den Speicherzugriff eine wertvolle Informationsquelle darstellen, um die Qualität der Ablaufpläne hinsichtlich Effizienz und Vorhersagbarkeit zu verbessern. Das Wissen über den Speicherzugriff kann in Ereigniszählern und der Speicherverwaltungshardware gesammelt werden.

Auf der Ebene des Seitenzugriffs wurde ein neuartiger Ansatz entwickelt, um zu untersuchen, welche Einflüsse das Zusammenspiel von Cache-Effekten und virtueller Speicherverwaltung auf die nutzbare Rechenleistung hat. Insbesondere wurde eine dynamische Strategie zur Ablaufsteuerung vorgestellt, welche diejenigen Effekte abmildert, die direkt nach einem Kontextwechsel zu beobachten sind.
Die gemeinsame Nutzung von Speicherseiten ist ein häufig anzutreffender Fall in einem Rechner mit Mehrprogrammbetrieb. Die virtuelle Speicherverwaltung bietet die notwendigen Informationen an, um das Vorhandensein und die Intensität der gemeinsamen Nutzung beurteilen zu können. Die *Follow-On* Ablaufsteuerung sammelt diese Information und plant den Ablauf von logisch ‘verwandten’ Aktivitätsträgern, die viele Speicherseiten gemeinsam nutzen, so daß sie im Ablauf aufeinanderfolgen. In einer Umgebung mit korrekt eingefärbten Seiten and speicherbewußter Systemsoftware stellt die *Follow-On* Ablaufsteuerung einen wesentlichen Schritt hin zu einer durch den Betriebssystemkern unterstützten effizienten Nutzung von Cache-Speichern dar.

Speicherzugriffsmuster auf der feinkörnigen Ebene der Caches können gesammelt werden, indem die Ereigniszähler der Cache-Steuerung ausgewertet werden. Diese Arbeit zeigt auf, wie die Information über Cache-Zugriffsfehler in verschiedenen Abarbeitungsstrategien genutzt werden kann, um die Affinität zum Cache zu verbessern und die Anzahl der Cache-Zugriffsfehler zu verringern. Um die etablierten Strategien mit den neuartigen zu vergleichen, wurde die Architektur des ***E****rlangen* ***Li****ghtweight* ***T****hread* ***E****nvironment* (ELiTE) entwickelt und implementiert. ELiTE ist eine Laufzeitsystem auf Benutzerebene, das eine einfache Integration neuer Affinitätsstrategien ermöglicht. Etablierte Strategien, die Zeitinformation benutzen, bieten einen guten Kompromiß zwischen Aufwand und Nutzen auf Multiprozessorarchitekturen mit uniformem Speicherzugriff. Eine der neuartigen vorgestellten Strategien nutzt Information über Cache-Zugriffsfehler in einer Markov-Analyse, um den aktiven Arbeitsbereich (*footprint*) zu bestimmen, der im Cache eingelagert ist, sowie den Aufwand, um diesen Arbeitsbereich nach einer Verdrängung wiederherzustellen (*reload transient*). Diese Strategie bietet trotz ihres algorithmischen Aufwands die beste Ablaufsteuerung in Architekturen mit nicht-uniformem Speicherzugriff. Je höher die Speicherlatenz, um so besser ist der Leistungsgewinn, der aus Information über die Speicherzugriffsmuster gezogen werden kann. Dies macht einen feinkörning-parallelen und architektur-unabhängigen Programmierstil erst möglich.

Der gegenseitige Einfluß von Aktivitätsträgern in einer Echtzeitumgebung geht weit über die gemeinsame Nutzung der Recheneinheiten hinaus. Mehrere Ressourcen, die nicht räumlich unterteilt werden können, wie z.B. die Hauptspeicher- oder Netzwerkbandbreite, sind für Anwendungen kritisch, die eine bestimmte Dienstgüte erbringen sollen.
Um die unangenehmen Effekte der Bandbreitenverdrängung (*memory preemption*) zu vermeiden, wurde eine Reservierungsstrategie für die Hauptspeicherbandbreite entwickelt. Basis dieser Ablaufsteuerung ist Information aus Ereigniszählern der Hardware zur Speicheransteuerung. Ein neuer Mechanismus, der *Process Cruise Control* genannt wird, hält die *Ausführungsgeschwindigkeit* von Anwendungen mit schwachen Echtzeitanforderungen aufrecht und

drosselt Anwendungen mit hohen Speicheranforderungen und niedriger Ausführungspriorität. *Process Cruise Control* hat sich als gangbarer Ansatz herausgestellt, um die Ausführung von Anwendungen mit schwachen Echtzeitanforderungen klar von anderen parallel ablaufenden Anwendungen abzukoppeln, gleichgültig, welches Zeit- und Speicherzugriffsverhalten diese aufzeigen.

Wir haben aus unseren Erfahrungen bei der Beschäftigung mit Speicherzugriffsmustern eine Menge gelernt. Der gegenseitige Einfluß aller Komponenten auf den sich kreuzenden Datenpfade innerhalb eines Multiprozessors ist wesentlich intensiver, als zu Beginn der Betrachtungen angenommen wurde. Wir konnten aufzeigen, welches gewaltige Potential in der Ausnutzung von Lokalitätsinformation auf allen Ebenen der Speicherhierarchie liegt. Diese Arbeit konnte den Leistungsgewinn an einigen ausgewählten Beispielen aufzeigen.

Ein langfristiges Forschungsziel ist es, zu einer ganzheitlichen Sicht der Ablaufsteuerung zu gelangen, die das Verhalten von Aktivitätsträgern in allen Teilen eines Rechners vorhersagen und gezielt steuern kann.

Kurzfassung

Zusammenfassung

Lebenslauf

Persönliche Daten	Name:	Frank Bellosa
	Adresse:	Martensstraße 1 D-91058 Erlangen
	E-Mail:	bellosa@informatik.uni-erlangen.de
	URL:	http://www4.informatik.uni-erlangen.de/~bellosa
	Geburtsdatum:	15. August 1967
	Geburtsort:	Coburg
	Familienstand:	Ledig
Ausbildung	11/98	Abschluß des Promotionsverfahrens Thema: *Three Dimensions of Scheduling*
	2/93 - 4/98	Forschung im Rahmen des SFB Multi-prozessoren und des ELiTE Projekts
	11/87 - 1/93	Studium der Informatik an der Universität Erlangen Schwerpunktfach: Rechnerarchitektur Nebenfach: Elektrotechnik Diplomarbeit: *Architekturunabhängige Programmierung strömungsmechanischer Probleme mit funktionalen Sprachen*
	10/86 - 12/87	Grundwehrdienst als Stabsdienstsoldat
	6/86	Abitur, Gymnasium Casimirianum, Coburg
Berufliche Erfahrung	5/98 - 1/99	Mitarbeiter der Basisentwicklung im Unterneh-mensbereich Öffentliche Netze der Siemens AG
	2/93 - 4/98	Wissenschaftlicher Mitarbeiter am Institut für Informatik der Universität Erlangen (Betriebssysteme)
	2/93 - 6/96	Mitarbeiter der Systemgruppe am Rechen-zentrum der Universität Erlangen
	1990 - 1992	Studentische Hilfskraft am Institut für Informatik der Universität Erlangen (Rechnerarchitektur)
	1988 - 1990	Studentische Hilfskraft am Lehrstuhl für Technische Elektronik der Universität Erlangen