Instrumentation and Measurement of Multithreaded Applications

Alexander Voß

Januar 1997 DA-25-95

Instrumentation and Measurement of Multithreaded Applications

Studienarbeit im Fach Informatik

vorgelegt von

Alexander Voß

geb. am 13. August 1972 in Bergisch Gladbach

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	***Prof. Dr. Fridolin Hofmann*** ***Dipl. Inf. Frank Bellosa***
Beginn der Arbeit:	3. April 1996
Abgabe der Arbeit:	03. Januar 1997

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 28. Dezember 1996 ______________________________

Zusammenfassung

Ziel dieser Studienarbeit ist es, eine Programmierumgebung zu schaffen, in der Programme mit mehreren Aktivitätsträgern entwickelt und vermessen werden können. Die Prozesse der Programmentwicklung und Instrumentierung sollen dabei möglichst unabhängig von Zielsystem ablaufen, dessen spezifische Eigenschaften trotzdem genutzt werden können. Wichtige Fragestellungen, die es im Rahmen dieser Arbeit zu beantworten gilt, sind:

- Wie können die gewünschten Analysedaten ohne wesentliche Beeinflussung des Programmablaufs gewonnen und gespeichert werden?
- Welche Modifikationen müssen evtl. am Laufzeitsystem vorgenommen werden, um Ereigniszähler bei der Umschaltung zwischen verschiedenen Aktivitätsträgern zu sichern?
- Wie können diese Daten graphisch sinnvoll aufbereitet werden, um dem Programmierer Auskunft über das Laufzeitverhalten seines Programmes zu geben?
- Wie kann die Instrumentierung abstrahiert werden, damit eine einfache Portierbarkeit des Systems gegeben ist?

Es ist geplant, die Leistungsfähigkeit des entwickelten Systems an einem Beispiel aufzuzeigen.

Contents

Friedrich-Alexander-Universität Erlangen-Nürnberg

Abstract

Instrumentation and Measurement of Multithreaded Applications

by Alexander Voß

How to write an abstract

- readers of an abstract want to decide weather to read the full thesis
- present the outcomes of your work
- an abstract is a small article in itself and thus stands separate from the thesis
- keep it simple
- do not make claim that are not justified in the paper
- claim some new result unless you write a survey
- avoid "'In this paper"' or similar introductions

Keywords Multithreading, Measurement, Instrumentation

Chapter 1

Introduction

Problem Statement

Computer systems continue to become more powerful, but as they do so the demands always grow beyond the reach of any existing system. The performance increase that can be achieved by reducing cycle times is limited because of the finite speed at which information can propagate and because of energy dissipation problems. The only other way to increase the performance of a system is to exploit the parallelism inherent in the application.

Parallelism on the level of (micro-)instructions has been used for a long time – most of the technology is in use since the advent of supercomputing. Today, even the cheapest microprocessors have multiple independent execution units and implement pipelining. Optimising compilers that help to exploit this fine-grained parallelism are widely available. As technology matures, its limits become visible. The parallelism that can be achieved at this level is very limited. Scalar arithmetic instructions for example cannot make efficient use of pipelines deeper than fourteen stages [HP96, page 210].

The advent of cheap single-chip microprocessors has made it feasible to build multicomputer and multiprocessor systems that exploit parallelism at a much higher level. Applications are split into multiple threads of control that cooperate in order to achieve a common goal.

Development of applications that contain multiple threads of control is still poorly understood and resembles an art rather than an engineering discipline. Only few tools are available that aid in managing the complexity of concurrent computation. Most of these only support one step in the development cycle, e.g. debugging.

Another problem is that computer systems become more complex as the number of processors and levels in the memory-hierarchy increase. Some of this complexity leaks through even to the userlevel. Todays machines no longer provide the programmer with a consistent single system image.

Goals of this Thesis

The primary goal of this project is to provide tools that allow the programmer to write multithreaded applications in an architecture-independent way and evaluate their performance in different environments, i.e. different hardware, operating systems, and runtime systems.

Performance measurement shall provide an insight into the dynamics of the program under development as well as the underlying system. As portability is a primary goal, no additional hardware may be used to record the data. It follows that a software solution has to be found that meets the following criteria:

- Influence of the measurement part of the system on the execution of the application program must be minimised. All blocking operations must be done by separate kernel-threads.

- The system must allow the user to specify the set of events that are to be recorded. Producing too much data slows down the process and hides the important facts.

- Instrumentation and Visualisation should use the identifiers given in the application's source code. The programmer must be able to see the relation between code and program dynamics.

more

Outline of the Thesis

Performance of multithreaded applications heavily depends on the properties of the underlying system. Hardware architecture, operating system design, and design of the runtime system interact with each other in intricate ways. To understand the reasons for an application's behaviour, one has to look at all levels of the system.

Thus the first chapters discuss some selected topics from the fields of computer architecture, operating system design, runtime support, as well as performance evaluation that constitute the research domain for this thesis. This material is meant to show possible performance problems and programming errors as well as ways to improve the software.

Beginning with chapter 5, the programming environment Socrates is presented. Its functionality is described and some insight is given into its implementation. A simple example illustrates the process of developing and evaluating multithreaded code. Conclusions are drawn in chapter 6. Appendices inform about topics such as other measurement tools or the software and hardware that has been used.

Chapter 2

Multiprocessor Architecture

This chapter presents some aspects of computer architecture that are closely related with multiprocessor performance. The goal is to show possible performance problems and programming errors. *For further information on the architecture of multiprocessor systems see [HB84, HP96, Kai96].*

2.1 Tightly vs. Loosely Coupled Systems

Multiprocessors are classified as being tightly vs. loosely coupled, depending on whether they share a global physical memory system or not. In tightly coupled systems, processors have direct access to all memory locations and thus memory can be used for communication and synchronisation whereas in loosely coupled systems message passing has to be used. Tightly coupled architectures can be further subdivided as follows:

2.1.1 Uniform Memory Access (UMA)

In systems with uniform memory access, each processor has direct access to a single shared memory. All memory locations are equidistant (in terms of access times) to each processor. Most UMA-systems incorporate caching to eliminate memory contention but this mechanism is hidden from applications.

2.1.2 Cache Only Memory Access (COMA)

Systems with cache only memory access do not have a physically shared memory but caches only. These caches constitute the machines memory and, together, form a single address space. Access times vary depending on whether the requested memory location is in the local cache or in a remote one. Application software may be ignorant of the systems architecture as the machine behaves very much like a UMA-machine with caching.

2.1.3 Nonuniform Memory Access (NUMA)

Systems with nonuniform memory access have a distributed shared physical memory. Each partition of it is directly attached to a node but can be accessed by processors in other nodes via the interconnection network. Thus memory access times differ depending on whether the requested location is local to the node or remote. This added level of complexity may be hidden from application software but doing so leads to suboptimal performance. To make best use of the hardware, the programmer must take its architecture into consideration. Caching may be used between processors and local memory as well as between nodes. Machines with coherent caching at the hardware level are called cc-NUMA.

2.2 Interconnection Networks

Interconnection networks can be built using either packet- or circuit-switching technology and can use various topologies. The more parallelism and redundancy is incorporated, the more complex and expensive the interconnection network gets. Most solutions in use today represent a reasonable compromise between performance and cost:

Buses are the least expensive interconnection networks. They are also very easy to manage but do not provide multiple connections at the same time or failure tolerance. Buses do not scale well – the only way to introduce parallelism is to replicate them. Systems with a very limited number of processors use a bus or a small set of buses. Many larger systems, however, use buses at node-level and another topology at the global level.

Rings allow more simultaneous data transfers than busses because all segments (the paths between two nodes) of the ring may be used for communication at the same time. They can also provide fault tolerance if data can be sent in both directions. On the downside, data that is sent between nodes that are not adjacent has to travel through switches of other nodes before reaching the destination, thereby being delayed. This performance problem grows with the number of nodes connected to the ring.

Cross-bar Switches provide multiple exclusive communication lines between processors and memory[1]. If memory is partitioned into n blocks, n concurrent memory accesses can be accommodated. Contention can only occur if two processors try to access the same memory module. Cross-bar switches scale well but the costs grow quadratically which makes them prohibitively expensive with larger numbers of processors.

[1]They may also be used to connect multiple processors.

Multistage switches provide a means of interconnection that scales reasonably well in both performance and cost. The complexity of the network grows logarithmically with increasing parallelism. As opposed to cross-bar switches, contention can occur at the network-level. Buffers can be introduced to address this problem.

Hypercube networks are often used in loosely coupled systems with store-and-forward communication. Like multistage switches, their complexity increases only logarithmically. They scale well but the number of processors always has to be a power of two. Contention at network-level can occur but is addressed by using store-and-forward communication.

2.3 Caching

A cache is a buffer used to hide differences in operating speed. The general scheme can be applied in a lot of situations. Introducing caches into the memory system has three advantages: the latency of accesses to main memory is hidden, bandwidth increased, and contention of interconnection structures reduced.

Processor speed continues to grow faster than main memory speed and novel multiprocessor architectures have deep memory hierarchies with latencies that can reach a couple of hundred processor cycles. These developments make caches the central factor in system performance. This section describes the key attributes of cache subsystems and their impact on performance.

For further information about caching see [HP96, Kai96, Sch94].

2.3.1 Direct Mapped vs. Associative Caches

In a cache subsystem that uses direct mapping, a data item can be stored in only one cache line. Its address is split into three parts: the first is used to address the line that the data can be found in, the second is stored in the tag ram to uniquely identify the data in the cache, and the third is the offset of the data in the line.

Associative caches use fewer bits to address the cache line than would be necessary to uniquely identify it. Thus data items can map to multiple lines. A cache that can map data to n distinct locations is called n-way associative. In order to identify which data is in the cache, more bits have to be used in the tag ram. Accessing the cache now requires a search process as there are multiple lines in which a data item may be stored. In order to keep the speed of the lookup operation at the level of a direct mapped cache, the search must be done in parallel, requiring n comparators for an n-way associative cache.

There are three factors that make associative caches more expensive than direct mapped ones: the comparators needed, the increased size of the tag ram, and the need for a more complex replacement policy (see below). As these costs increase with the size of the cache, small on-chip caches usually employ a higher level of associativity than the larger external caches. Making a cache associative has its advantages: data items that would map to the same line in a direct mapped cache can now be stored in the cache at the same time. Trashing is reduced and utilisation improved.

2.3.2 Virtual vs. Physical Caches

Most systems today have a memory management unit (MMU) providing virtual address spaces. The cache can be placed between CPU and MMU, or between MMU and main memory. The former is called a virtual cache, as it only sees virtual addresses, while the latter is referred to as a physical cache. Both solutions have their specific (dis-)advantages:

- Access to virtual caches need not go through the MMU and is thus faster when a hit occurs.
- Virtual addresses might be larger than physical ones, making virtual caches more expensive.
- Virtual caches require more control logic as they have to deal with multiple address spaces.
- Physical caches can to bus-snooping in order to provide cache consistency.

Most CPUs today have the memory management unit integrated so that external caches are always physical. It is possible to improve the performance of a physical cache by using virtual addresses for indexing and physical addresses for tagging. This way, address translation in the MMU may be overlapped with cache operation. The resulting architecture is called a virtual cache with physical tags (see [Sch94]).

2.3.3 Replacement Policies

When new data is written to the cache, old entries may have to be evicted. Choosing a victim in a direct mapped cache is trivial as there is only one candidate. In associative caches, multiple solutions exist. The algorithm that chooses an item to remove is called the replacement policy. Replacement in cache systems is very much like page replacement in virtual memory systems. The policies used, however, are much simpler because they have to be implemented in expensive hardware and have to be fast.

LRU works by adding state information to the tag ram which is updated with every cache reference on the corresponding line. The line that was least recently used is selected for eviction. Strict LRU is usually used for two-way associative caches only as it would require too much state information for caches with higher associativity.

Pseudo-LRU is used in cache systems with associativity greater than two. The state information recorded is limited but still enough so that lines that are used often do not get evicted.

Random replacement requires no state information at all and still yields good results. It is best suited for large highly associative caches because it saves a lot of hardware while still performing well.

2.3.4 Write Policies

Write operations by the CPU may update both memory and cache, or the cache only. The first solution is called write-through and has the advantage that memory always contains up-to-date information. The second policy is termed write-back. It has the advantage of being faster, as the data need not be written to the slow main memory. Its disadvantage is that it leaves the memory in an inconsistent state, requiring additional logic in the case of parallel operations (e.g. DMA).

missing: write-allocate

2.3.5 Line Size

Another factor that determines the performance of a cache is its line-size. It determines the amount of data that is moved between main memory and the cache. Large cache-lines mean that data is prefetched, i.e. loaded into the cache before the CPU requests it. Assuming spatial locality of the memory references, this can improve performance and reduce contention of interconnection networks. Smaller lines, however, can be written back to memory faster. Choosing the right line-size means finding a tradeoff between possible performance gains and losses. Another factor are costs as smaller lines mean larger tags and thus require more hardware.

2.3.6 Cache Consistency

Caches introduce replication into the memory system. Whenever memory is accessed by multiple units, care must be taken that the information in the memory system stays consistent. Cache consistency in uniprocessor systems can be controlled by soft- or hardware. The only critical operations are interactions with peripheral devices. A simple solution is to mark all I/O-buffers as non-cacheable. The presence of multiple processors introduces new difficulties. Although software solutions would be possible (see [Sch94]), it is necessary to implement consistency protocols in hardware in order to achieve optimal performance.

Consistency of caches is implemented by enforcing a set of rules that determine when a data item may be cached. As long as the data is only read, there is no problem at all. On a write operation, however, measures must be taken to keep the view of the memory system consistent. If the location that is written to is cached, the cache entries have to be either updated or marked as invalid. Thus cache consistency protocols fall into two categories: write-update and write-invalidate.

Write-Update Protocols

missing

Write-Invalidate Protocols

missing

2.3.7 Prefetching

Multiprocessors today allow programs to prefetch data from main memory into caches by issuing special prefetch instructions. Prefetching is done in parallel to normal CPU operation and can thus help to hide memory latency.

2.3.8 Non-blocking operation

missing

2.4 Memory Models

missing

A = 0	B = 0
.	.
.	.
.	.
A = 1	B = 1
if(B == 0)	if(A == 0)
printf("B");	printf("A");

Chapter 3

Multithreading

This chapter deals with how multithreaded programs can exploit the parallelism provided by multiprocessors and what influence the operating system and the runtime system have on their performance.

3.1 Motivation

One goal of operating system design has always been to improve the utilisation of the underlying hardware. Thus batch-processing was superceded by multitasking. Multithreading is a refinement of traditional multitasking that separates the notions of a thread of control and an address space. The following advantages justify the use of this concept:

Increased Throughput A single-threaded program has to use non-blocking primitives if it wants to overlap communication with its execution, which leads to a dramatic increase in program complexity. Multithreading allows overlapping of communication and execution while using higher-level (i.e. blocking) services.

Efficiency and Ease of Communication Applications that are distributed over multiple processes have to use some kind of interprocess communication. Multithreaded applications automatically use the most efficient IPC-mechanism available – shared memory. In most applications, shared memory is also most intuitive to program.

Program Structure Designing an application as a set of cooperating threads helps to clarify program structure. Many problems are parallel by nature – servers might start one thread per request and numeric applications might divide their data into subsets that are worked on by cooperating threads.

Multiple Processors Multithreaded applications make use of hardware parallelism on a fine level of granularity. A single application program may be executed in parallel using multiple processors which reduces their execution time. Compared to traditional multitasking, utilisation of multiprocessor systems is improved.

Availability The interactivity of applications is increased by multithreading. A user need not wait for each of his commands to finish execution but can issue new requests while the system works on completing those he recently made. The need to show a "please wait"-message decreases.

System Resources Multithreaded applications make better use of system resources than multiprogrammed applications do. Threads need less state information than processes and context switches are much cheaper.

3.2 Drawbacks

Like every technique, multithreading has its limitations and drawbacks. The following summarises some aspects that have to be taken into consideration whenever a decision has to be made whether to use multiple threads or not:

- Some applications do not lend themselves very well to multithreading. Synchronisation and communication eat up much of the speedup achieved by parallelisation.
- Parallelising an application is a tedious job that introduces new potential programming errors. It is very difficult to test or proof the correctness of multithreaded code.
- Most operating systems today provide buffering and prefetching for I/O operations. The use of separate threads for I/O does not improve performance very much on such systems.
- Interactivity of an application can be achieved by using an event-driven programming style rather than threads. Ousterhood

3.3 Kernel- vs. Userlevel Threads

Threads can be implemented either as an operating system concept or as a user library. Both possibilities have their (dis-)advantages:

- Kernel threads are more expensive as they require administrative data in kernel space and have to do their context switches via the kernel. All functionality that any application might need has to be included in the kernel.

- Userlevel threads are not known to the operating system – kernel services can only be used by processes or kernel threads. If a userlevel thread calls a blocking system-call, all other threads are blocked with it. Signals cannot be sent to a single userlevel thread.
- With userlevel threads, scheduling is independent of the operating system and can thus be tuned to fit the applications needs. Synchronization between userlevel threads need not use expensive system calls.

To overcome these problems and combine the advantages of both concepts, it is now common to use kernel threads as "virtual processors" for userlevel threads. This, however, introduces a new level of complexity: How is the mapping between userlevel threads, kernel threads, and physical processors to be done? How can the kernel and the userlevel library interact in order to optimize system performance?

3.4 Scheduling

The scheduler is the entity in the operating system or runtime library that maps threads of execution onto processing elements. Its impact on system performance is significant. This section presents some aspects of scheduling systems and shows why traditional scheduling policies are unsuitable for multiprocessor systems. *Scheduling in uniprocessor systems is described in [Hof91, Tan92]. Issues that arise in multiprocessors and distributed systems are covered in [SS94].*

3.4.1 Preemptive vs. Nonpreemptive

The scheduler may be invoked whenever a thread of execution enters the kernel or the runtime library. If this is the only way to initiate scheduling, the system is said to be nonpreemptive. (in a cooperative system, the scheduler only runs when a process explicitly calls it). Although this scheme minimizes context switches, it does not optimize overall system performance as it has no way of assigning priorities (e.g. to I/O-bound tasks). Another problem is that a thread of execution may monopolize the CPU and decrease the system's interactiveness or even bring it to a standstill.

A solution to these problems is preemtiveness. Timer interrupts are used to periodically force the CPU into kernel mode, where the scheduler may be invoked. This increases flexibility but also means that an application cannot make any assumptions about when scheduling will take place. Some race conditions only occur in preemptive systems. Programmers have to be well aware of these potential problems.

3.4.2 Policy vs. Mechanism

The decision which thread of execution is to be scheduled next is independent of the mechanism that actually invokes the thread. Only the latter part of the scheduler has to be part of the operating system or runtime library. Policy (what is to be done?) may be separated from mechanism (how is it done?). Performance can be increased if scheduling decisions are made by the application programs themselves.

A program that enters a state in which is does a lot of computation might decide not to preempt threads. Thus the tradeoff between interactivity and throughput can be dynamically adjusted, reducing the number of context switches.

3.4.3 Scheduling Policies

The following paragraphs introduce some scheduling policies that are common in modern operating systems and runtime libraries:

Round Robin Each thread is assigned a time quantum. If it does not voluntarily relinquish the CPU before its quantum has expired, it is preempted and inserted at the end of the queue of runable processes. When a new thread needs to be selected for execution, the first in the runqueue is selected. The only important parameter in this scheme is the length of the quantum. If it is too long, responsiveness will suffer; if it is too short, context switches will degrade performance.

Priority Scheduling It may be necessary (or convenient) to give some processes priority over others. With priority scheduling, the process with the highest priority is run. To keep low-priority jobs from starving, the priority of a process that has just run is decreased. Priorities may be assigned statically or dynamically.

Multiple Queues This scheduling policy was developed for a system that had very expensive task-switches. The designers recognised the fact that long-running compute-bound processes should be run less often but with larger quanta. Multiple queues hold processes with different runtime behaviour.

Other Policies There are a lot of other scheduling policies, most of which are tailored for a specific purpose (e.g. realtime processing). Examples are shortest job first, guaranteed scheduling and two-level scheduling. Consult [Tan92] for further information.

3.4.4 Scheduling in Multiprocessor Systems

In multiprocessor systems, the scheduler has to map n processes onto m processors. Issues like caching and communication costs have to be taken into consideration. The main goal in todays systems is to keep all CPUs busy by providing data fast enough – CPU power is no longer the limiting factor for system performance.

Affinity Scheduling

Loadbalancing has to be done in order to keep all CPUs busy but removes threads from the caches that hold their data. On NUMA systems, a thread might even be migrated to another node that has to access its data remotely. Affinity scheduling [BS96, GTU91, TTG95] takes these aspects into account and tries to migrate only those threads that do no have much data in caches. Threads are kept on the same node and processor whenever possible. This scheme is very successful, especially on NUMA systems that have caches between hypernodes (for an example of such an architecture see [Con94]).

3.5 Synchronization

Support for synchronization of concurrent operations has to be provided at three levels: hardware, operating system, and programming language. Hardware support in the form of atomic instructions is needed, if multiple processors access shared resources such as shared physical memory. Multiprogramming operating systems have to deal with mutual exclusion between multiple threads of execution. Programming languages should provide means of abstraction for the services of hardware and operating system.

This section describes synchronisation in multithreaded environments. The basic synchronisation problems are discussed as well as the mechanisms that runtime systems provide to solve them. Finally, techniques that help to make synchronisation efficient on multiprocessor systems are discussed. *Synchronization at the software level is discussed in [Hof91, SS94, Tan92]. Hardware aspects are adressed in [HB84].*

3.5.1 Deadlocks and Starvation

A deadlock is a situation in which all processes in a system are blocked waiting for some resource. Because none of these processes will become runable, no resources can be freed and the system grinds to a standstill. Another problem arises if processes never become runable although resources are available. This situation is called "starvation". It is likely to occur in a system with static priorities. Deadlocks are examined in depth in [SS94, Tan92]. Deadlocks appear in the following situations:

- AB-BA deadlock
- recursive locking

more

3.5.2 Synchronization Problems

This section describes four synchronisation problems, each of which represents a class of similar problems. Almost every situation in which synchronisation is needed can be reduced to one of these "basic" problems:

Mutual Exclusion The basic problem of synchronization is that of mutual exclusion. Operations on shared data structures have to be "atomic", i.e. at any point of time, only one thread of execution may work on a given set of shared data. If mutual exlusion is not guaranteed, data inconsistencies may occur.

To model mutual exclusion, the notion of a "critical section" has been introduced. If a thread of execution needs to access a shared data object, it first has to acquire a lock (i.e. enter the critical section). Then it performs operations on the data and finally releases the lock (i.e. leaves the critial section) when it is finished and the data is in a coherent state again. Locks may either be associated with the data objects or to the code that modifies them.

Bounded Buffers This synchronization problem is also known as the producer-consumer problem. Producer threads send messages to consumer threads via a buffer of limited size. Clearly, access to the buffer has to be mutually exclusive. Another problem arises whenever a producer needs to send messages via a full buffer or a consumer wants to read from an empty one. In these cases, the processes have to wait for the buffer's state to change.

Readers/Writers Shared data may be read by multiple threads, but a thread may only write data if it has exclusive access, i.e. there are neither other writers nor readers. Two cases have to be distinguished: in the first, priority is given to the readers while in the second the writer may block arriving readers and thus takes priority. If readers are assigned priority, a writer may starve if new readers keep arriving.

Dining Philosophers Multiple threads may contend for limited resources, each needing more than one resource. The classical description of this problem is that of five philosophers alternately thinking and eating. For each philospher, there is a bowl of spaghetti but there are only five forks between the bowls. A philosopher needs both the left and the right fork in order to eat. Clearly, no two neighbouring philosophers may eat simultaneously and access to the forks needs to be synchronized. A situation like this is likely to cause deadlocks.

3.5.3 Synchronization Mechanisms

This section describes the four most common constructs used for synchronisation in multithreaded environments. They can be found in most runtime libraries and are implemented as normal API calls. Thus no support by the compiler is needed. *Programming language constructs for synchronization are discussed in [SS94]. A formal treatment of synchronization mechanisms can be found in [Hof91].*

Mutexes

Mutexes are synchronization objects that only have the two states "locked" and "unlocked". Atomic operations can be performed to lock and unlock mutexes. The lock operation blocks a thread until it acquires the mutex. Most implementations provide a trylock function that tries to lock a mutex but does not block on it. If the mutex was already locked, trylock simply returns an error-code. This functionality can be used to spin on a mutex.

Even scalar values may have to be protected by mutexes, as access to them might not be atomic on multiprocessors: if the size of the data is larger than the width of the memory subsystem or in case of misalignment, multiple memory accesses have to be performed in order to manipulate the data. Optimizations may be made if it is known that a scalar is accessed atomically, but code using such optimizations is not portable.

Whenever multiple mutexes have to be acquired, deadlocks may arise. They can be avoided by using lock hierarchies (i.e. acquiring locks only in a predefined order) or by using the trylock operation and releasing all other locks if it fails (a stategy called "backoff").

Mutexes are the basis for all other synchronization mechanisms. Their implementation is straightforward if hardware constructs like test-and-set instructions are used.

Condition Variables

Condition variables provide a mechanism to wait for a condition to be satisfied (the bounded buffer problem can be solved using condition variables). They are always used in conjunction with a mutex that provides mutual exclusion. After locking the mutex, the condition may be checked. If the condition is satisfied, the program unlocks the mutex and continues execution. Otherwise, it performs a wait operation that atomically inserts the thread into a list of waiting threads associated with the condition variable and unlocks the mutex. Whenever the data that is associated with a condition is changed, a signal operation has to be performed in order to wake up a thread that waits for such a change. Most implementations provide two more operations: a broadcast wakes up all threads waiting on the condition variable and timedwait allows threads to wait a certain time only.

Semaphores

were introduced by Dijkstra in 1968 [Dij68] as a mechanism for mutual exclusion and synchronization. Semaphores can be thought of as an object consisting of a signed integer value, a condition variable and a mutex. Two methods are defined, namely P and V, both of which are atomic (this is why the mutex is needed). P decrements the integer and blocks on the condition variable if the new value is less than zero. If the semaphore contains a negative number, then this designates the number of processes waiting on it. V increments the integer and unblocks one process from the condition variable if the new value is zero or less.

Mutual exclusion can be achieved with a semaphore by initializing it to one and protecting critical sections with a pair ov P and V operations. Semaphores used in this way are called binary semaphores.

Synchronization can be achieved by initializing a semaphore to zero. Now processes can block on the semaphore with P operations and an event can be signaled to one waiting process by invoking V.

Resource allocation can be controlled by initializing a semaphore to the number of resources available. Now processes can allocate and free resources by invoking P and V respectively.

Barriers

are used to synchronize a set of threads at a specific location in the program. Threads reaching a barrier wait until all other threads have also reached it.

more

3.5.4 Synchronisation in Multiprocessor Systems

revise

On uniprocessors, mutual exclusion can be achieved by making the kernel nonpreemptive (thus exactly on process can be in kernel-mode) and disabling interrupts during critical sections (thus keeping interrupt-handlers from interfering). All other synchronization problems can be solved building upon these concepts, e.g. ressources can be protected using the sleep/wakeup-meachnism that introduces condition variables into the operating system.

Synchronization on multiprocessors is a more complex theme. The non-preemption-assumption no longer holds and disabling interrupts on all processors of a system is hardly a good idea. Furthermore, it is desirable to have multiple processors execute kernel-code concurrently in order to achive higher parallelism. For a discussion of multiprocessor kernels see [Sch94].

Most computer systems provide at least some variation of a "Test-and-Set" operation. This instruction provides atomic access to one memory address. The old value can be read and subsequently modified with one operation. Shared memory multiprocessor systems need at least this basic form of synchronization in order to function properly.

Thread-Level Synchronisation

As parallelism increases, so does the need for synchronization. In a multithreaded environment, very lightweight mechanisms have to be provided in order to achive a maximal performance improvement. Threads rather than processes should be blocked on resources and synchronization among userlevel threads should be implemented at this level. more

Spinning vs. Blocking

Spinning (aka. "Busy Waiting") is usually avoided as it waists precious CPU cycles and memory bandwidth. If however, the process only has to wait for a short period of time, blocking it would be a bad idea. The cost of a task-switch may then be greater than the cost for spinning. If the time that a process has to wait cannot be predetermined, a strategy might be employed that spins for a specific time and then blocks.

Chapter 4

Performance Measurement

This chapter deals with the art and science of performance evaluation. The first section provides a broad overview of performance evaluation, the context in which measurement stands. The second section describes theoretical aspects of performance measurement as well as current methodology and tools. Tuning of application software and the runtime environment is discussed in the last section. *For further information about Performance Measurement see [FSZ83, McK88, Lan92, Jai91, ea95, SK90]*

4.1 Introduction to performance evaluation

Performance can be defined as a sum of properties of a system that make it fit for a specific purpose. The individual properties are called performance indices. Different people regard different aspects of a system as important and each person's view might change in different contexts. Thus, performance is a weighted sum of performance indices:

$$P = \sum_i w_i p_i$$

The goal of performance evaluation is the improvement of the absolute performance or of the cost/performance ratio. Even if absolute performance is the primary goal, only limited resources are available to achieve it. Thus, speedup and efficiency are important:

$$S = \frac{P^\star}{P} \text{ and } E = \frac{P}{C}$$

Speedup is the ratio of the modified system's performance to the original performance. Efficiency is performance per cost.

System performance is limited by the weakest link in the chain, in computer science termed a bottleneck. As we eliminate one bottleneck, others show up while the costs of improving the performance of a system normally

Level	Performance Index
System	throughput, capacity, availability
Program	time to completion, responsiveness
Process	number of page faults, memory usage
Language block	execution time, number of executions
Operating System	cost of context switches, resource utilisation
OS Resource	utilisation, average number of waiting processes
Hardware	cache efficiency, bandwidth, cycle time

Table 4.1: Examples of performance indices

increase with every step of improvement. Thus, every system has a state of optimal operation, where the cost per performance is minimal or where the cost of performance improvement becomes unacceptable.

Information processing systems are usually structured into multiple layers, each of which provides an increased level of abstraction. Accordingly, performance indices can be defined in each of these layers. Some examples are given in table 4.1. Evaluation of a system's performance is needed in all stages of its life-cycle:

Design: The developer has to repeatedly evaluate the current state of a design until the goals have been reached.

Improvement: No product coming to market is perfect. Problems not encountered in the design phase now become visible and new demands develop. Thus the design process has to continue with this new input.

Acquisition: Before a system is bought, it has to be verified that is meets the requirements.

Operation: Most systems are parameterised in some way. To attain maximum performance, these parameters have to be tuned.

Capacity planning: A system's workload will probably increase during operation. The operator has to calculate when the maximum workload will be reached and how much added performance has to be bought or whether the system has to be replaced.

There are four basic techniques for evaluating the performance of a system, differing greatly in requirements, cost and precision:

Analysis of mathematical models A system is described im terms of mathematical formulas that describe its behaviour.

Simulation A model of the system is generated and each step of operation is simulated and evaluated.

Measurement The behaviour of (parts of) the system is measured under reproducible conditions. This requires that the system be operational.

Benchmarking The time that specific programs need for execution is measured. These programs may be existing applications or artificially designed benchmark programs. Benchmarking can be seen as a special case of measurement.

Of these, measurement is the one examined in this thesis. The following two sections provide an overview of the theory and practice of performance measurement.

4.2 Measurement Methodology

The goal of performance measurement is to gain insight into the dynamics of a system's operation. In contrast to an examination of the (static) description of a system, performance measurement can provide information about its runtime behaviour and thus lead to performance improvements and the discovery of race-conditions. The measurement process consists of five basic steps:

Instrumenting the system: sensors and stimuli are attached to the system in order to extract information and to influence the target system.

Generating traces and profiles: traces record the chronology of events in the system while profiles count them or measure durations.

Storing the collected information: interference with the systems operation must be minimised.

Processing the data: measurement will usually yield a huge amount of data that must be post-processed.

Visualising the results of processing: collected data has to be presented to the user in a way that emphasises important information.

Common characteristics of measurement tools are: more

Interference: Measurement tools influence the measured system. This interference has to be kept to a minimum.

Accuracy: The system must be able to collect as much data as the user wishes. Recorded values must be as precise as possible and error ranges must be known.

Level of Abstraction: Measurement can record data about objects on any level of a systems hierarchy – from logic gates to complex application programs.

Data reduction capabilities: It must be possible to reduce the amount of data that is collected to a meaningful quantity. Recording all available data might hide the important facts.

Scope: A measurement system that is tightly fitted to just one application cannot be reused and its price will probably be too high compared with general-purpose systems.

Compatibility: Target- and measurement system have to interact and thus either have to be built using the same basic technology or be adapted to each other.

Integration: Measurement is mostly done in the context of a design process which it has to be integrated into. An ideal system does not interrupt the design process but rather accompanies it.

Ease of use: Of course, a measurement system has to have a user interface that makes it easy to use. Another issue is the turnaround time, i.e. the time between the specification of the measurement parameters and the display of the results. So-called offline systems display data only after measurement is completed. Online systems display it directly and may even support interactive changing of measurement parameters.

Costs: Costs are always an issue. A system that costs more than it is worth is unlikely to be used. One must always consider the running costs of the system which depend on its resource usage and its ease of use.

4.2.1 Hardware monitors

Hardware monitors provide means to trace the low-level operations of a system. They can collect data almost without interfering with the target system and their accuracy is very high. Unfortunately, hardware monitors are mostly highly specialised tools that have a limited range of applications, which makes their use very expensive.

4.2.2 Software monitors

Very often, the advantages and drawbacks of software complement those of hardware. Measurement programs can be far more flexible, adaptable, easy to use and easy to integrate. They can be significantly cheaper. Their drawbacks are in the fields of interference and accuracy. Software solutions have to use existing hardware and thus contend for resources with the processes that are to be measured. Influence to the target system is high and the amount of data that can be collected very limited.

4.2.3 Hybrid monitors

Hybrid monitors are a way to alleviate the deficiencies of software monitors. Special measurement hardware is introduced into the target system that allows the collection of low-level data that would not be accessible to a pure software measurement system. Another purpose of the hardware component may be to reduce the influence that the measurement system takes.

Most microprocessors today have at least some support for performance measurement. There are usually several counters for events like cache-misses, page-faults, or pipeline stalls (for an example see [WG94]).

4.3 Tuning

reorder

Performance is measured in order to improve a systems behaviour. Whenever needs are not satisfied, the system must be changed. One way of improving performance is to buy a more powerful machine. Although this option is theoretically available in most cases, it is seldom the most cost-effective way. There are two causes for bad performance and thus two ways to improve the behaviour of an existing system:

4.3.1 Program Behaviour

The most obvious question when addressing performance problems is: Does the application software make optimal use of the computers resources? It is so obvious that many people tend to shut their eyes to it. Let me illustrate this with a short example:

find a better example

Most software today makes no use of parallelism at all. Thus, although multiprocessing can improve the systems overall performance by running applications in parallel, an individual application cannot profit from it. In todays workstation-oriented environments, there mostly is only one active process. A workstations resources are unused most of the time and overloaded whenever the user makes a request. Nevertheless, the workstation-model is quite popular.

Here are some examples of how an individual application program can be improved:

- Parallelism can be introduced by splitting it into multiple processes or threads. This can be a reasonable step even if the program runs on uniprocessors as it may improve the applications responsiveness and hide the latency of communication operations [].

- Locking of shared resources may be made finer in order to achieve more parallelism [Sun94, page 116].

- Alternative algorithms may be considered. If memory usage is the critical factor, a suboptimal algorithm (in terms of theoretical time complexity) may be appropriate.
- On a NUMA machine, memory allocation may be modified in order to bring data closer to the processor working on it. []

Another possibility is to improve the behaviour of an application program by changing its environment:

- Most operating systems have a large number of tuneable parameters that may be adjusted to best fit the most commonly used applications.
- Scheduling is an important factor. Most of todays systems use static schemes that do not take the program's behaviour (e.g. its usage of caches) into account. A scheduling policy that adapts to demands of the running programs can bring great performance improvements [Ste95, page 14].
- In most systems that use kernel threads as virtual processors for the user threads, concurrency is reduced whenever a thread makes a blocking system call. This may even lead to deadlocks if the number of running kernel threads reaches zero. There are ways to keep the number of virtual processors and thus concurrency constant [Kop95].
- Synchronisation mechanisms may be made more efficient by introducing a strategy of spinning for some time and then blocking [Ste95, page 46].

Tuning need not be an action that is performed only once. Systems have been built that monitor a systems operation and try to optimise it without intervention by the user [Yan88].

4.3.2 Bottlenecks

Performance is limited by the weakest link in the chain: a bottleneck. Buying faster CPUs will not bring a performance gain if the limiting factor is memory bandwidth. Bottlenecks can be found in both hard- and software. Here are some examples:

Software

- Most libraries have been made thread-safe by simply adding one synchronisation object to each subsystem (e.g. the malloc calls). An application using a lot of dynamic data structures looses parallelism to this simple scheme.

- Centralised kernel data structures may become a major bottleneck in multiprocessor systems. This is especially bad as it affects all processes.

- If all request to a client-server database are handled by a single process then this will likely be a bottleneck.

Hardware

- The performance of the memory subsystem and of the communication paths between processors is today's major bottleneck.

- Mass storage may be a bottleneck in applications that require large amounts of data but do not perform complex operations on them.

- A MMU may be a bottleneck if it has to access main memory too often to provide the processor with data fast enough.

How can the bottleneck problem be addressed? There are two basic ways: Bottlenecks may be eliminated (e.g. by replicating the critical resource) or they may be hidden (e.g. by introducing caches).

4.3.3 Improving Software

Only seldom is it possible to make changes beneath the level of the application software. Better hardware may be unavailable or too expensive. Changing the operating system may be even more expensive. Even if there was a better system, switching to it would mean that applications would have to be rewritten, people be retrained, and so on. A solution that is feasible is performance tuning of the application software.

Optimising Software for Cache Efficiency

Performance of a computer system can be improved if software is optimised to make good use of the cache subsystem. The primary goal is to maximise the cache hit-ratio. Causes for cache misses fall into three categories (called the "three C's" [HP96]):

Compulsory: Whenever a data item is accessed for the first time, a cache miss occurs. There is no way to prevent these so-called cold start misses.

Capacity: If a program references more data than fits into the cache, misses occur because cache-lines have to be evicted to make room for others. A way to alleviate this problem is to increase the temporal locality of the data (e.g. by reordering loops).

Conflict: Data items mapping to the same cache lines may expel each other from the cache. The higher the associativity of the cache is, the less severe is this problem. Programmers can avoid collision misses by allocating data structures that are used together in one block.

Compilers can perform code optimisations that improve a programs utilisation of hardware resources. Higher level reordering, however, still has to be done by the programmer. Hennessy and Patterson [HP96] present four basic optimisations that improve cache use:

Merging Arrays: Cache use can be improved if data structures that are accessed together are merged or at least allocated together. The first optimisation increases spatial locality while the second simply reduces the probability of collision misses.

Loop Interchange: Programs accessing data in nested loops often trash caches because the data items are not addressed in the order in which they are stored. Interchanging inner and outer loops can yield great performance improvements.

Loop Fusion: If a program accesses the same data in different loops in order to perform different computations, temporal locality may be improved by joining these loops.

Blocking: The last optimisation technique tries to eliminate capacity misses by dividing the problem that is to be solved in multiple subproblems. An good example for this is matrix multiplication which can be done block-wise and results in a very unfortunate access pattern when implemented naively.

Improving Multithreaded Code

Re-blocking

Optimizing Synchronisation

Improving the Runtime System

Chapter 5

Socrates

Developing multithreaded programs is a difficult and still ill understood task. Only few tools are available that aid the programmer in designing correct and efficient code. Socrates is an environment that allows programs to be written in an architecture-independent way and their performance to be evaluated. Figure 5.1 shows an overview of the system.

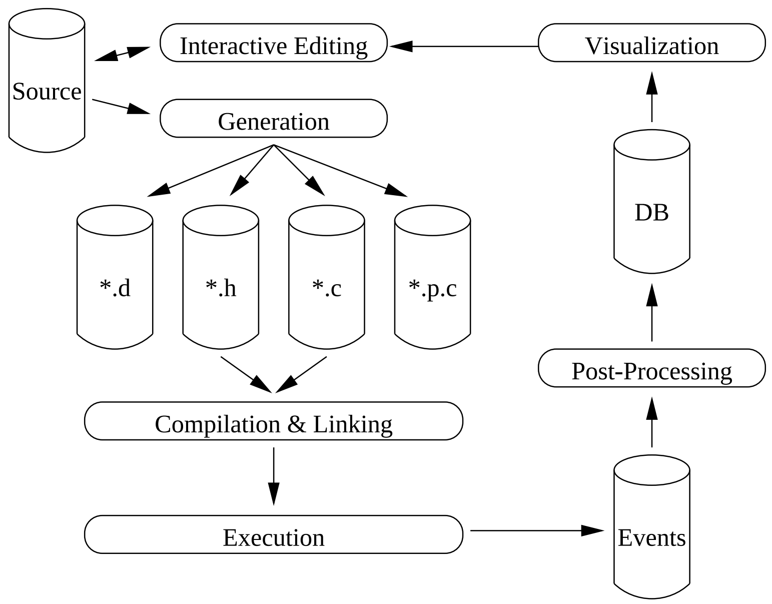

Figure 5.1: Overview of the Socrates system

Applications are written in ANSI C using a graphical environment. Support for multithreading and performance measurement is added in an orthogonal way. Instead of extending the language to support parallelism, a generator has been built that inserts the necessary code into the source. Chapter 5.1 will show this in detail.

The graphical user interface allows the programmer to browse and edit the source code and specify instrumentation parameters. The user must specify the set of events that are to be generated during execution. This step is described in chapter 5.2.

The code that is generated can be compiled using an unmodified C compiler but may have to be linked with modified versions of the runtime libraries and be run using an extended system kernel. During the execution of the program, an event trace is recorded and saved in a file in raw form. This process is discussed in chapter 5.3.

The generator produces a post-processing program which inserts the raw data into an SQL database. This allows the programmer to perform complex queries using a standardised language. Socrates does not provide support for the analysis of the data as there exist a wide variety of programs for this purpose. Data may be exported to spreadsheet applications, statistical analysis systems, or plotting software. Chapter 5.4 discusses the benefits and costs of this approach.

An example program which illustrates the whole process of writing multithreaded code, instrumenting it, generating for a specific platform, and analysing the resulting data, is presented in chapter 5.5.

reflect this in the corresponding chapters

5.1 Programming with Socrates

5.1.1 The User Interface

The user interface of Socrates allows the programmer to browse the source code and edit it. The code is split multiple parts: types, variables, functions, and so on.

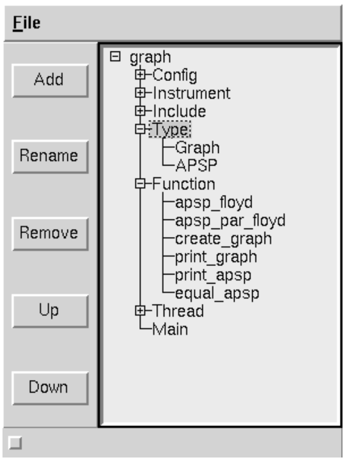

Figure 5.2: Socrates User-Interface

5.1.2 The Generator

Once a program has been written using Socrates, C source files may be generated that can be compiled using an unmodified C compiler. Declarations and definitions are separated and written to the corresponding files.

Socrates contains a preprocessor that is able to manipulate the code that is generated. It allows arbitrary Tcl code to be inserted into the C source code. Anything between two at-signs is passed to the Tcl interpreter, the return value being inserted into the output of the generator. This scheme makes Socrates very easy to extend and adapt to new environments. It also provides the programmer with more powerful preprocessing functionality than the standard C preprocessor does. In the current version, Tcl commands cannot be nested. This will change in the future, making the preprocessor even more powerful.

5.1.3 Support for Multithreading

Support for writing architecture-independent multithreaded programs is implemented by the use of the preprocessor and Tcl commands that generate code for the chosen platform. In the code browser, threads exist as abstract entities and are thus separated from normal functions. Like functions, they have arguments and variables. Unlike functions, they have attributes that specify whether they are bound to a processor, detached from their patent thread, and so on.

An example of a call to the thread library is @thr_self@ which produces thr_self() when Solaris threads are used and pthread_self() with POSIX threads. Socrates provides the same functionality for all runtime libraries. This does not mean that only the common subset is supported as some functionality that is missing in a library may be emulated. Solaris threads, for example, do not have a mechanism for barrier synchronization but code containing barriers may still be generated for this platform. The preprocessor is able to generate code that implements barriers using mutexes and condition variables.

Appendix A contains a table of all thread operations and synchronisation mechanisms supported by Socrates.

5.2 Instrumenting Programs

5.2.1 Overview

The measurement process can produce too much data. Some systems address this issue by letting the user filter events after the measurement process. As Socrates uses the same hardware as the target program, a goal of the instrumentation must be to reduce the amount of data that is generated.

The user may define which classes of event are to be generated. Only the corresponding code is instrumented.

5.2.2 Function Calls

5.2.3 User Events

5.2.4 Thread Operations

5.2.5 Synchronisation Objects

5.3 Execution

5.3.1 Collecting Event Traces

During execution of the program, an event trace is generated and stored in a file. Here are a few things that the user has to keep in mind:

- In order to use Socrates, it may be necessary to boot special versions of the operating system kernel. Using Solaris, a version of the kernel must be used that allows the manipulation of the performance counter registers and saves their contents during all context switches.
- The event trace file should be stored on the fastest disk that is available. If there is enough memory available, it is best to use a ram disk. This can speed up the process dramatically and help to reduce interference with the application program.
- The system should be unloaded during measurement. Other processes may interrupt the program under inspection and thus falsify measurement results (e.g. by evicting data from the caches).

5.4 Post-Processing and Visualization

The post-processing program `pp` transfers the event data into an SQL database. No filtering or other manipulation of the data is done. This is left to the database system because it provides high-level functionality for exactly this type of operation.

As soon as the data is loaded into the DBMS, the user may to perform complex queries that are not normally supported by performance evaluation tools. An example might be a program that checks the events for the occurrence of deadlocks or starvation situations. Freedom, of course, means work. Interaction with the system is done using SQL which may be embedded into other programming languages such as C or Tcl.

There are, of course, some programs that perform standard tasks like determining the number of threads in existence, the number of threads waiting on a condition variable, and so on. They work for every application program and can be used as starting-points for more specialised analysis tools.

5.4.1 Example: Barriers

This example program written in Tcl/Tk counts the number of threads waiting in a barrier.

Code barriers.graph

```
#!/bin/sh
# the next line is executed by sh, but not by tcl
# because for tcl, it is a continuation of this comment \
exec pgtclsh $0 ${1+"$@"}
```

The pgtclsh is a Tcl-shell linked with routines that allow access to a Postgres server. A Shell is used to search is using the PATH environment variable.

```
set dbhandle [pg_connect "graph"]
set plotfd [open "graph.db" w]
```

A connection to the postgres server is established using the pg_connect command. The resulting handle is stored in the variable dbhandle. Likewise, a file is opened for the results of this script.

```
set res [pg_exec $dbhandle
        "select address from barrier_init where name = '${barrier}';"]
set address [pg_result $res -getTuple 0]
pg_result $res -clear
```

Barriers are identified by their memory address. During initialization, an event is generated that contains an identifier for the barrier. This is now used to map the name of the barrier to its address. In applications that allocate barriers dynamically, the mapping of addresses to identifiers may not be one to one.

```
set res [pg_exec $dbhandle
        "select max(time) from barrier_waited where address = ${address};"]
set maxx [pg_result $res -getTuple 0]
pg_result $res -clear
set res [pg_exec $dbhandle
        "select min(time) from barrier_wait where address = ${address};"]
set minx [pg_result $res -getTuple 0]
pg_result $res -clear
```

In order to scale the time axis of the graph, the minimum and maximum timestamps are determined.

```
set res [pg_exec $dbhandle
        "create table barrier_events (time float8, event char, eid int);"]
pg_result $res -clear
set res [pg_exec $dbhandle
        "insert into barrier_events select time,'e',eid \
         from barrier_wait where address = ${address};"]
```

```
pg_result $res -clear
set res [pg_exec $dbhandle
         "insert into barrier_events select time,'l',eid \
          from barrier_waited where address = ${address};"]
pg_result $res -clear
```

A table is created that holds all barrier events with their timestamp, event id, and a tag that identifies whether the barrier was entered or left.

```
set res [pg_exec $dbhandle
         "select time - (cast '${minx}.0' as float8) as time,event,eid \
          from barrier_events order by time,eid;"]
set ntups [pg_result $res -numTuples]
```

Now the table of barrier events is read. Time is normalized to begin with zero and the events are totally ordered by time and event id. The resulting handle will be used to transfer the data from the Postgres server.

```
set nthreads 0

for {set i 0} {$i<$ntups} {incr i} {
set e [pg_result $res -getTuple $i]
if {[lindex $e 1] == "l"} {
incr nthreads -1
} else {
incr nthreads
}
puts $plotfd "[lindex $e 0].0 $nthreads"
}
```

Iterating over all events, the number of threads in the barrier is determined for every event. The result is written to the output file.

```
pg_result $res -clear
pg_disconnect $dbhandle
close $plotfd
```

The last result handle is cleared, the database connection ended, and the output file closed.

The results of such a program may be further processed with programs for statistical analysis like SPSS or Origin. They can be visualised using a data plotting program like gnuplot, xmgr, or plotmtv. Figure 5.3 shows a plot that was generated from the output file of the above example using xmgr.

find GNU
soft

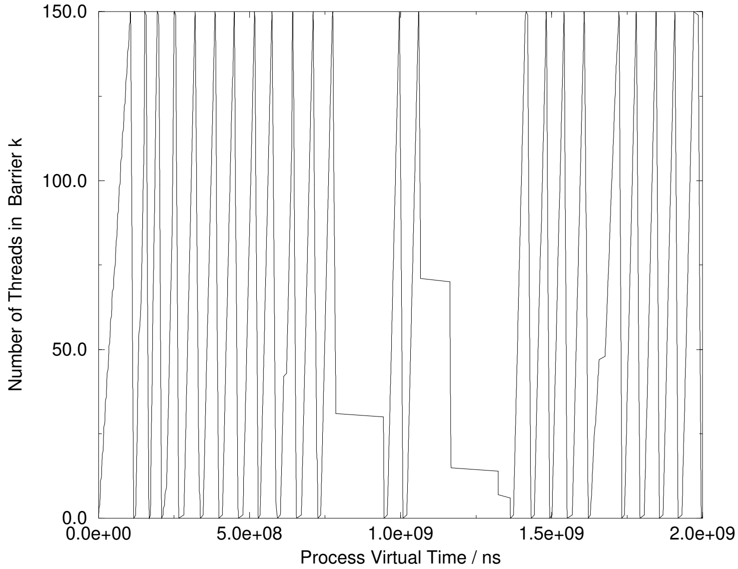

Figure 5.3: Barrier Synchronizations.

5.5 Example: All Pairs Shortest Paths

This sample program implements a simple parallelised version of Robert D. Floyd's algorithm [Flo62, Jun94] for finding the shortest path between all pairs of nodes in a directed weighted graph.

5.5.1 The Algorithm

Data Types: Both the graph and the shortest paths found so far are stored in $n \times n$ matrices where n is the number of vertices in the graph. The elements store information about the existence and length of a path (i, j) between nodes i and j. A barrier b is provided for synchronisation between the worker threads.

Initialisation: For each pair of nodes i and j, the result matrix is initialised to d if there exists a vertex (i, j) with length d and to ∞ if no vertex

(i, j) exists. The barrier b is initialised to n.

Worker i: Each worker is assigned a row of the result matrix that it works on. It then repeats the following code $k = 1 \ldots n$ times:

- For each j see if there exists a path from i to j via node k and, if it is shorter than the shortest path found so far, store it in the result matrix.
- Synchronise with the other threads at barrier b.

5.5.2 The Implementation

The following shows an implementation of the algorithm given above.

```
Thread Worker {

Arguments {
int i,
APSP* apsp
}

Variables {
int j,k;
int r;
}

Definition {
for(k=0;k<apsp->n;k++) /* make k runs through the algorithm */
{
  for(j=0;j<apsp->n;j++) /* for each (i,j) with i fixed per thread */
  {
    if((i!=k)  /* nothing to be won in these cases */
     &&(j!=k)
     &&(i!=j)
     &&(apsp->d+apsp->n*i+k)->exist /* if ex. (i,k) and (k,j) */
     &&(apsp->d+apsp->n*k+j)->exist)
    {
      float d;
      if(i>k)
      {
        @mutex_lock (apsp->d+apsp->n*i+k)->m r@;
        @mutex_lock (apsp->d+apsp->n*k+j)->m r@;
      }
      else
      {
        @mutex_lock (apsp->d+apsp->n*k+j)->m r@;
        @mutex_lock (apsp->d+apsp->n*i+k)->m r@;
      }
      d = (apsp->d+apsp->n*i+k)->dist  /* d = |(i,k)|+|(k,j)| */
                + (apsp->d+apsp->n*k+j)->dist;
      @mutex_unlock (apsp->d+apsp->n*k+j)->m r@;
      @mutex_unlock (apsp->d+apsp->n*i+k)->m r@;
```

```
      if((apsp->d+apsp->n*i+j)->exist) /* was there a path before? */
      {
        if(d<(apsp->d+apsp->n*i+j)->dist) /* see if d is shorter */
        {
          (apsp->d+apsp->n*i+j)->dist = d; /* new shortest path found */
        }
      }
      else /* new path was found */
      {
        @mutex_lock (apsp->d+apsp->n*i+j)->m r@;
        (apsp->d+apsp->n*i+j)->exist = 1; /* (i,j) now exists */
        (apsp->d+apsp->n*i+j)->dist = d; /* and has length d */
        @mutex_unlock (apsp->d+apsp->n*i+j)->m r@;
      }
    }
  }

  @barrier_wait apsp->k@; /* synchronize all workers */
  @barrier_wait apsp->k2@ /* before starting a new run */
}
@barrier_wait apsp->finish@; /* synchronize all threads on exit */
@thr_exit 0@; /* exit */
}
```

5.5.3 Measurement

Mutex Operations

Every access to the result matrix r is protected by a mutex. Figure 5.4 shows the number of threads waiting for or holding a mutex vs. time. As can be seen from the graph, contention for mutexes is high. A solution to this performance problem would be to replace mutexes by reader-writer locks, thus allowing for more concurrency.

Scheduling

Scheduling has a great influence on program performance. Figure 5.5 shows when a userlevel thread changes its LWP and its CPU. As the Solaris thread library does not use memory conscious scheduling, there is only little affinity to the caches.

Cache Utilisation

Modern microprocessors provide hardware support for performance measurement in the form of event counters that are able to record such information as the number of cache references and hits. Figure 5.6 shows the cache usage of a single userlevel thread. The cache hit rate is about 93 percent.

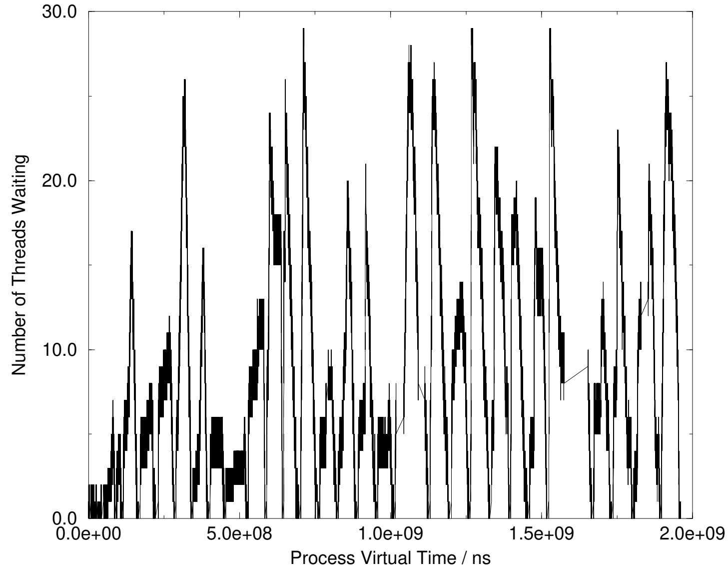

Figure 5.4: Number of threads waiting for mutexes.

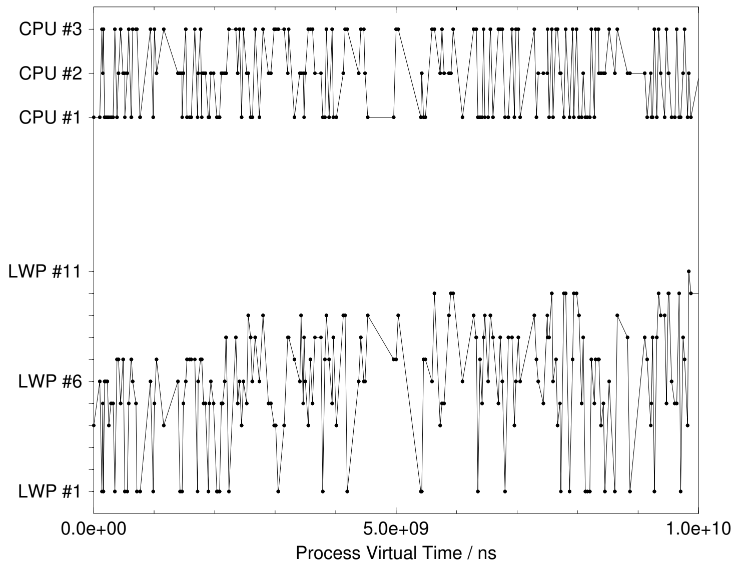

Figure 5.5: Scheduling of a userlevel thread.

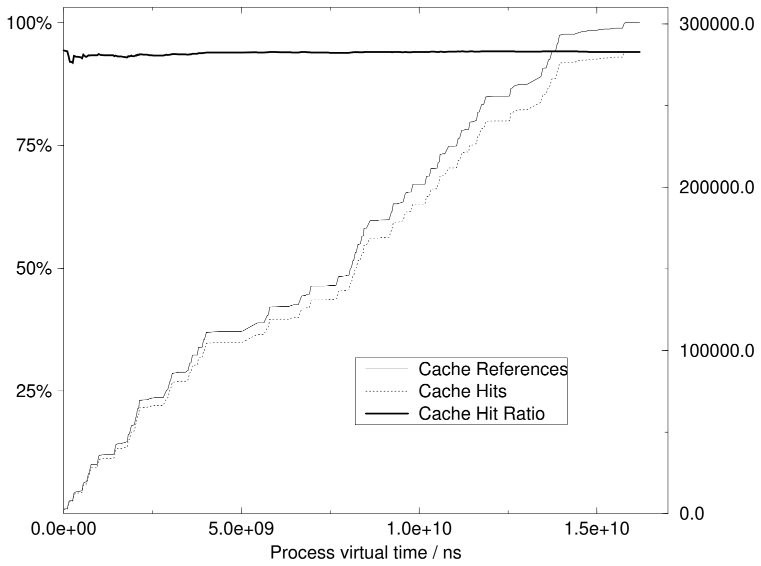

Figure 5.6: Cache usage of a userlevel thread.

5.6 Implementation

5.6.1 Implementation

Events that are generated by the instrumented program are collected in buffers. Whenever a buffer is full, it is inserted into a list. A writer-thread which is bound to its own LWP scans the lists and writes their contents to the event trace file. Generation of events does not introduce much overhead into the application's code as almost all blocking operations are done by the writer-thread. Figure 5.7 shows this process.

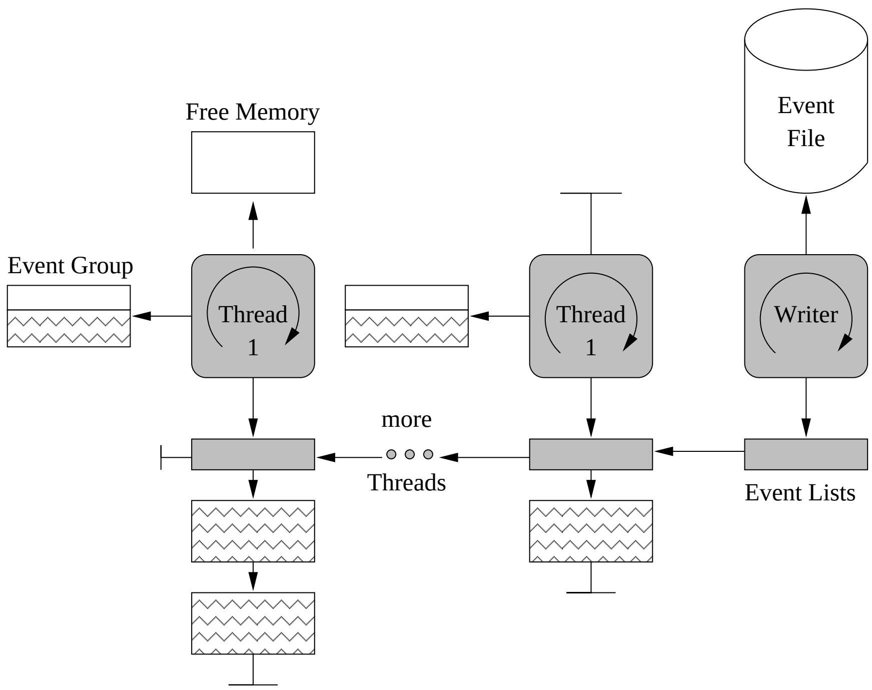

Figure 5.7: Generating and writing event traces.

Chapter 6

Conclusion

What has been achieved?

Not yet bug-free User interface lacks integration

Future Work

As it is, Socrates supports only one platform. Ports are planned to the MThreads library [Ste95] both on UltraSparc machines and the Convex SPP1600. This will allow a comparison between different thread libraries. It is hoped that the benefits of memory conscious scheduling can be shown and different strategies be evaluated.

The user interface lacks integration. The goal is to combine all parts of the system into one application. Ultimately, all steps of program development shall be integrated into a single environment.

Appendix A

Socrates Support for Multithreading

A.1 Thread Management

Data Type	Description
thrid	The type of an identifier for a thread

Function	Parameters	Description
thr_start	*name*	name of the thread to start
	arg (*name*_args*)	pointer to the argument structure
	options	list of options
	thrid	thread identifier
	result	result
thr_exit		
thr_join		
thr_self		
thr_suspend		
thr_continue		

A.2 Thread-Local Variables

Function	Parameters	Description
tgv_create		
tgv_delete		
tgv_get		
tgv_set		

A.3 Mutexes

Data Type	Description
mutex	

Function	Parameters	Description
mutex_init		
mutex_destroy		
mutex_lock		
mutex_unlock		

A.4 Condition Variables

Data Type	Description
condv	

Function	Parameters	Description
condv_init		
condv_destroy		
condv_wait		
condv_signal		
condv_broadcast		

A.5 Semaphores

Data Type	Description
sema	

Function	Parameters	Description
sema_init		
sema_destroy		
sema_wait		
sema_trywait		
sema_post		

A.6 Barriers

Data Type	Description
barrier	

Function	Parameters	Description
barrier_init		
barrier_destroy		
barrier_setnothr		
barrier_wait		

Appendix B

Existing Performance Measurement Tools

B.1 CXtrace

B.2 tha/tnf

B.3 Paradyn

B.4 Pablo

B.5 JED

Appendix C

Sun Enterprise X3000

The machine used for development and testing of Socrates was a Sun Ultra Enterprise X3000. The following sections provide a broad overview of the systems architecture and the particular configuration used; more information can be found in [Sun96].

Memory Subsystem

On-Chip Instruction Cache: 16KB, 2-way associative, line-size of 32bytes

On-Chip Data Cache: 16KB, direct-mapped, line-size of 32 bytes with 16 byte subblocks, write-through with no allocate on write miss, non-blocking with up to nine outstanding load operations

External Unified Cache: 512KB, direct-mapped, line-size of 64 bytes with 16 byte subblocks, write-back, cache controller integrated in the cpu, access by the cpu is pipelined

System Bus Architecture

Configuration

- 3 UltraSparc I modules at 167MHz

Bibliography

[Bac86] Maurice J. Bach. *The Design of the Unix Operating System.* Prentice-Hall, Inc., 1986.

[BS96] Frank Bellosa and Martin Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):113–121, 1996.

[Con94] Convex Press. *Convex Exemplar Architecture*, 2nd edition, 1994.

[CT96] Alan Chalmers and Jonathan Tidmus. *Practical Parallel Processing.* International Thomson Computer Press, 1996.

[Dij68] E. W. Dijkstra. Co-operating sequential processes. *Programming Languages*, pages 43–112, 1968. Citation taken from [CT96].

[ea95] Rainer Klar et al. *Messung und Modellierung paralleler und verteilter Rechensysteme.* B.G. Teubner, Stuttgart, 1995.

[Flo62] Robert W. Floyd. Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[FSZ83] Domenico Ferrari, Giuseppe Serazzi, and Alessandro Zeigner. *Measurement and Tuning of Computer Systems.* Prentice-Hall, Inc., 1983.

[GTU91] Annop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods of the performance of parallel applications. *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 19(1), 1991.

[Han95] Olav Hansen. *Leistungsanalyse paralleler Programme.* Spektrum Akademischer Verlag GmbH, 1995.

[HB84] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw Hill, Inc., 1984.

[Hof91] Fridolin Hofmann. *Betriebssysteme: Grundkonzepte und Modellvorstellungen.* Leitf"aden der Angewandten Informatik. B.G. Teubner, Stuttgart, 1991.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 2nd edition, 1996.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, 1991.

[Jun94] Dieter Jungnickel. *Graphen, Netzwerke und Algorithmen.* BI-Wissenschaftsverlag, 3rd edition, 1994.

[Kai96] Richard Y. Kain. *Advanced computer architecture: a systems design approach.* Prentice-Hall, Inc., 1996.

[Kop95] Christoph Koppe. Sleeping threads: A kernel mechanism for support of efficient user level threads. In M. H. Hamza, editor, *Proceedings of the 7th IASTED/ISMM International Conference Parallel and Distributed Computing and Systems.* IASTED-ACTA PRESS, 1995.

[Lan92] Horst Langendörfer. *Leistungsanalyse von Rechensystemen.* Hanser Studienbücher der Informatik. Carl Hanser Verlag, 1992.

[McK88] Philip McKerrow. *Performance Measurement of Computer Systems.* International Computer Science Series. Addison-Wesley, 1988.

[Qui94] Michael J. Quinn. *Parallel computing: theory and practice.* McGraw-Hill, 2nd edition, 1994.

[Sch94] Curt Schimmel. *Unix Systems for Modern Architectures.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, 1994.

[SK90] Margaret Simmons and Rebecca Koskela, editors. *Performance Instrumentation and Visualization.* ACM Press, 1990.

[SS94] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems.* McGraw Hill, Inc., 1994.

[Ste95] Martin Steckermeyer. *Einbeziehung von Lokalitätsinformation in Mechanismen zur Prozeßverwaltung auf Benutzerebene.* Diploma thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1995. DA-I4-95-25.

[Sun94] Sun Microsystems, Inc. *Multithreaded Programming Guide*, 1994.

[Sun96] Sun Microsystems, Inc. *Ultra Enterprise X000 Server Family: Architecture and Implementation*, 1996.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Inc., 1992.

[TTG95] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24:139–151, 1995.

[Wal95] Klaus Waldschmidt, editor. *Parallelrechner, Architekturen - Systeme - Werkzeuge*. B.G. Teubner, 1995.

[WG94] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., 1994.

[Yan88] J. C. Yan. *Post-game Analysis – A Heuristic Resource Management Framework for Concurrent Systems*. PhD thesis, Department of Electrical Engineering, Stanford University, 1988.