Benchmarking significant DBMS costs on Niagara in order to perform a relative performance comparison between the Shared Nothing and the Shared Everything DBMS memory architectures

Lars-Erik Bjørk
Truls Rinnan Jørgensen

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Svein-Olaf Hvasshovd, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The trend in processor development is to include multiple cores, each with multiple native threads on the same die. The Niagara processor from Sun Microsystems is such an hardware architecture. The design of a DBMS is highly dependent of the choice of the underlying memory architecture. Two such memory architectures are Shared Nothing and Shared Everything.

The goal of this report is to carry out a relative performance comparison between these memory architectures on the Niagara processor. In order to do this, the most significant DBMS costs for each memory architecture must be benchmarked on Niagara. The benchmarked costs may then be used to calculate the performance of both memory architectures. In addition to Shared Nothing and Shared Everything, it would also be interesting to assess alternatives to these memory architectures.

Assignment given: 23. January 2006
Supervisor: Svein-Olaf Hvasshovd, IDI

Benchmarking significant DBMS costs on Niagara in order to perform a relative performance comparison between the Shared Nothing and the Shared Everything DBMS memory architectures

Lars-Erik Bjørk and Truls Jørgensen

PREFACE

This report is the result of our diploma in Database Technology and Distributed Systems pursued in the 10th semester of the masters programme in computer science at the Norwegian University of Science and Technology, NTNU.

The motivation for performing this research was given by professor in database science, Svein-Olaf Hvasshovd. He has been our teaching supervisor, providing invaluable insight to the problem scope. His support has been a great factor of success for this project.

When carrying out our diploma, we faced several obstacles. The larger of these was the need to learn the C programming language in a limited amount of time. Not only the core language, but also additional libraries were needed to realize different benchmarks. We would like to thank senior research scientist Øystein Torbjørnsen for guiding us when traveling in the world of C. His willingness to share of his coding expertise has indeed improved the quality of the code substantial parts of this report relies on.

The task of becoming familiar with the Solaris operating system has also been a challenge. We had not been able to perform the micro benchmarks at all without senior engineer Øystein Grøvlen at Sun Microsystems, Trondheim. He made the reservations for the highly requested Niagara machine, as well as helping us out with tips and tricks in Solaris and promptly answering our sometimes somewhat naive questions.

We would also like to thank professor Lasse Natvig and Ph.D. candidate Marius Grannæs for providing insight to the scope of memory access time.

Trondheim, June 8, 2006

Lars-Erik Bjørk Truls Jørgensen

Abstract

This report carries out a relative performance comparison between two DBMS architectures on the Multi Core, Single Die (MCSD) realization Niagara. The two DBMS architectures in question are Shared Nothing (SN) and Shared Everything (SE). The MCSD field is rapidly evolving, and we expect that this technology will become increasingly important in the near future.

In order to carry out the comparison, the performance of the architectures must be calculated. This calculation depends on the cost figures associated with each architectural approach. To identify these costs, we present the design solutions made and results discovered in our previous work. Based on this, the most significant costs are determined and scheduled to be micro benchmarked.

The natural next step is to examine possible techniques to implement the benchmarks. In order to do this, we first expand on the Niagara chip and the platform on which the micro benchmarks will run. Having a sufficient theoretical platform to continue, we move on to describe the implementation of each micro benchmark in detail.

After benchmarking all the most significant costs, we thoroughly discuss the results, some of which are indeed surprising. The costs which are not benchmarked are based on assumptions from our previous work and recalculated to apply to Niagara.

For both SN and SE, we evaluate the system for two classes of transactions. The first class is transactions touching one tuple (called simple), the second is transactions touching four tuples (called complex). Each class has two instances, read and update. In order to perform the subsequent analysis, the decomposition of each transaction is presented in detail.

When analyzing the outcome of the calculations, interesting results emerge. First, we note that SE is the cheapest alternative when evaluating the simple transactions. This is because the SN approach includes an administrative overhead component that does not pay off when the transaction only touches one tuple. However, for complex transactions, the overhead component results in a parallel gain for SN which outperforms SE.

Based on the most dominant costs of both architectures, we perform a sensitivity analysis. For SN, the analysis is based on the cost for message passing. For SE, it is based on the cost for synchronization. The goal of this analysis is two folded. First, it is interesting to see how the results vary. For example, what the ratio between the cost for message passing and the cost for synchronization must be in order to make the two approaches perform equally well. Second, the analysis indicate how error-prone each architecture is to erroneous estimation.

The sensitivity analysis examine the performance of SN and SE when the ratio between the cost for message passing and the cost for synchronization is varied. This is done in both the read and the update cases. In addition to examining the simple and the complex transactions, we examine general transactions were the number of operations are not predetermined.

The analysis of the general read transaction suggests that when the number of operations increases, the message passing and synchronization costs wipe out the impact of the other costs. It also suggests that when the cost of message passing is greater than 4 times the cost of synchronization, SE performs better when increasing the number of read operations. Sim-

ilarly, if message passing is cheaper than 4 times the cost of synchronizing, SN is preferable. When increasing the number of update operations, the ratio is 3.33.

After concluding the analysis, we suggest a hybrid architecture that might combine the advantages of SN and SE. At the cost of introducing both message passing and synchronization, the architecture introduce parallelism in SE.

Lastly, we identify suggestions for future work. Realized and applied to the DBMS model introduced in this report, we believe that several of these suggestions can shrink some of the costs presented.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

The goal of this report is to perform a relative performance comparison between the two memory architectures Shared Nothing (SN)[1] and Shared Everything (SE) on Niagara. Sun Microsystems terms the Niagara processor "Radical CMT". In order to make the difference between ordinary Chip Level Multi Threading (CMT) processors and Niagara clearer, we have chosen to term the Niagara chip Multi Core Single Die (MCSD).

The research presented in this report is based on our earlier research presented in [BJ05]. This research was considered an early step on the road towards a consensus about the preferred Database Management System (DBMS) memory architecture on an MCSD chip. Because the research only sought to identify trends, the cost approximations used needed only be good enough to capture the relative merits of the different architectural approaches. The research therefore investigated a generic hardware architecture, consistently assuming minimum costs for all calculations.

This report, on the other hand, is the second step. Based on the DBMS design solutions suggested in [BJ05], we benchmark the most important cost figures used in the calculations and perform a second comparison between the two architectures.

This calculation is based on Sun Microsystems' Niagara. Thus, although some of the assumptions made in [BJ05] are still valid, this research has a real world approach. Even though this report bases its calculations on either SN or SE, it is possible that an architecture combining the advantages of both could yield even better results. We therefore briefly propose such a hybrid architecture. Validating this architecture is left for further work.

In the following we first state our motivation for the report. We then propose a route ahead.

1.1 MOTIVATION

The volume of information produced over the last decade has grown exponentially. In fact, as stated by [Swe01], the global storage has increased from 90 TB in 1983 to 160 000 TB in 1996 and to almost 3 000 000 TB in 2000. This growth continues.

In the past decades, the trend in processor development has closely followed the predictions of Gordon Moore [Moo65]. Moore's law originally states that the number of transistors on a chip is doubled every 12 months. In an update article from 1975, Moore adjusted the rate to every 24 months to account for the growing complexity of chips, [Moo75]. As the years went by, Moore's law evolved to be interpreted as the doubling of microprocessor power every 18 months [Tuo02].

However, according to [Sun03], the gain in overall system performance is much poorer due to the lower increases in memory speeds. Memory speeds double approximately every

[1]All abbreviations used in this report, such as these, are explained in this manner at first use. For reference, the abbreviations are also listed in appendix A.

six years. This is illustrated in figure 1.1. Due to this widening gap, processors are idle up to 75% of the time while waiting for memory fetches.

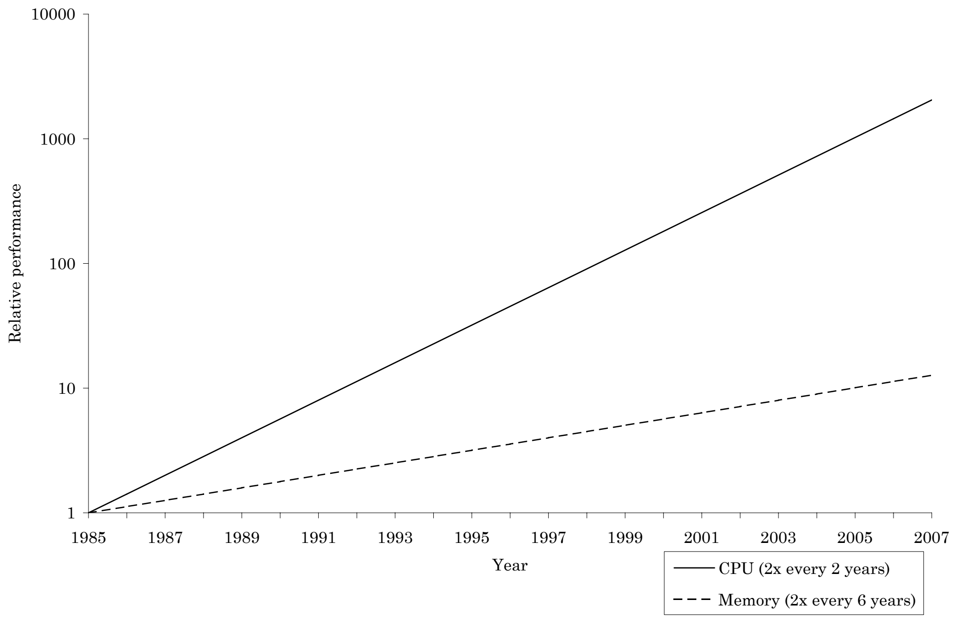


Figure 1.1: The memory bottleneck. The CPU power doubles every 2 years, whereas memory speed only doubles every 6 years. This leads to a widening gap.

This calls for new ways of thinking about data processing. We also see several new computer science fields gaining high attention, such as data mining and data distribution. These fields have all in common the need for raw processor power.

1.2 The Route Ahead

Our previous report closely followed the method of work used in [HT93], which presented a high availability DBMS architecture and a set of benchmark transactions. These transactions were decomposed into sub tasks, and the cost associated with each task was calculated. Although this report mainly follows the same route as our previous work, it differs in content, because we are now basing our most significant cost figures on micro benchmarks rather than theoretical minimal costs. Therefore, the report structure deviates from the previous until we present the cost figures in chapter 6. The cost figures which are not benchmarked are based on the assumptions made in [HT93].

1.2.1 Chapters

This section briefly presents all the following chapters in this report.

Chapter 2 Chapter 2 presents the most important theory and findings from our previous report. This chapter is important as the design solutions presented still are valid. The chapter sums up with identifying the most significant sub tasks that are subject to be benchmarked

Chapter 3 In order to perform the micro benchmarks, a pre study is necessary. Chapter 3 presents a discussion on alternative strategies for performing the micro benchmarks, as well as related Niagara research.

Chapter 4 Chapter 4 describes how the micro benchmarks are approached. The chapter describes how time keeping is performed and how the benchmarks are scheduled.

Chapter 5 Chapter 5 presents the results obtained by the benchmarks described in the previous chapter.

Chapter 6 Having presented all results from the benchmarks, chapter 6 presents the typical cost figure for each benchmark. The chapter also includes a theoretical calculation of the cost figures for the tasks that are not benchmarked.

Chapter 7 Following the structure from our previous report, chapter 7 presents a thorough cost calculation of four different transaction types with increasing complexity in an SN context [2] .

Chapter 8 Following the structure from our previous report, chapter 8 presents a thorough cost calculation of four different transaction types with increasing complexity in an SE context [2] .

Chapter 9 Chapter 9 presents an analysis based on the calculations presented in the two previous chapters. This analysis includes a summary of the results, a comparison and a sensitivity analysis.

Chapter 10 Based on the findings made in our report, chapter 10 suggests a possible hybrid architecture. This is done in an attempt to combine the advantages of both SN and SE.

Chapter 11 Finally, chapter 11 points in the directions of interesting fields of further research.

1.2.2 Appendices

This section briefly presents all appendices in this report.

Appendix A Presents the abbreviations commonly used in this report.

Appendix B Presents the shell scripts used to schedule the benchmarks

Appendix C Lists all source code for the micro benchmarks

Appendix D Presents all result data returned from the micro benchmarks

1.2.3 Type conventions

Table 1.1 describes the type conventions used throughout the report. The following lists all conventions, and their meanings.

[2]Regarding chapter 7 and 8, it is worth noting that these calculations are indeed very similar in structure to the calculations presented in [BJ05]. However, because all the cost figures are different, these calculations must be performed nonetheless. It is also necessary to be familiar with the transactions' decompositions presented to make the most of the analysis presented in chapter 9.

Typeface	**Usage**
Italic	• Processes • References to names used in figures • References to other sections by name
Boldface	• Table headings • Table captions • Figure captions
`Typewriter`	• References to file names • References to code • References to shell scripts • Command prompt
`Boldface typewriter`	• Terminal commands

Table 1.1: Type conventions.

CHAPTER 2 FOUNDATION

This chapter presents a brief summary of the previous work, where we performed a relative performance comparison between the two memory architectures SN and SE.

First, the method of work used to approach the problem is presented. Thereafter we present the transaction classes used in this report. Before moving on to the design solutions, it is necessary to present the assumed DBMS kernel this report relies on.

The following sections describe design solutions made and results discovered for both SN and SE. These solutions are valid also for this report. After a brief comparison between the two architectures and a sensitivity analysis varying the most interesting costs, the chapter concludes with identifying the most significant costs which form the basis for the micro benchmarking performed later.

As stated in the introduction, this report is a continuation of our previous work [BJ05]. The interested reader will find the content in this chapter to have clear similarities with our previous work. Indeed, except from the last section, this chapter is included for completeness.

2.1 METHOD OF WORK

Because our previous report was an early step on the road towards a consensus about the preferred DBMS architecture on an MCSD chip, there was not much former work on which the report could be based. This section presents the problem addressed and the method of work used.

For two reasons it was not possible to perform a simulation of the SN and SE architectural approaches. First, an MCSD chip was not to our disposal. Second, both time and personnel were limited resources. Several major assumptions were therefore made on which the research rested. The main assumptions were:

- Although utilizing most of the properties of the Niagara chip, the processor examined was not a specific realization of the MCSD architecture, but merely a generic MCSD chip. For example, the generic chip run at 3 times the clock speed of Niagara executing twice as many instructions per cycle.
- The costs presented were the absolute minimum theoretical costs, assuming full Operating System (OS) support for the different tasks and no real-world overhead such as house keeping and stack delays.
- It was assumed possible to write the log to two independent parts of the memory with independent failure modes. This would be done in parallel, facilitated by hardware.

In addition, none of the costs used in calculations were based on real-world benchmarks on an MCSD chip. The costs were based on earlier work ([HT93]) and on calculations made

ID	Abbreviation for	Description
STU:	Simple Transaction Update	Update a single tuple.
STR:	Simple Transaction Read	Read a single tuple.
CTU:	Complex Transaction Update	Update four tuples.
CTR:	Complex Transaction Read	Read four tuples.

Table 2.1: The four transactions used

possible by design choices stated in the report. These costs were a great factor of uncertainty when the report identified trends of the relative merits of the architectural approaches.

[BJ05] started with describing both the SN and the SE memory architectures in general. Thereafter, both SN and SE were mapped to an MCSD context.

The rest of this chapter summarizes and compares the costs and findings for each memory architecture, before the final section states the transaction sub tasks that are considered to have the greatest impact on the relative performance of the SN and SE architectural approaches. First, however, we define the transaction classes used in this report.

2.2 Transaction classes

The transactions assumed when calculating the performance of SN and SE are presented in table 2.1. These transactions were chosen because they illustrate different behaviors of the DBMS. The two simple transactions, STU and STR, are used to estimate the workload for the simplest possible case. The only operation performed by STR is a simple read of one tuple. Likewise, STU updates one tuple. The two larger transactions, CTU and CTR, are considered to be of sufficient complexity to approximate the workload presented by an average transaction in a DBMS. The CTR transaction reads four tuples, and the CTU transaction updates four tuples. In SN, the complex transactions show the DBMS behavior when more than one node is involved in a transaction. In SE, all transactions show how the responsible node behaves in order to ensure that the transaction execution does not interfere with other transactions run by other nodes.

Each of these transactions were decomposed into to simple sub tasks such as creating a transaction context, setting a lock, setting a mutex, building a log post, writing to log, sending a message etc. An estimate was made for the cost associated with each sub task. Based on these estimates and the DBMS architectures presented earlier in the report, the costs associated with executing the transactions were calculated for each architecture. A similar method of work is used in [HT93].

The reason for not choosing transactions following a benchmark standard i.e. TPC-B (as defined in [Gra91]), is that it would be too big a task within the scope of this research project. Albeit mentioned in our previous work, it is necessary to justify their use, because they will indeed be important with regards to the evaluation presented later.

Before we continue, it is necessary to define a transaction's critical path. This is described as the sum of all operations needed to be performed before a response can be given back to the transaction service requester.

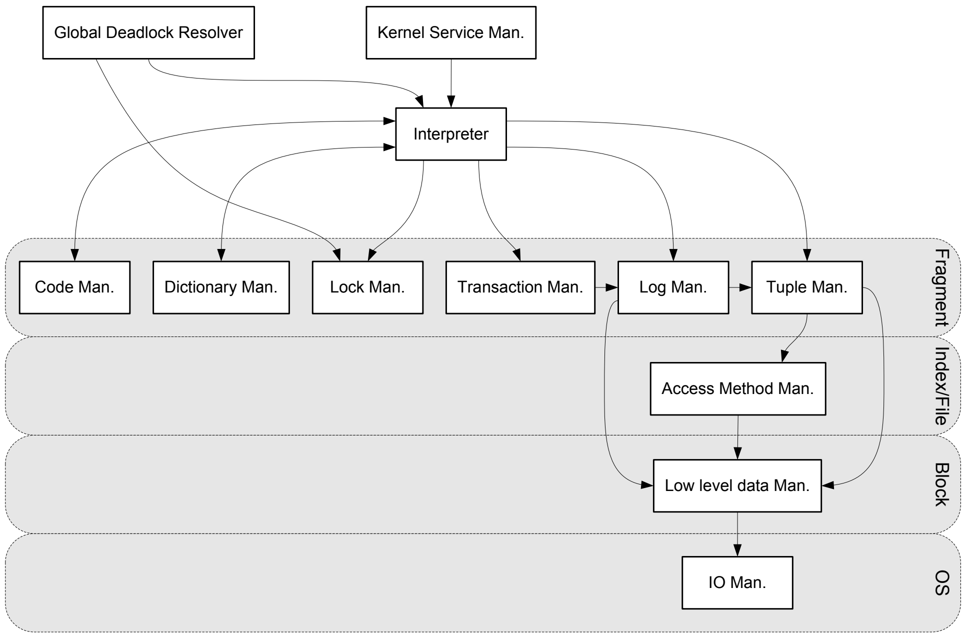


Figure 2.1: The DBMS kernel processes. The Code Manager contains ready loaded code to be executed. The Dictionary Manager provides content information. The Lock Manager supports locking of tuples. The Transaction Manager maintains the state of all active transactions of the node. The Log Manager provides the ability to log. The Tuple Manager supports the normal set of tuple operations (Read, Write, Update, Delete). The Log Manager and Tuple Manager use in turn services provided by the Access Method Manager. The Access Method Manager provides support for file access methods, and relies in turn to lower level data management.

2.3 THE DBMS KERNEL

Our design for the DBMS Kernel is identical to that of our previous work. It is heavily influenced by [HT93]. This section describes the DBMS Kernel processes illustrated in figure 2.1. First, the *Kernel Service Manager* is responsible for access control. The Kernel Service Manager inserts operations in the execution queue, which is handled by the *Interpreter*.

Every operation is run to completion, unless it is running into a deadlock. Then the execution halts until it is discovered by the global deadlock resolver, which reinserts the operation in the execution queue.

The Interpreter executes one operation at a time from the execution queue. The Interpreter depends on services from the *Fragment* layer. Executing in a typically from left-to-right manner, the first fragment, *Code Manager* manages the ready compiled, loaded code which typically is the result of an SQL-optimizer. The *Dictionary manager* keeps track of where the tuples are stored (and, for the SN case, partition information). It then utilizes the *Lock Manager*, which supports locking of tuples.The *Transaction Manager*, maintains the state of all active transactions of the node. The Transaction Manager utilize the services from another fragment, namely the *Log Manager*. The Log Manager is in turn depending on the *Tuple Manager*. The Log Manager and Tuple Manager both depend on lower layer components, such as the *Access Method Manager*.

The Log Manager and Tuple Manager use in turn services provided by the Access Method Manager. The Access Method Manager provides support for file access methods, and relies in turn to lower level data management.

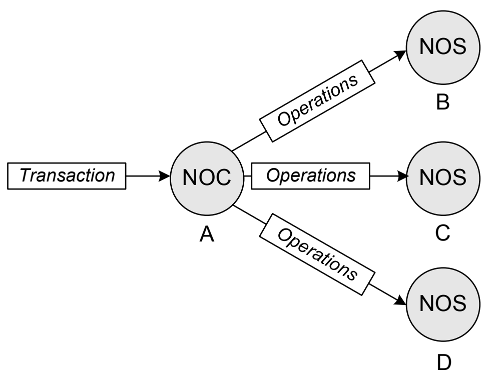


Figure 2.2: Shared Nothing on MSCD in a nutshell. Node *A*, which receives a transaction, forwards operation messages to all participating nodes.

2.4 THE SN DESIGN SOLUTION

This section first briefly explains the concept of SN in a general MCSD context, before explaining major design decisions for the SN architectural approach.

2.4.1 SN on MCSD in brief

In an MCSD context, each logical processor can be seen as a node. Distributed transactions frequently occur in SN. In a distributed setting, a transaction needs to perform operations on tuples that reside in different subsets of the database. Each node has exclusive access to its respective subset. Figure 2.2 illustrates SN behavior on a MCSD in a nutshell. If a transaction is executed at node *A* and needs to update tuples owned by the nodes *B*,*C* and *D*, node *A* must send a message to these nodes containing the operations to be executed. Nodes *B* - *D* will then, acting as Node Slave (NOS) for the Node Controller (NOC) *A*, execute these operations in accordance to a First-Come-First-Serve policy[1](see section 2.4.5) at each node.

In such a distributed setting, each node is designed to have a transaction controller and a DBMS kernel. Each node has private access to a distinct portion of the database, as illustrated in figure 2.3. The transaction controller acts as a coordinator for the node, and is responsible for initiating operations and coordinating the distributed commit. The DBMS kernel is responsible for performing operations on the database. As illustrated in the figure, there is direct communication between the controller and the kernel on a different node. This is done in order to avoid the process switch imposed by strict inter-controller communication.

For a more general explanation of the concept of SN, see [DG92]. For a more detailed explanation of SN in a MCSD context, see [BJ05].

2.4.2 The 2PC protocol

As mentioned above, a transaction is often distributed in SN. That is, several nodes can be involved in a transaction. Clearly, one needs a way to ensure that all participating nodes in such a transaction know whether the transaction eventually committed or not. The Two-phase commit (2PC) protocol is used for this. Mohan et al. describes a variant of this protocol

[1]We assume no queuing of operations on the NOSes for simplicity.

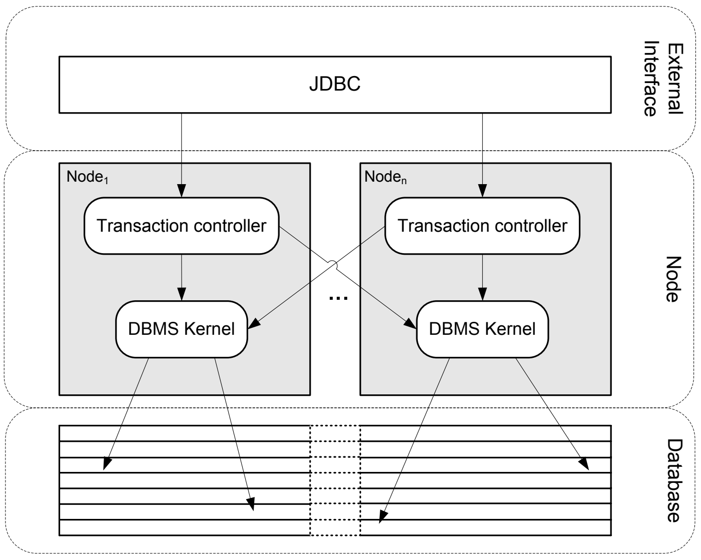


Figure 2.3: The Shared Nothing process architecture. The node has private access to a distinct portion of the database. The transaction controller acts as a coordinator for the node, and is responsible for initiating operations and coordinating the distributed commit. The initiating node, which receives a transaction from the external interface, is called a Node Controller (NOC). The NOC's transaction controller communicates directly with the DBMS kernel of the performing node, called Node Slave (NOS). The DBMS kernel performs operations on the database.

called 2PC *presumed commit* in [MLO86]. Figure 2.4 illustrates this latter scheme. For a more thorough explanation, consult [BJ05].

2.4.3 Synchronization

The SN architecture does not need heavy synchronization, because each node has a distinct, private part of the memory on which it operates. This principle does not only apply to the database content, but it also implies that all database data structures are maintained within each node. Thus no synchronization on the database structures is needed.

2.4.4 Locking

Transactions often need to perform operations on tuples that reside in subsets owned by other nodes. Because operations passed to a DBMS kernel (see figure 2.3) of another node could be sent in several batches, maintaining transaction lock mechanisms is crucial. The Two-phase locking (2PL) mechanism, thoroughly explained in [Tan01], ensures that the interleaved execution of the transactions is serializable. In brief, 2PL works in the following manner: There are two phases, one for locking and one for releasing. During the first phase, a process is only allowed to lock needed records. As soon as the process releases a lock, it enters the second phase which consists of releasing locks. Interleaving the locking and releasing of resources is not allowed.

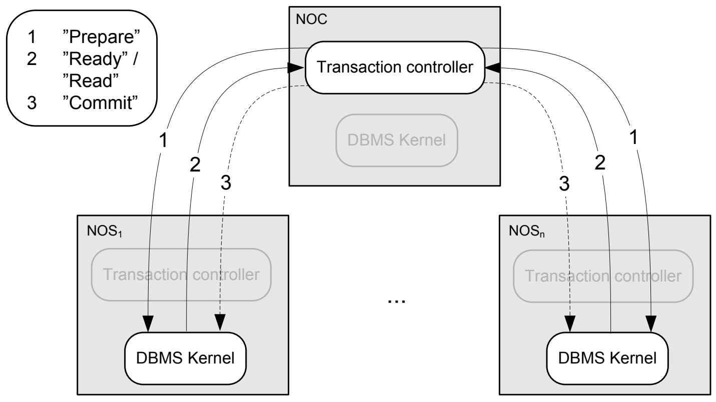


Figure 2.4: The SN communication between nodes - the 2PC presumed commit. Message *1*, '*prepare*', is sent from the NOC to all participating NOSes. The '*prepare*' message is piggybacked on the operation message. For simplicity, only two NOSes are shown. The content of message *2* depends on whether the operations to be performed are pure reads or involve updates. For pure reads, a '*read*' is returned. This message is piggybacked on the message containing the read data. If one or several write operations are involved, a '*ready*'message is sent back to the NOC when the NOS has performed the operations in question. Message *3*, '*commit*', is sent back to all participants that sent a '*ready*' in message *2*. The NOC then returns the transaction outcome to the external interface (as seen in figure 2.3).

2.4.5 Scheduling

The DBMS maintains a straight forward First-Come-First-Serve scheduling policy at each node. Requests to perform operations on tuples, originating from the owning node or other nodes, are executed in turn[2]. Thus, it is not performed any optimization of the operation execution on the node level.

2.4.6 Background activities

Several background activities are performed in the DBMS. These activities can be an important factor with regards to system performance on a real system. However, as these activities are outside the transaction scope, they are not taken into account when measuring the relative difference between SN and SE.

In the SN case, the identified background activities are:

DB buffer flushing Because this is a main-memory database, the choice of buffer flushing strategy is different from that of disk-based databases. Using a regular Least Recently Used (LRU) strategy could have the unfortunate side effect of leaving dirty blocks in main memory for a long time before eventually flushing them to disk. Therefore, we use the strategy presented in [HT93], where dirty pages are flushed to disk in fixed intervals, typically every 10 seconds.

Checkpoint logging The background activity of logging dirty pages to disk follows the same strategy as the one for buffer flushing.

[2]For this to scheme to be effective, there can be no waiting for the disks in the critical path.

Transaction	**NOC** (ns)	**NOS** (ns)	**Sum** (ns)
STU:	741	1957	2698
STR:	441	1558	1999
CTU:	1686	1957	3643
CTR:	1086	1558	2644

Table 2.2: Cost summary for SN

2.5 SN RESULTS

This section presents the results from the calculations made by [BJ05] regarding the SN case. The cost figures the calculations are based on are for brevity not included in this summary. The times needed to execute the different transactions are presented in table 2.2

The NOC is run in interleaved execution with the NOSes. The benefit of this approach compared to SE is the speedup gained by the parallel execution of the NOSes involved in a transaction execution. Assumed that the executing transaction only performs operations on tuples that reside at other nodes, the NOC is pure overhead. Section 2.8 presents a comparison discussing the differences between SN and SE.

2.6 THE SE DESIGN SOLUTION

This section first briefly explains the concept of SE in a general MCSD context, before explaining major design decisions for SE.

2.6.1 SE on MCSD in brief

In SE, each node has access to the entire database. Each node has a DBMS Kernel that is responsible of performing database operations. In the SN case the transaction controller is used to coordinate distributed transactions (see figure 2.4). Because no such transactions occur in the SE case, there is no need for a controller as illustrated in figure 2.5. However, because all nodes have access to the same memory, a tight synchronization scheme is necessary. In the following subsection, we briefly explain how thread synchronization is carried out for the different data structures.

2.6.2 Synchronization

The typical behavior of a transaction operation can be decomposed into the following:

- Access the dictionary
- Set a lock on some data resources
- Perform some low level block operations
- Create log records

As an example, consider the STU transaction, which updates a single record. This transaction starts with doing a dictionary operation. In order to do this, the thread executing the

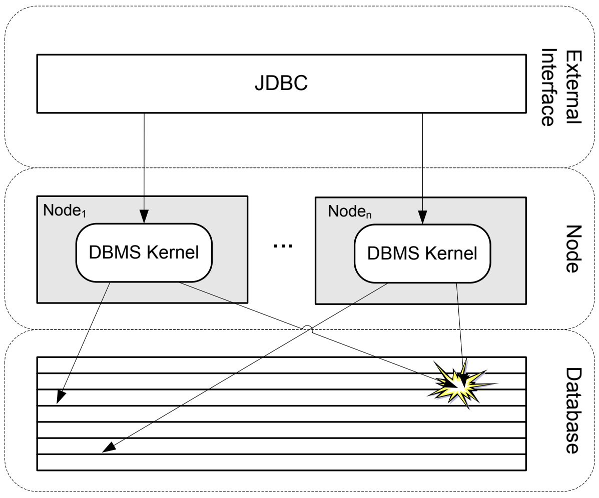


Figure 2.5: The Shared Everything process architecture. Each node has access to the whole database, and there are no distributed transactions. However, different nodes can interfere with each other, as illustrated with the explosion mark to the right. In this case, a synchronization scheme is needed to ensure serializable execution. We briefly explain how thread synchronization is carried out for the different data structures in the following subsections.

transaction must be sure that no other threads are writing to the dictionary at the same time. Thus, there is a need to synchronize the use of the dictionary. The next step is to perform an update operation on a record. According to the 2PL protocol (see section 2.4.4 and [Tan01]), the transaction must first hold an exclusive lock on the respective record. In order to achieve this, the transaction must add an entry to the lock hierarchy. To ensure that no other thread sets an exclusive lock simultaneously, the use of the lock hierarchy must be synchronized. To ensure that no other thread changes the b-tree at the same time as the thread updates the record, the use of the b-tree must also be synchronized. After the record operation is completed, the change must be written to log. To ensure that two threads do not simultaneously write to the same log record, the use of the log needs to be synchronized. Finally, the transaction must release the lock, which yields yet another synchronization of the lock hierarchy.

The data structures that must be synchronized were identified to be

- Lock hierarchy
- Dictionary
- B-tree (Managing data blocks)
- Log

The following subsections briefly describe the data structures and the synchronization schemes used for each structure. Note that although mutexes was used as the primary synchronization mechanism in [BJ05], the synchronization mechanism for this report is not to be decided until chapter 4. Severable possible solutions were discussed on how to best make use of the mutexes. The interested reader is referred to the complete report for a full argument to why the different approaches were taken.

Lock hierarchy

Locks are organized in a lock hash table. Hashing is performed on the data resource's primary key. Resources hashing to the same hash bucket are connected using a linked list. The transaction IDs of transactions holding a lock on the same resource are connected using a linked list associated with the respective resource. In order to synchronize the lock hierarchy, a mutex is set for the entire lock hash table for every single instruction. In other words, there is only one level of synchronization. See [Tan01] for a detailed explanation of mutexes.

Dictionary

In this context, the dictionary is assumed to be a table residing in main memory that defines the basic organization of the database. In the SE case, the dictionary will mostly be used to locate files and may therefore also be called a file dictionary. The synchronization scheme used to synchronize the dictionary, is indeed similar to that of the lock hierarchy. The only difference is that because reading from the dictionary is far more common than writing to it, database locks are used instead of mutexes, so that several read-locks may coexist. A lock is therefore associated with the entire dictionary, ensuring that only a single thread may access the dictionary at any time if its intention is to write. However, if the intention is to read, several threads may read from the dictionary simultaneously.

B-tree

A major assumption made in our previous work was that the whole b-tree resides in main memory. With the goal of minimizing the use of mutexes in the b-tree, the approach to hold a single mutex for the entire b-tree, during an update or read, is chosen.

Log

The log is organized in a circular buffer. This means that the buffer size is fixed, and when the buffer is full, the oldest part of the log is overwritten. In order to ensure durability and atomicity, log records must be written to stable storage before this happens. It is assumed that it is possible to write the log to two independent parts of the memory with independent failure modes. As suggested in [HTBH95], this should be sufficient to ensure durability. Facilitated by hardware, it is also assumed that this can be done in parallel, thus implying that the cost of logging can be estimated simply as writing to main memory. The synchronization scheme presented is to write the log post to memory while holding a mutex for the entire log buffer.

2.6.3 Background activities

As explained in section 2.4.6, some background activities are performed in the DBMS. These activities can be an important factor with regards to system performance on a real system. However, as these activities are outside the transaction scope, they are not taken into account when measuring the relative difference between SN and SE. In the SE case, the identified background activities are:

Cache invalidation Although cache invalidation is not a part of the critical path of the transaction execution, it is an important background activity that must be taken into consideration when evaluating SE. The reason, as we described at page 25 in [BJ05], is the cache update challenge SE must face. One update to a tuple can require cache invalidation for all other cores holding a local copy of that tuple in their private cache. The application programmer has limited control over such cache invalidation, as this task indeed resides in the OS domain. As a frequent background activity, this will consume some of the available resources, adding to the total costs of the SE approach.

DB buffer flushing and Checkpoint logging This background activity is described for the SN case in 2.4.6. These costs also apply for SE.

2.7 SE RESULTS

This section briefly presents the results from the calculations made in [BJ05] regarding the SE case. As for the section presenting the design solutions in the SN case, the summary given here is brief and the interested reader is referred to the complete report for a thorough presentation.

The time needed to execute the different types of transactions is presented in table 2.3.

Transaction	**Cost** (ns)
STU:	2141
STR:	1845
CTU:	7346
CTR:	6570

Table 2.3: Cost summary for SE

In SE, all operations of a transaction are performed in a serial execution. Opposed to SN, there is no parallelism[3].

As figure 2.6 suggests, there is a linear increase in time when increasing the number of operations. The curves for the read and update transactions are plotted using the ratio between the simple and complex transactions, $\frac{CTR}{STR}$ and $\frac{CTU}{STU}$, respectively. The curves are for comparison plotted against curves derived from the costs of performing a single update or read. The gain in SE is caused by the tasks that need not be run for each operation.

2.8 COMPARISON

Figure 2.7 shows a comparison between SN and SE for transactions containing up to four operations. In SN it is assumed a 1:1 relationship between the number of operations and the number of NOSes until 32 operations are reached. The solid lines illustrate the two architectures in the update case, the dashed lines illustrate the read case. As illustrated by the figure, SN is more expensive than SE when the transaction only has one operation. The reason is that the transaction is not actually distributed. For one operation, the NOC is pure overhead. In the SE case, on the other hand, there is no such controller and hence no overhead.

However, as the number of operations increases, the parallel gain of the SN approach emerges. In SE, all operations are executed at the same node. Thus, the execution time increases linearly with the number of operations. For SN, however, different operations will be executed at different nodes. An assumption made in [BJ05] is that different operations are

[3]As previously mentioned, in SN there is only a small increase in time when executing an additional operation. This increase is due to the work added to the transaction controller. The additional operation, however, is performed in parallel with existing operations. This is not the case with SE.

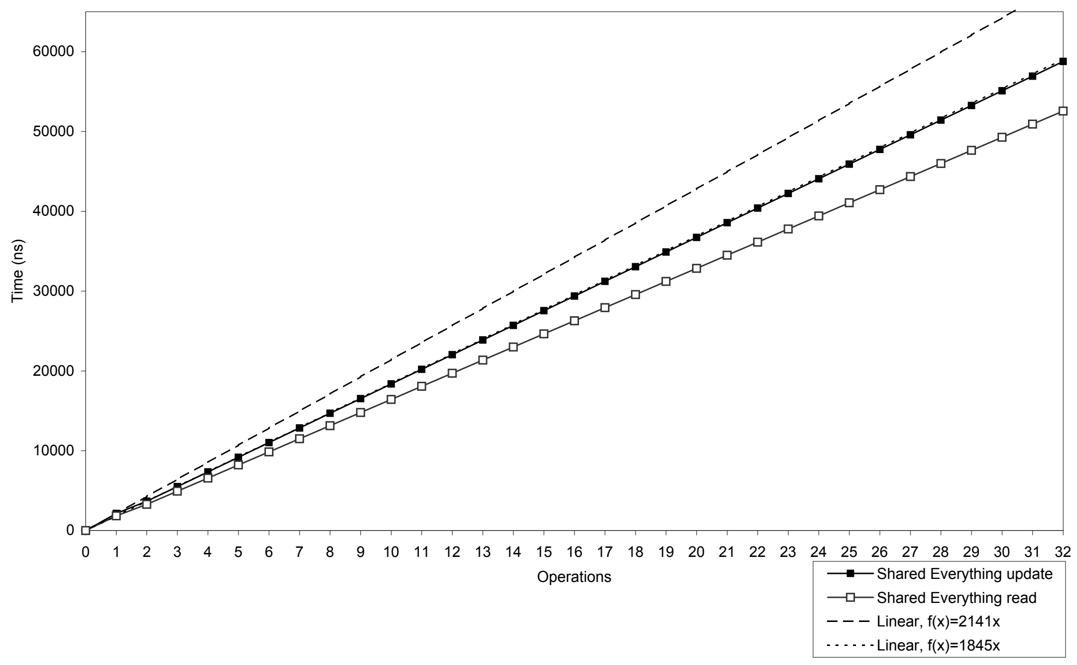


Figure 2.6: Cost figures for SE. The cost figures are almost linear, both for the update and read case.

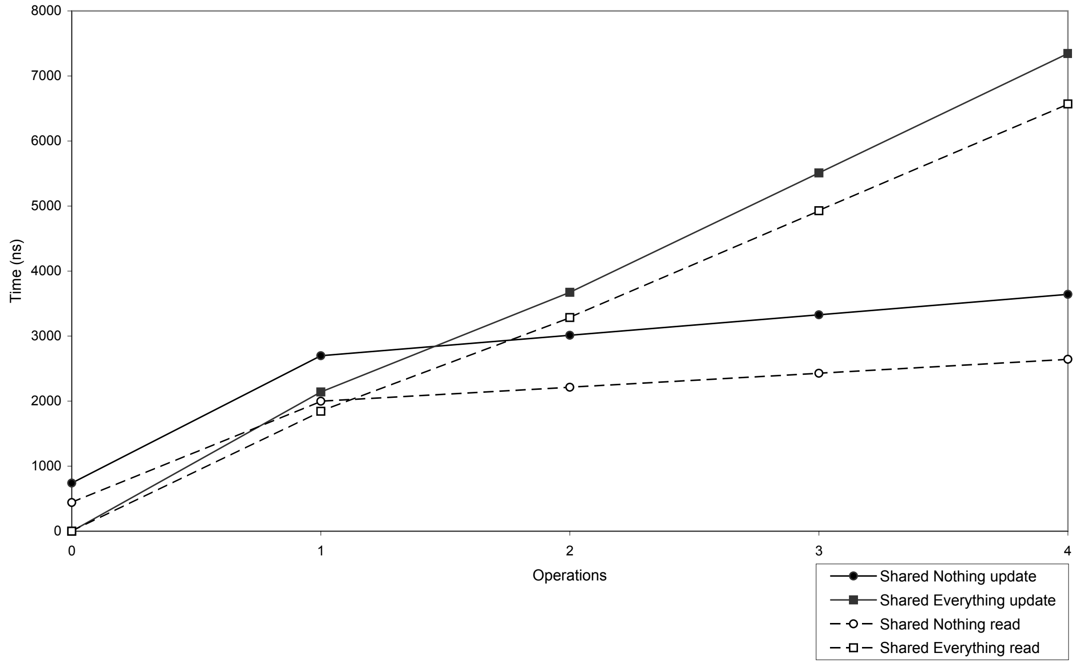


Figure 2.7: A 4-node comparison graph between SN and SE DBMS architecture.

solely sent to different nodes in SN. This implies that only the extra work done by the NOC will increase the time used to execute a transaction. The execution of the additional operations is done in parallel. This is valid as long as there is only one operation per node.

The same comparison is illustrated by figure 2.8, spanned over 32 nodes and with time measured in a logarithmic scale. Both approaches has a linear growth, but at 32 operations,

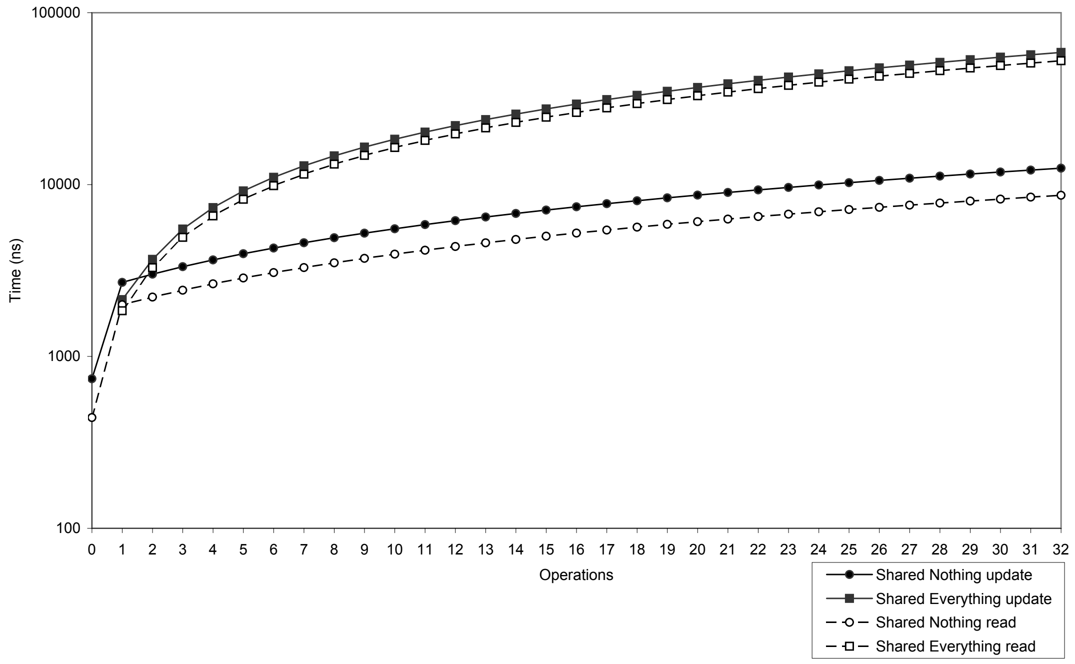


Figure 2.8: A comparison graph between SN and SE in logarithmic scale. For 32 operations, SN performs almost one order of magnitude better than SE.

SN performs almost one order of magnitude better than SE.

2.9 SENSITIVITY ANALYSIS

This section includes a brief summary of the sensitivity analysis presented in [BJ05]. The analysis is performed for two distinct costs, namely the costs for mutex setting and message passing. These costs were chosen because they were believed to be the two most significant isolated costs.

Two sensitivity analyses are performed. The first analysis is based on the STU transaction. This transaction is chosen because it is interesting to look at the case where SN does not yet gain any advantages from its ability to parallel the execution. The second analysis is based on the CTU transaction. This transaction is chosen to get an idea of how expensive message passing must be in order for SE to be the cheaper alternative when executing complex transactions.

In the original analysis, several parameters were varied for each approach. Because this is just a summary, only the most interesting variations are included.

2.9.1 Sensitivity analysis based on the STU transaction

This section gives a sensitivity analysis based on the STU transaction. In this analysis the cost for setting mutexes in SE is varied, while the cost for message passing in SN is held constant. This is illustrated by the graph in figure 2.9. There are three horizontal lines in the graph. These lines represent the cost of executing the STU transaction on SN when the cost of message passing is the original cost, 2 times the original cost and 3 times the original cost. One might see from the intersection between the curve representing SE, and the curve

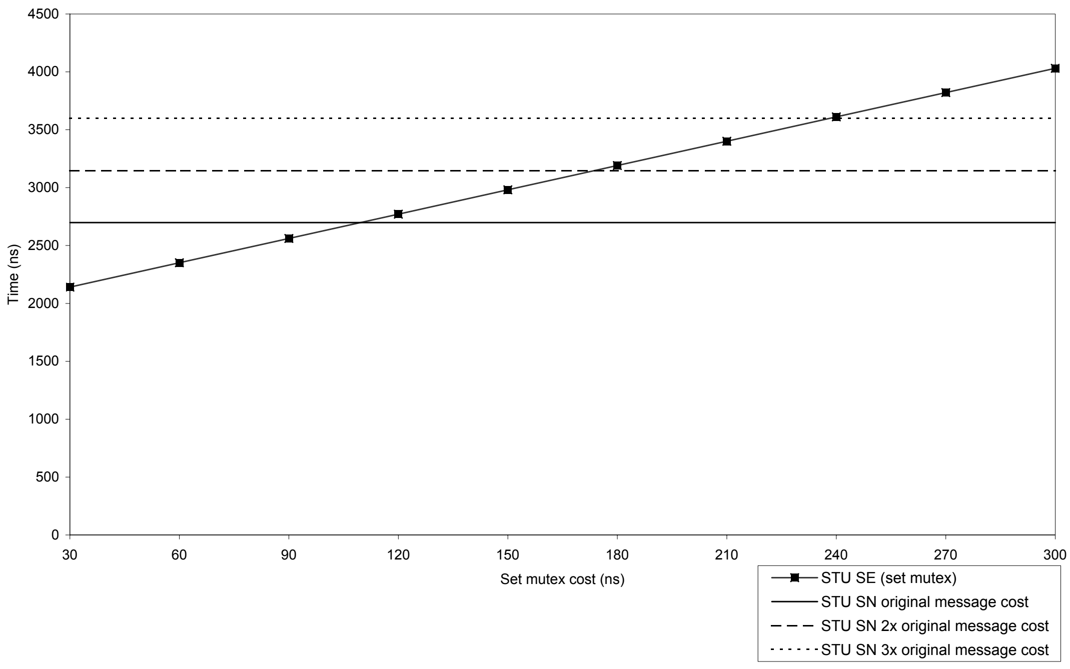


Figure 2.9: Varying the cost for setting mutexes when running the STU transaction. This graph shows one curve for the costs associated with SE when varying the costs for setting mutexes. Three other curves are shown for SN when using the original cost for sending and receiving messages, 2 times the original cost and 3 times the original cost.

representing SN when using original costs, that the original estimates for setting mutexes must be underestimated by a factor between 3 and 4 before SN yields better performance.

2.9.2 Sensitivity analysis based on the CTU transaction

When the CTU transaction is executed, SN is preferable due to its ability to parallel transactions. Is is however interesting to see how much the cost for message passing may increase before the parallel gain is consumed.

Figure 2.10 illustrates the scenario where the costs for sending and receiving messages are varied, while the costs for setting and releasing mutexes are held constant. There are three horizontal lines in the graph. These represent the cost of executing the STU transaction on SE when the cost of mutex setting is the original cost, 5 times the original cost and 10 times the original cost. The value of the x-axis at the intersection between the two solid curves, shows that in order to make the total costs of SN and SE equal, the cost for message passing must be multiplied by approximately 4. If the transaction to be executed involves an even higher number of operations, the cost for message passing has to be even higher before the two approaches perform equally good.

2.10 SIGNIFICANT COSTS

This section describes the costs considered to be most significant when executing a transaction. These are costs that either are exclusive to one architecture, or far more frequent in one

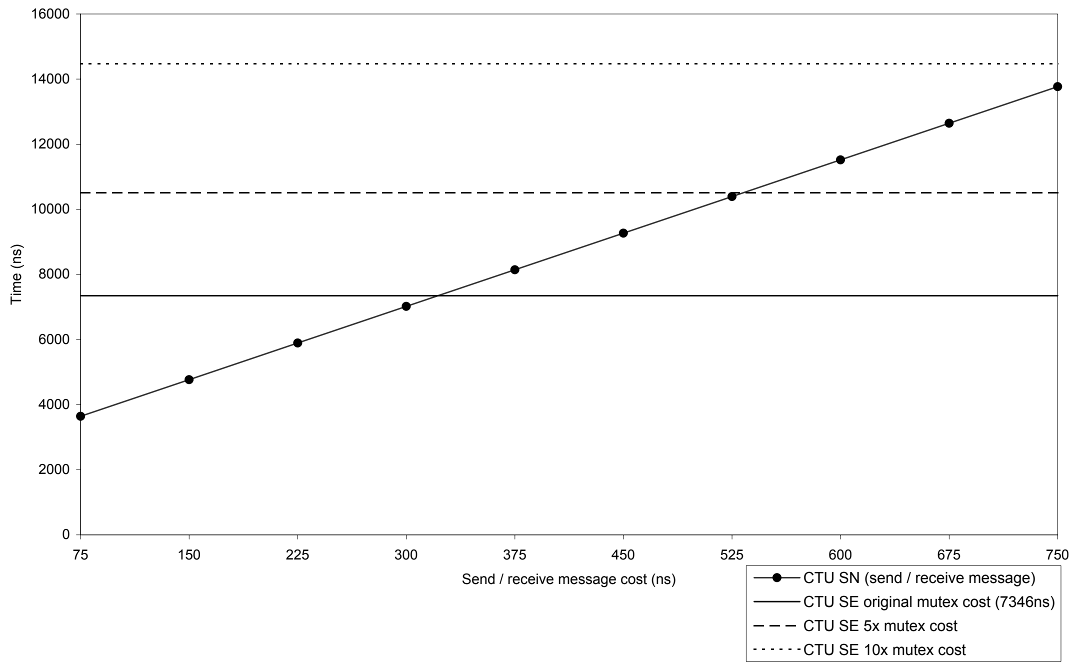


Figure 2.10: Varying the costs for sending and receiving messages when running the CTU transaction. This graph shows one curve for the costs associated with SN when varying the costs for sending and receiving messages. Three other curves are shown for SE when using the original cost for setting mutexes, 5 times the original cost and 10 times the original cost.

architecture than in the other.The following subsections describe each cost and the reason to why it is considered significant.

2.10.1 Sending and receiving messages

The task of sending and receiving messages is only performed in SN. In SN, all nodes have access to distinct portions of the database, and a transaction may require data from several locations, as explained in section 2.4. The SE approach however, implies that all nodes have access to the entire database. At the cost of heavy synchronizing, SE has no need for the message passing scheme that SN utilizes.

The message passing cost is considered to be significant for two reasons. First, our previous work suggested that this cost is dominant in SN. We proposed a theoretical minimal cost of 75ns for sending, and likewise, 75ns for receiving. We assumed complete and optimal OS support, which is necessarily not the case in practice. Second, because this cost is exclusive to SN, it is vital in order to establish the relative performance between the two architectural approaches.

2.10.2 Building messages and log posts

Because the task of writing to log is performed much more frequently in SN, due to the 2PC protocol, it is important to decide the cost associated with building log posts. Along the lines of the previous discussion (see section 2.10.1), this is also the case with the cost associated with building messages.

2.10.3 Interpreting messages and log posts

For each message or log post built, interpreting that message or post is also a task that needs to be taken into consideration. The same reasoning as in section 2.10.2 applies for this cost.

2.10.4 Writing to log

Both architectural approaches need to write to log when updating the database. However, because SN utilizes the 2PC protocol (see section 2.4), logging is performed much more frequently in SN than in SE. It is therefore important to determine the cost associated with writing to log.

2.10.5 Synchronizing shared data structures

As mentioned in 2.10.1, because all nodes in SE have access to the entire database, a tight synchronization scheme is needed in order to ensure serializable execution. This was further described in section 2.6. Although some internal process synchronization most likely is needed also in SN, such a tight synchronization scheme is obviously not necessary, as each node has exclusive access to a distinct portion of the database.

Because the synchronization of shared data structures only applies to SE, it is considered significant to establish the cost associated with such synchronization.

CHAPTER 3 PRE STUDY

Having established which significant costs to benchmark in section 2.10, the natural next step is to examine possible techniques to use when micro benchmarking these costs.

This chapter is however twofold. For our purposes, we covered the necessary topics regarding DBMS design choices for both SN and SE in chapter 2. Niagara is however yet to be covered. Although a thorough dissection of the Niagara chip is outside the scope of this report, there are some key concepts that must be examined before we continue. The first goal of this chapter is to expand on these concepts.

In section 3.2, the chapter continues with examining the platform on which the benchmarks will be executed. Section 3.3 looks into potential programming and scripting languages before we are turning our attention to possible programming techniques in section 3.4.

3.1 THE NIAGARA PROCESSOR

The MCSD realization Niagara has gathered much attention after the launch on December 6 2005. Sun Microsystems offers two server models featuring Niagara, the Sun Fire T1000 and T2000 server.

Although Niagara shares many of the properties with the generic MCSD chip examined in our previous work, there are two key concepts in Niagara that needs to be pointed out. After we have examined how thread switching and cache communication is carried out on Niagara, we refer to a related Niagara research.

3.1.1 Niagara thread switching

When a cache-miss occurs during execution of a thread, fetching data from memory may take several hundred cycles, leaving the processor idle. According to [Sun03] this may be as much as 75% of the time. When this happens, Niagara is immediately able to switch to another thread, reducing the impact of the cache-miss.

Figure 3.1 examines the masking of memory latency in one of the eight Niagara cores[1], directly addressing the problem with the widening gap mentioned in the introduction of this report (see figure 1.1).

If the mean number of cycles until a miss is at least $\frac{1}{3}$ of the number of cycles used to fetch data from memory, the latency may be completely masked. The relative time where a thread is run by the Central Processing Unit (CPU) is illustrated in the figure by the blocks named C.

[1]Each Niagara core utilize the Chip Level Multi Threading (CMT) property – The execution of instructions from multiple threads within one processor chip at the same time. Each core in Niagara utilizes this property. Sun Microsystems terms the Niagara chip “Radical CMT”. In order to make the difference between ordinary CMT processors and Niagara clearer, we have chosen to term the Niagara chip MCSD throughout the report.

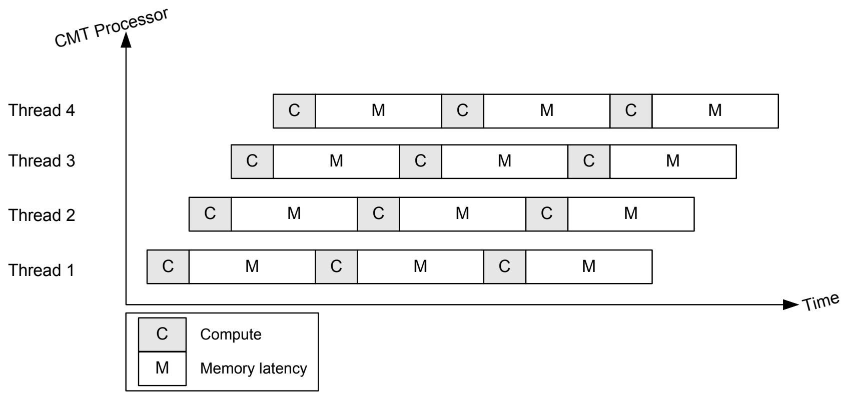


Figure 3.1: Eliminating latency. Running four threads within each core, the memory latency might be completely masked.

The relative time where the thread is waiting for a memory fetch is illustrated by the blocks named M. As seen from the figure, increasing the clock frequency of a processor will only reduce the size of the C-blocks, leaving the M-blocks, memory latency, unchanged.

This efficient thread switching scheme is indeed crucial for Niagara. As we shall see, the L1 caches are quite small. However, with multiple threads available, L1 misses become less critical, as threading can hide the L2 cache latency.

3.1.2 Niagara cache communication

Communication between processors in an ordinary multiprocessor system needs to go across expensive integrated circuit interfaces. In the case of current systems based on the SE architecture, the communication between processors is realized through main memory. In the case of the SN architecture, it is realized through message passing over an interconnection network. These ideas were described in more detail in [BJ05].

Neither of these approaches are necessary using the Niagara MCSD processor. It is in theory possible to perform all communication between threads by dumping information directly to L2 cache, as illustrated in figure 3.2. In the figure, C_1 to C_8 are the eight cores on the Niagara

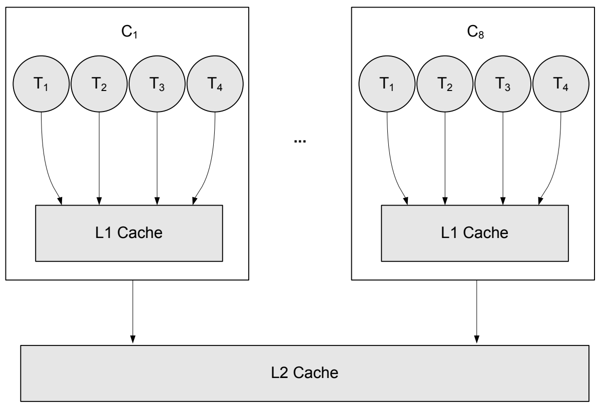


Figure 3.2: Cache architecture. Each core can run 4 concurrent threads. Each core has private access to L1 cache, whereas L2 cache is used for communication between the cores.

Feature	Sun Fire T1000	Sun Fire T2000
Form Factor:	1RU, 18.75" deep	2RU, 24" deep
CPU:	UltraSPARC T1 1.0 GHz	UltraSPARC T1 1.2GHz (4, 6, & 8 core)
Memory:	DDR2, 512MB/1GB/2GB 8 x memory slots, 16GB total	DDR2, 512MB/1GB/2GB 16 x memory slots, 32GB total
Network (10/100/1000):	4x	4x
Internal Storage:	1 x 3.5" SATA, non-hot-swap	Up to 4 x 2.5" SAS, hot-swap
Removable Media:	None	1x DVD-ROM
Serial:	1x RS-232, No USB	1x RS-232, 4x USB
PCI Express slots:	1x (low profile)	3x (low profile)
PCI-X slots:	None	2x (low profile, 1 occupied)
Redundant Power Supplies & Fans:	No	Yes
Power Supply:	1x300W	2x550W

Table 3.1: Sun Fire T1000 vs T2000. Gathered from Sun [Sun06b].

processor. T_1 to T_4 are the four threads running at each core. The L2 cache is shared between all the cores, while the L1 cache is internal to each core.

3.1.3 Related Niagara Research

As mentioned in the introduction of this section, the Niagara processor is bundled in two server models. Our test model is T1000, whereas German researchers D. an Mey, et. al., [aMSST06a], at RWTH Aachen University have performed an extensive test on the T2000.

As our test model is different from the model used for this research, it is necessary to examine the differences between the two system models, as listed in table 3.1. The table suggests that there are only a few differences between T1000 and T2000 from a DBMS point of view. The vital differences are the memory capacity, which is doubled at the T2000, and a slightly faster CPU clock. However, we expect that this difference is insignificant with regards to the micro benchmarking to be performed.

Having established that the research performed on T2000 is also valid for our purposes, we can move on to examining the findings done by Mey et. al.

Establishing memory latency

According to [BJA05], the main memory latency is 100ns. Mey et. al. benchmarked memory latency by executing small C programs explained at the corresponding web site [aMSST06b]. The memory performance was benchmarked, and found to scale very well: It scales with only $129ns - 107ns = 22ns$ latency difference between 32 and 1 processes. The findings are in accordance with Sun's own figure. Similar benchmarks were also performed to establish the bandwidth. The memory performance analysis is thorough. It is found that it is profitable to distribute the processes across all eight cores when running up to eight processes. This yields better results than filling up two cores leaving the other cores idle (as previously mentioned, Niagara can execute four concurrent threads within each core).

The most surprising about this benchmark, is the result when running 16 processes. It is found that it is more profitable to start four threads on four cores (and thus leaving the other four cores idle), than to distribute the workload evenly across all nodes[2].

Other benchmarks

This section mentions other benchmarks performed that are considered less significant for our purposes.

- In the EPCC OpenMP Micro Benchmark, Sun Fire T2000 is benchmarked contra Sun Fire E2900, testing synchronization overhead and loop scheduling overhead. The T2000 outperforms the competitor by a factor of two. Mey et. al. relates this finding to the difference in memory latency.

- A benchmark regarding sorting of integers finds that T2000 indeed is outperformed by E2900, due to its smaller caches than its competitor.

- In the Integer Stream Benchmark, a benchmark called The OpenMP Stream Benchmark was changed to do integer instead of floating point operations, as the processor does not handle floating point operations effectively compared to integer operations. It was evident that the UltraSPARC T1 system scales well, in some cases even up to 32 threads

- A benchmark measuring the performance of parallel partitioning of graphs with ParMETIS shows that Sun Fire E2900 outperforms T2000 by a factor of two, due to the fact that the competitor is a multi socket Symmetric Multiprocessing (SMP) machine, compared to the single socket architecture of Niagara.

- A benchmark regarding password cracking with "John the Ripper" shows that the Niagara processor does not scale due to the small L1 cache Niagara utilizes of 8KB. The competitor E2900 scales well, but has in comparison 64KB L1 cache. When more processes are running on each core on Niagara, the number of data misses increases dramatically.

3.2 Platform

The first part of the pre study is to examine the hardware platform on which the micro benchmarks are executed. Needless to say, prior knowledge of the platform is essential in order to be able to execute the benchmarks properly, as well as understanding the results returned. This section first presents facts and figures for the hardware. Second, the OS running on the described hardware is examined.

3.2.1 Hardware

It is necessary to go into the physical architecture of the T1000 Server in more detail. Figure 3.3 includes key figures which are used in section 6.2 when costs are calculated.

[2]This is surprising, but as we will see when we examine our own benchmark results in chapter 5 – Niagara is indeed full of surprises.

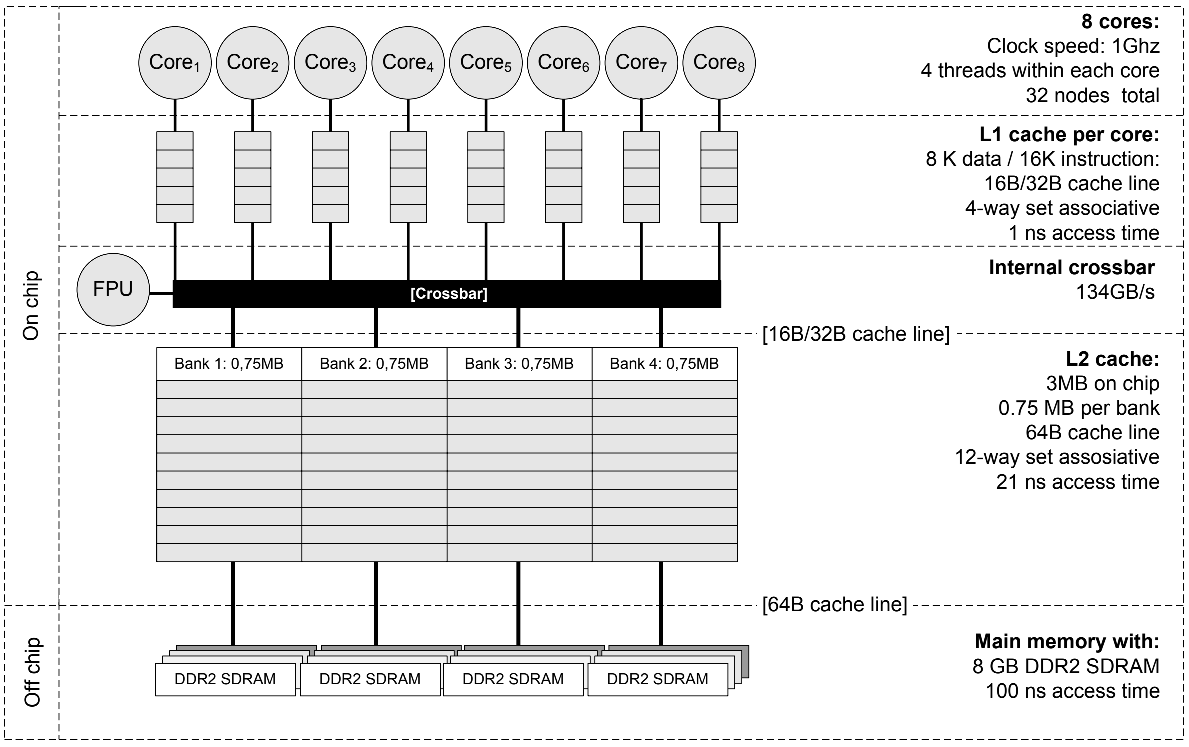


Figure 3.3: Niagara architecture with key figures included.

The details regarding the hardware on which the micro benchmarking is performed are listed in table 3.2. The details are gathered both directly from the server[3] and from the following sources:

- Sun paper "*Sun Fire T1000 Server Overview*" [Sun05d]
- Sun paper "*UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005*" [Sun06d]
- Sun blueprint "*Developing and Tuning Applications on UltraSPARC T1 Chip Multithreading Systems*" [She05]
- Sun paper "*The UltraSPARC T1 Processor - High Bandwidth For Throughput Computing*" [BJA05]
- The research "*Der UltraSPARCT1 Prozessor("Niagara") – Erste Erfahrungen*" by researchers D. an Mey, et. al.[aMSST06a].
- Sun white paper "*Sun Fire T1000 and T2000 Server Architecture*" [Sun05d]

3.2.2 Operating System

The Operating System (OS) running on the Sun Fire T1000 Server is an early build of Solaris 11. As explained in [Sun06a], Solaris is a UNIX based OS developed by Sun Microsystems. Solaris supports both x64-based, x86-based and SPARC-based systems. The Sun Fire T1000

[3]Issuing the command # **`/usr/platform/`uname -i`/sbin/prtdiag -v`** lists processor information.

Hardware	Sun Fire T1000 Server
System model:	Erie 64bit
Processor:	Ultra SPARC T1
Architecture:	Sun4V
Address space:	48-bit virtual 40-bit physical
Clock speed:	1.0 GHz
Cores:	8 cores running 4 threads, utilizing 32 logical CPUs
IPC:	One instruction per cycle per core
Cycle time:	$\frac{1}{1GHz} \approx 0,93ns$
L1 cache size (per core):	16 KB/8 KB (data/instruction). 4-way set-associative 16B/32B cache lines (data/instruction)
L1 access time:	1 cycle or $\approx 1ns$
L2 cache:	3 MB on chip, 4 banks, 12-way set-associative. 64B cache lines
L2 access time:	23 cycles ([Sun06d]) or $\approx 21ns$ (measured by [aMSST06a] to be $22ns$)
Memory:	8 slots that can be populated with one of the following types of DDR-2, 400 MHz DIMMS with ECC: 512 MB (4 GB maximum) 1 GB (8 GB maximum) 2 GB (16 GB maximum)
Memory size:	8GB [4]
Memory access time:	100ns ([BJA05] (measured by [aMSST06a]to be between $\approx 107ns$ (1 process) and $\approx 132ns$ (32 processes))

Table 3.2: Sun Fire T1000 "Erie" specification, utilizing the Niagara processor.

server is an example of the latter. At the time of writing, Solaris 10 is the featured OS from Sun, and Solaris 11 is not yet well documented. It is however natural to assume that the early build of Solaris 11 share most of the features in Solaris 10.

It is well worth pointing out that the OS running on our test machine is an early build. Although it claims to be optimized for Niagara, we have been unsuccessful in the search of finding affirmative documentation that states in detail to which degree the many optimization possibilities of Niagara indeed is implemented.

It is important that the OS actually is tailored to Niagara due to the fact that our micro benchmarks will be measuring low level system costs, whose results will depend on the level of support from the OS. This is valid for all the micro benchmarks presented later – all of them are in some degree dependent on the OS. When the results from the micro benchmarks are presented later, it is important to keep in mind that the results obtained are indeed dependent on the OS support for the hardware architecture on which it is running.

[4]Regarding the field *Memory* in table 3.2, we have not been able to figure out whether the main memory on our server were populated with 1GB or 2GB DIMMS. However, we do not expect that either should yield significant different results.

3.3 PROGRAMMING LANGUAGE

This section presents an assessment of possible programming language candidates to realize the benchmarks. It is important to decide upon this at an early stage, as the choice of programming language also restricts the implementation techniques to consider.

3.3.1 The objective of the micro benchmarking

The objective for micro benchmarking is to establish cost figures associated with the decomposed sub tasks of a transaction. These tasks are described in section 2.10. We expect these tasks to be indeed small and the time needed to execute them is not anticipated to exceed a couple of hundred μs at most. It is therefore important that the programming language does not add a considerable amount of overhead.

3.3.2 Programming language candidates

Having the previous section as a premise for the choice of programming language, the following candidate languages are considered:

- Java
- C# .net
- C++
- C
- DTrace

While the first four candidates are established, popular and well documented programming languages, the latter candidate needs an introduction.

Introduced in Solaris 10, Dynamic Tracing (DTrace) is a diagnostic monitoring tool for troubleshooting systemic problems in real time provided by the OS. DTrace allows the user to instrument the system by using the D programming language. Using DTrace, it is possible to monitor the system using a number of probes inserted in the Solaris OS. This may be helpful in order to find out exactly what is happening in the different parts of the system. DTrace is further explained in [Sun05c].

Both Java and C# are popular object oriented languages. However, Java utilizes a Virtual Machine between the running code and the machine. The .net-framework assumed by C# works in a similar fashion. Although ensuring portability, rich libraries and several other tractable features, these languages are not the answer to the quest of performing our micro benchmarks, measuring μs (and sometimes, even ns). Both JVM and the .net-framework would introduce a lot of overhead, and the results of the micro benchmarks would not be sufficiently accurate. In addition we do not expect to have the possibility to install such virtual machines on the Sun Fire T1000 Server.

Excluding the languages assuming virtual machines, C, it's object oriented big brother, C++ and DTrace are left. C++ is excluded because the programs used for the micro benchmarks are small and do not utilize objects of any kind. The C programming language is chosen over DTrace for the following reasons:

- The C programs running the micro benchmarks are easy to understand
- C is used by other similar benchmarks, i.e. [aMSST06a]
- C is a well known programming language [KR88] opposed to the less familiar probing tool DTrace

3.3.3 Shell scripting

Although the benchmarks are written in the C programming language, using the scripting facilities of a UNIX based OS significantly simplifies the automation in scheduling the micro benchmarks. The benchmark programs is to be executed a number of times with different parameters. Running the programs without automation is both error-prone and time consuming. As mentioned in 3.2.1, time is a limited resource. Therefore, the use of scripts is tractable for both reasons.

There are three main types of shells, with benefits and drawbacks:

sh The Bourne Shell is the original shell language, `sh`. It is virtually without bugs, and on UNIX systems, most of the scripts used to start and configure the operating system are written in this shell. However, it has a somewhat obtuse syntax compared to `csh`.

csh The C Shell. This shell is named so because of the similar syntactical structures to the C language. This is attractive, considering that the micro benchmarks themselves are written in C. However, according to the **`man`** pages of `csh`, it has quite a bug list opposed to `sh`

ksh The Korne Shell. This shell language is a superset of `sh`. In addition, it contains more advanced functionality on top. However, the `ksh` language is not as portable as `sh`.

Because of the constraint in time available when performing the benchmark, we settle for the original Bourne Shell language defined in [Ste78]. We can not afford to explore that a `csh` script contains bugs, or that the `ksh` language is not supported at testing phase on site.

Having settled for the `C` programming language and the `sh` shell language, the following section explores possible implementation techniques to realize the micro benchmarks.

3.4 TECHNIQUES

This section describes potential techniques to realize the micro benchmarks. As pointed out in section 2.10 at page 17, the significant costs to be micro benchmarked are:

- Sending and receiving messages
- Building messages and log posts
- Interpreting messages and log posts
- Writing to log
- Synchronizing shared data structures

Some of these costs can possibly yield different results depending on the technique used. The following subsections describe potential techniques to approach each of these costs.

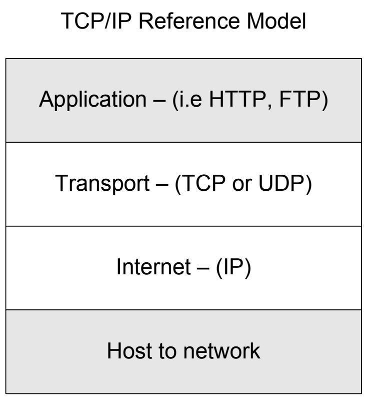


Figure 3.4: The TCP/IP reference model as given in [Tan96]. The Transport and Internet layer is the central layers for our purposes, they are therefore unshaded. IP is the underlying protocol on which the transport layer depends on. The task of the Internet layer is to provide correct packet routing. TCP is a reliable connection-oriented protocol that guarantees correct byte stream transmission from one machine to another. UDP is on the other hand an unreliable, connectionless protocol that does not utilize the sequence control of TCP, at the gain of less overhead involved.

3.4.1 Sending and receiving messages

We have identified five techniques to approach message passing between processes. All of them use some way of Inter process communication (IPC).

TCP Transmission Control Protocol (TCP) is a network protocol that guarantees reliable and in order delivery of sender to receiver data. Using TCP, applications on networked hosts can create connections to one another, over which they can exchange data or packets. As thoroughly explained in [Tan96], TCP is part of the TCP/IP reference model, named after its two primary protocols. As described in figure 3.4, TCP is a protocol in the transport layer in the TCP/IP reference model.

UDP As TCP, User Datagram Protocol (UDP) is a protocol in the transport layer in the TCP/IP reference model (figure 3.4). Contrasting TCP, UDP has less overhead, and is thus traditionally considered faster, in the face of no sequence and flow control. UDP is useful for applications where prompt delivery is more important than accurate delivery.

Solaris Doors According to [Ste99], Doors were first developed for the Spring distributed operating system. Doors later appeared in Solaris 2.5, although at an experimental stage. Solaris Doors is a Remote Procedure Call (RPC) programming interface that allows a procedure in one process to call a procedure in another process. When passing pointers to the messages as parameters, this provides a method for message passing. According to [Ste99], Solaris Doors is as fast, if not faster, than all other forms of message passing. This statement was made in 1999. It is not necessarily true today.

Fast Sockets Two researchers at Sun Microsystems have developed a library called `speed library` or `fastsockets`, as documented in [NV01]. The motivation for the development was that although Doors is the fastest IPC, socket based IPC (such as TCP) is portable, flexible and has the ability to communicate across a network in addition to communication between processes on the same processor. In the face of these properties, Doors IPC is cumbersome. Fastsockets was developed to meet these attractive properties while being based on Doors. Three different implementations of `fastsockets` are discussed in [NV01]:

- Using Doors IPC to transfer data and signal the other process of data availability
- Using memory maps to transfer data while using Doors IPC to signal data availability
- Using Doors IPC to set up the initial connection and memory mapping to transfer data, while using semaphores to signal the other process of data availability

System V Message Queues As explained in [Sun05a], System V IPC Message queues allow processes to place messages into a queue where any process can retrieve the message. Whereas System V IPC communication is reliable, it is also heavyweight. A process can create a new message queue, or it can connect to an existing one. In the latter case, two processes can exchange information through the same message queue[5].

3.4.2 Building messages and log posts

For our purposes, there is no need for sophisticated techniques in order to build a message. The `stdlib` library is sufficient. No messages or log posts are too big to be on the stack.

3.4.3 Interpreting messages and log posts

For our purposes, interpreting a message does not call for any special technique. Although some kind of parsing is required for interpreting, the `stdlib` library should be more than sufficient.

3.4.4 Writing to log

For our purposes, writing to log does not require other techniques than introducing the `malloc` call. This call is needed because the log buffer is potentially to big to be on the stack, and needs to be allocated on the heap.

3.4.5 Synchronizing shared data structures

We have identified two different candidate techniques to approach the problem of synchronizing shared data structures.

POSIX Mutexes POSIX Mutexes are either intra or inter process. A mutex guarantees mutually exclusive access to a shared resource at any given time. Threads that want to access a resource protected by a mutex must wait until the currently active thread is finished and unlock the mutex. Mutexes are easy to use, but can drastically slow down threaded code when overused.

[5]A sample ping-pong implementation of System V Queue is found at http://www.ecst.csuchico.edu/ beej/guide/ipc/mq.html

System V Semaphores Opposed to mutexes, semaphores allow simultaneous access to resources. Semaphores represent "available resources" that can be acquired by multiple threads at the same time until the resource pool is empty. Additional threads must then wait until the required number of resources are available again.

Chapter 4
Approach

This chapter describes the approaches to realize the micro benchmarks.

As explained in section 3.2.1, the server running the micro benchmarks is a Sun Fire T1000 Server, [Sun05e], code named *Erie*. The server is physically resident at a Sun Microsystems computer lab in California. Exclusive access to this machine is of quite high demand, but a two week exclusive time window was allocated to execute all the benchmarks. Due to the very limited amount of time available for us to perform the benchmarks, benchmarking every single DBMS cost is not feasible. Therefore, we settle for 6 micro benchmarks. Each benchmark are presented in detail in section 4.1. Section 4.2 then describes how shell scripting is used to automate the scheduling of the benchmarks.

4.1 Benchmark realization

Based on the significant costs presented in section 2.10 and the techniques presented in section 3.4, the following benchmarks are decided upon.

Benchmark 1: Message passing using TCP/IP sockets The first benchmark uses the TCP/IP protocol to send messages between logical nodes. We felt it natural to perform a TCP/IP benchmark. TCP/IP is well known, easy to use for the application programmer, portable and can scale to communicate over an external network. In addition, much work has been done in order to make TCP/IP efficient by short-circuit the stack as early as possible. Clearly, for our purposes, it is not necessary to go off chip. According to [Van05], Sun Microsystems has confirmed that the Niagara processor and Solaris 10 OS have indeed been tweaked for TCP/IP. The Sockets library `socket` defined in [Sun05b] is used realize this benchmark. This benchmark is described in section 4.1.1.

Benchmark 2: Message passing using Solaris Doors Solaris Doors is chosen because it is regarded as the fastest form of IPC. Although somewhat cumbersome to the application programmer, it is chosen over `fastsockets` (which also uses Solaris Doors for communication) because `fastsockets` is lacking documentation and appears to be in testing phase. A DBMS must rely on stable libraries. The Doors library `door` defined in [Sun05b] is used to realize this benchmark. This benchmark is described in section 4.1.2.

Benchmark 3: Building or interpreting messages and log posts Although these costs are not identical, it is natural to assume that the cost associated with building and interpreting messages are indeed very similar. Because of the tight time schedule, we choose to perform only one benchmark for these two costs, and apply the result to both tasks when calculating later. No special libraries are used to build or interpret messages and log posts. The benchmark makes use of standard C primitives. This benchmark is described in section 4.1.3.

Benchmark 4: Writing to log No special libraries are used to write to log. The benchmark makes use of standard C primitives. This benchmark is described in section 4.1.4.

Benchmark 5: Synchronizing using POSIX Mutexes For synchronizing intra process, we benchmark using POSIX mutexes. These are lightweight synchronization tools, and useful for synchronizing within processes. The POSIX Threads library `pthreads` defined in [Sun05b] is used to realize this benchmark. This benchmark is described in section 4.1.5.

Benchmark 6: Synchronizing using System V Semaphores Opposed to mutexes, semaphores allow simultaneous access to resources. Synchronizing between processes requires a shared memory area, and in Niagara, L2 cache is the highest such area in the system architecture. We did choose System V Semaphores over the POSIX equivalent for the following reasons:

- System V has been in use longer and is more documented
- We are only using a set of one semaphore, so even though POSIX are not as heavyweight as System V, it is natural to assume that the differences between the two are not significant.

System V Semaphore library defined in [Sun05b] is used to realize this benchmark. This benchmark is described in section 4.1.6.

The following properties exists for the benchmarks:

Ensuring simultaneous execution The benchmarks including more than one process are synchronized using semaphores to ensure that all processes start simultaneously. The motivation for this is to simulate a real-world approach, where such a condition often is the case.

The benchmark to be synchronized first starts a process that creates two semaphores, `start` and `finish`, and initializes both semaphores to zero. The processes[1] executing the main part of the benchmark, starts up by trying to grab the `start` semaphore. That is, they try to decrement it by one.

When all processes are pending on the `start` semaphore, the semaphore is increased to a value equal to the number of processes pending. All processes thus starts executing the main part of the benchmark simultaneously. After the `start` semaphore is increased, a caretaker process is started. This process tries to decrement the `finish` semaphore by a value equal to the number of processes. When the processes running the main part of the benchmark are finished, they increment the `finish` semaphore by one. Thus, when the main part of the benchmark is executed by all processes, the caretaker process wakes up, and removes both semaphores.

Running the benchmark in this manner does not only synchronize processes within the benchmark, but allows several benchmarks to be run sequentially using a shell script without risking that benchmarks interfere with each other.

Time keeping Time keeping is done using the Solaris specific function `gethrtime()` which returns high resolution time expressed in nanoseconds since some arbitrary time in the past.

[1]Processes should here be considered either processes or process pairs

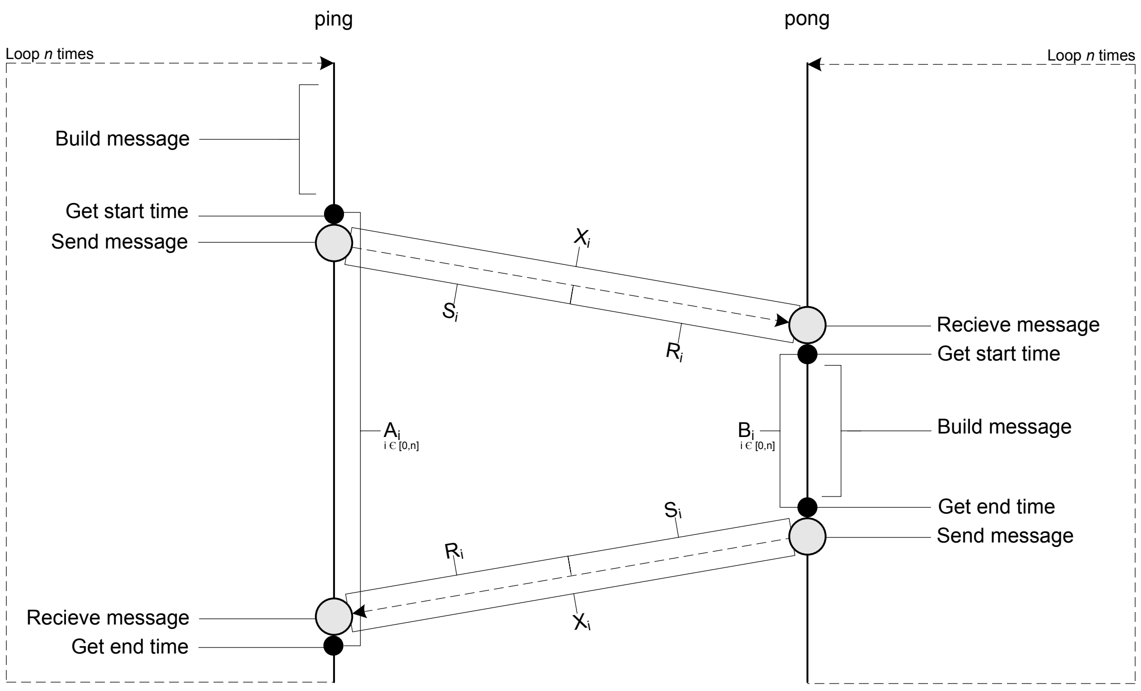


Figure 4.1: Establishing cost for message passing using TCP/IP sockets. The *ping* side builds a message and sends it to the *pong* side. The *pong* side builds a message of equal length and responds. On the *ping* side, time stamps are taken before sending the message and after receiving the response. On the *pong* side, time stamps are taken after receiving the message, and before responding. The total number of messages sent from each side is denoted n, whereas X_i is the time used to send and receive message i.

All programs are compiled using the C compiler `cc` that comes bundled with Sun Studio 11 [Sun06c]. The programs are compiled for Niagara using the following flags:

- `-fast` (specifying that the compiler should build speed optimized code)
- `-xtarget=ultraT1` (specifying that the target is Niagara)
- `-xchip=ultraT1`(specifying that the target chip is Niagara)
- `-xarch=v9b` (specifying the server architecture. `v9b` should be used for compiling on the 64-bit version of Niagara)

The following sections describe the approaches taken for the benchmarks listed above. Each section describes the micro benchmark in detail with input parameters and values, including a figure that illustrates the execution of the benchmark and the manner in which the time keeping is performed. The C programming code for all benchmarks are included in appendix C.

4.1.1 Benchmark 1: Message passing using TCP/IP sockets

In the SN case, it is important to establish the actual cost for sending and receiving messages between nodes on the Niagara chip. This benchmark uses TCP/Internet Protocol (IP) sockets to realize message passing. In order to find this cost, five programs are written, namely `ping.c`, `pong.c`, `pingpong_init.c`, `pingpong_start.c` and `pingpong_finish.c`. The three latter programs are used for synchronization purposes only, and are therefore not discussed further.

The execution is straight forward, as figure 4.1 suggests. First, the *ping* process builds and sends a message to the *pong* process. The *ping* side then waits for a response message from the *pong* process. At the *pong* side, a message (of equal length as the message just received) is built and sent back to the *ping* process.

The time used by the *ping* process is found by subtracting the system clock time value after receiving the *pong* response from the system clock time value before initially sending the message. In figure 4.1, this time interval is named A_i. The time used at the *pong* side is found in a similar fashion, by subtracting the system clock time value after sending the response back to the *ping* process from the system clock time value when the *pong* process received the message from the *ping* process. This time interval is named B_i in the figure. The i represents a number in a sequence of a total of n messages. In this benchmark n = 1 000 000, which means that both *ping* and *pong* send and receive 1 000 000 messages.

The time used to send and receive message i from one side to the other, (X_i in the figure) is found using the simple formula

$$X_i = \frac{A_i - B_i}{2}$$

The average time used to send *and* receive a message from one side to the other is found using

$$X = \frac{1}{2n}\sum_{i=0}^{n}(A_i - B_i)$$

In order to separate the time used to send a message from the time used to receive the message, we must assume that sending and receiving is equally expensive. Therefore, we set

$$S_i = R_i = \frac{X_i}{2}$$

Thus, the average time used to send *or* receive a message is calculated using formula 4.1.

$$X = \frac{1}{4n}\sum_{i=0}^{n}(A_i - B_i) \tag{4.1}$$

When running the benchmark, several parameters are varied as listed in table 4.1.

The message size is given as an argument directly in the code, whereas a script is utilized in order to be able to vary the number of process pairs. All shell scripts executed for this benchmark are presented in appendix B.1. The code for this benchmark is presented in appendix C.1.

Benchmark 1: Message passing using TCP/IP										
Messages	1 000 000									
Process pairs	2	4	6	8	16	32	64			
Msg size (B)	32	64	128	256	512	1024	2048	4096	8192	16384

Table 4.1: Input parameters to the benchmark for message passing using TCP/IP.

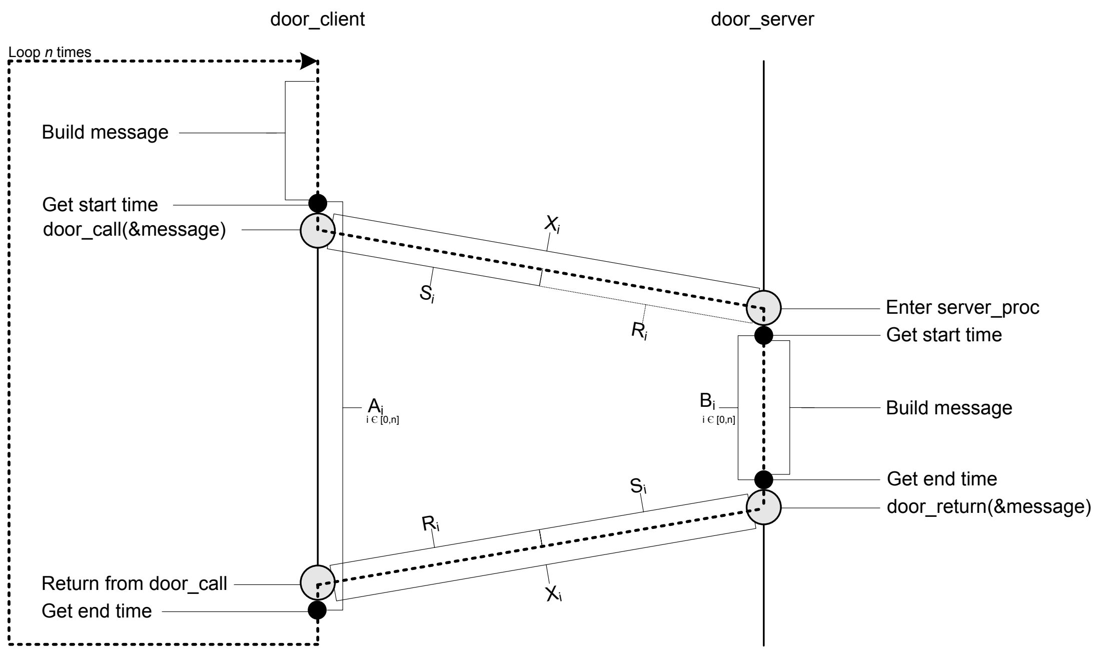


Figure 4.2: Establishing costs for sending and receiving messages using Solaris Doors. The *door_client* builds a message and passes the pointer to it to the *door_server* as an argument to the *door_call()* procedure. The server procedure which is invoked at the *door_server* side, builds a response message of equal length and passes a pointer to the message back to the *door_client* as an argument to the procedure *door_return()*. At the *door_client* side, time stamps are taken before and after the call to *door_call()*. At the *door_server* side, time stamps are taken at the beginning and the end of the server procedure. The total number of messages sent from each side is denoted n, whereas X_i is the time used to send and receive message i.

This benchmark is synchronized using semaphores as described in *"Ensuring simultaneous execution"* in section 4.1.

4.1.2 Benchmark 2: Message passing using Solaris Doors

As already explained in section 4.1.1, it is in the SN case important to point out the costs associated with sending and receiving messages between nodes on the Niagara chip. This section describes a benchmark, used to establish this cost, which uses Solaris Doors to realize message passing.

In order to establish the costs for sending and receiving messages when using Solaris Doors, five C programs are written. These are `doorserver.c`, `doorclient.c`, `door_init.c`, `door_start.c` and `door_finish.c`. The three latter programs are used for synchronization purposes only, and are therefore not discussed further.

As illustrated in figure 4.2, the *door_client* has a main loop that performs two task. The first task is to build a message. The second task is to send the message to the *door_server* by calling the function *door_call()* and passing a pointer to the message as a parameter. The function may also pass other parameters, but these are not of interest for our benchmark. Each time the *door_client* calls the server procedure, a thread in the *door_server* process handles the *door_client*'s call. Although this is what really happens, it is easier to understand the simplification that the *door_client*'s thread is executing the method inside the *door_server*, as illustrated in figure 4.2. The server procedure also consists of two tasks, namely building the response message and responding using the procedure *door_return()*. The response message is passed with the procedure as a parameter. Because RPC using Solaris Doors is synchronous, the *door_client* does not continue execution until the call to *door_call()* has returned. The time used for sending and receiving a message on the *door_client* side is measured by subtracting the time stamp taken after the call to *door_call()* from a time stamp taken before the call. This is illustrated in the figure. Similarly, to exclude the time not spent sending and receiving message, the *door_client* takes time stamps at the beginning and the end of the server procedure.

The time used to send and receive message i from one side to the other, (X_i) in the figure) is found using the simple formula

$$X_i = \frac{A_i - B_i}{2}$$

The average time used to send *and* receive a message is then calculated using the following formula

$$X = \frac{1}{2n}\sum_{i=0}^{n}(A_i - B_i)$$

As illustrated in the figure, A_i and B_i are the time used on the *door_client* and the *door_server* side, respectively, when sending message i from the *door_client* and responding with message i from the *door_server* . The number of messages sent from each side is denoted n. Thus, the total number of messages sent is $2n$.

In order to separate the costs for sending (S) from the cost for receiving (R), we must, as in the case for TCP/IP, assume that these costs are equal. Therefore, we set

$$S_i = R_i = \frac{X_i}{2}$$

Thus, the average time used to send *or* receive a message is calculated using formula 4.2:

$$X = \frac{1}{4n}\sum_{i=0}^{n}(A_i - B_i) \tag{4.2}$$

When the benchmark is run, several parameters are varied as listed in table 4.2. The message size is given as an argument directly in the code, whereas a script is utilized in order to vary the number of process pairs. As for the benchmark using TCP/IP, all processes sends and receives 1 000 000 messages. All shell scripts executed for this benchmark are presented in appendix B.2. The code for this benchmark is presented in appendix C.2.

This benchmark is synchronized using semaphores as described in "*Ensuring simultaneous execution*" in section 4.1.

Benchmark 2: Message passing using Solaris Doors										
Messages	1 000 000									
Process pairs	2	4	6	8	16	32	64			
Msg size (B)	32	64	128	256	512	1024	2048	4096	8192	16384

Table 4.2: Input parameters to the benchmark for message passing using Solaris Doors

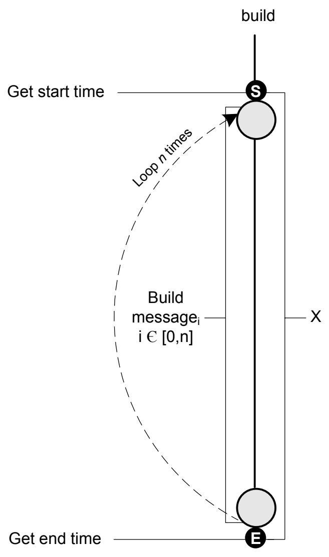


Figure 4.3: Establishing cost for building messages/posts. The program, `build.c`, builds n messages one byte at a time. The inner loop is excluded for simplicity. Time stamps are taken before and after the outer loop.

4.1.3 Benchmark 3: Building or interpreting messages and log posts

In the SN case, message passing and logging are two of the most important costs that may have an impact on the relative performance compared to the SE case. It is therefore important to establish the costs associated with these tasks. This section deals with this issue and describes a small program, `build.c`, used to point out these costs.

We assume that building a message and building a log post, is similar, and therefore the same benchmark is used to determine both costs. As stated earlier, the cost of building and the cost of interpreting is expected to be very similar. Therefore, the result from this benchmark will be used for calculation of both costs.

When a message/post is built, one does not necessarily know its length on beforehand because this depends on the contents, which may not be known until build time. In a complete DBMS, the contents of a message/post may be fetched from a variety of locations. Because we do not have a complete DBMS to our disposal, we do not have the possibility to simulate this property. We do however expect this shortcoming to be approximately compensated by our decision to insert words of size 1B, when building a message. Usually one would insert words of greater size, and thus achieve greater efficiency. Because all messages/posts used in this benchmark consist of predefined ASCII characters rather than values based on build time conditions, we expect our program to execute faster than would be the case in a real DBMS. We do however anticipate our decision to use words of size 1B to compensate for this probable miscalculation.

The program starts with creating an array, `buffer`, of the same size as the message/post size. A nested loop is performing the main part of the benchmark. The inner loop iterates over the `buffer`, filling out the message one byte at a time, while the outer loop iterates over the number of messages. This is illustrated in figure 4.3, except that the inner loop is excluded for simplicity. Time stamps are taken before the nested loop starts and after the nested loop ends. In the figure these points of time are named `S` and `E`, respectively. The number of messages/posts is denoted n. In this benchmark $n = 1\,000\,000$, which means that 1 000 000 messages/posts are built.

Benchmark 3: Building messages and log posts										
Messages	1 000 000									
Post size (B)	32	64	128	256	512	1024	2048	4096	8192	16384

Table 4.3: Input parameters to the benchmark for building messages posts

The average cost is then calculated using formula 4.3

$$\frac{E - S}{n} \tag{4.3}$$

When the benchmark is run, input parameters are varied as listed in table 4.3.

The message/post size is given as an argument directly to the `build.c` program, whereas the entire benchmark is scheduled using a script. The shell script executed for this benchmark is presented in appendix B.3. The code for this benchmark is presented in appendix C.3.

4.1.4 Benchmark 4: Writing to log

In the SN case, logging is performed time and again compared to the SE case. This is because distributed transactions are executed in compliance with the 2PC protocol in order to guarantee correctness, as described in [MLO86]. It is therefore of vital importance to establish the cost associated with writing a log post to memory.

We assume that the log is organized as a circular buffer. This means that the buffer size is fixed, and when the log buffer is full, the oldest part of the log is overwritten. Log posts are therefore written to buffer in a sequential manner, as illustrated in figure 4.4. In order to realize this, a C program is written, namely `myLog.c`. The program starts with allocating a memory area to be used as the circular buffer. The insertion of log posts is carried out through a nested loop. The outer loop iterates over the number of log posts, while the inner loop iterates over the size of a log post, filling ASCII character directly into the buffer. The outer loop loops a number of times equal to the number of log posts to be inserted. Posts are built directly into the buffer instead of being built first and the copied. For simplicity the inner loop is not illustrated in the figure.

Time stamps are taken before the nested loop starts and after the nested loop ends. In the figure, these points in time are named S and E, respectively. The average cost is calculated using formula 4.4

$$\frac{E - S}{n} \tag{4.4}$$

Inserting posts in this manner allows us to use the cache efficiently. This is because when inserting sequentially into an array of allocated memory, normal cache policy is to pre fetch cache words which will be needed in the near future. Therefore, less time is spent waiting for memory fetches.

However, it would not be realistic to fill up a cache line with log posts and then write all of them to the memory in a single batch, as this is not a property that is likely to occur in a real DBMS. To avoid this, there is a 256KB space between the log post inserted in the log. Sequential posts are therefore forced to different cache lines.

When the benchmark is run, several parameters are varied as listed in table 4.4. The post size and the buffer size are given as arguments directly to the `myLog.c` program, whereas the

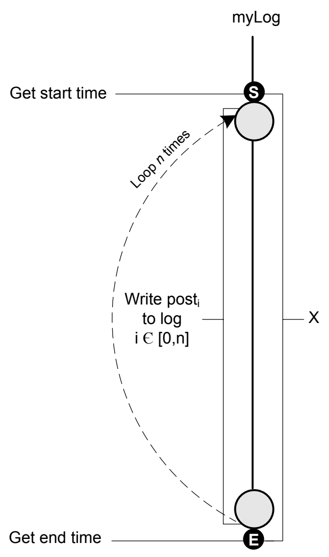


Figure 4.4: Establishing cost for writing to log. The program, `myLog.c`, builds n log posts directly into the buffer. There is a 256KB space between log posts to force subsequent posts to different cache lines. The inner loop is excluded for simplicity.

Benchmark 4: Writing to log										
Messages	1 000 000									
Post size (B)	32	64	128	256	512	1024	2048	4096	8192	16384
Buffer size (MB)	10	20	40	80	160	320	640	1280		

Table 4.4: Input parameters to the benchmark for writing to log

entire benchmark is scheduled using shell scripts. The shell scripts executed for this benchmark are presented in appendix B.4. The code for this benchmark is presented in appendix C.4.

4.1.5 Benchmark 5: Synchronizing using POSIX mutexes

In the SE case it is vital to determine the cost associated with synchronization. This is because such synchronization is not necessary in the SN case. The cost may therefore be cardinal to the relative performance of the two architectural approaches.

As mentioned in the introduction to section 4.1, this benchmark measures the cost of mutex synchronizing within a process. The cost of using mutexes does however continue to be interesting as they will much likely be needed to synchronize threads within a process. It is also interesting to compare the results with assumptions made by [BJ05]

To establish the costs related to setting and releasing mutexes a small program, `sr_mutex.c`, is written. The program creates a set of mutexes, and sets and releases each mutexes a number of times. This is illustrated in figure 4.5. The number of times each mutex is set and released is denoted n. The number of mutexes is denoted m. The program contains a nested loop. For each iteration of the outer loop, all of the mutexes are set and released in separate inner loops. Before and after each inner loop, time stamps are taken, separating the time spent setting from the time spent releasing.

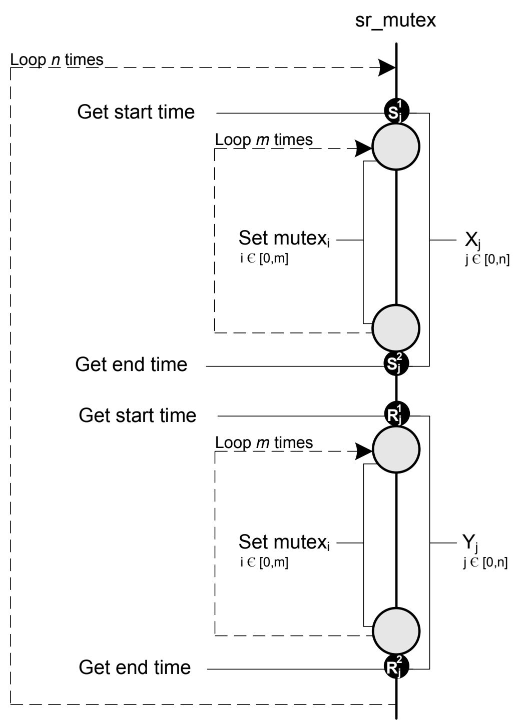


Figure 4.5: Establishing costs for setting and releasing mutexes. The program, `sr_mutex.c`, sets and releases m mutexes n times. This is executed in a nested loop. For each iteration of the outer loop, two inner loops iterate over a set of mutexes, setting and releasing every mutex one time. Time stamps are taken before and after both inner loops, separating the time used to set and release.

In the figure, these time stamps are named S_j^1 (before setting all mutexes the j'th time), S_j^2 (after setting all mutexes the j'th time), R_j^1 (before releasing all mutexes the j'th time) and R_j^2 (after releasing all mutexes the j'th time). At each iteration of the outer loop, the difference between the after and before time stamps are added to global variables keeping track of the total time spent setting and releasing.

The time used to set all mutexes the j'th time is calculated using the formula

$$X_j = S_j^2 - S_j^1$$

Similarly, the formula for the time used to release all mutexes the j'th time is

$$Y_j = R_j^2 - R_j^1$$

Benchmark 5: Synchronizing using POSIX Mutexes						
Mutexes	1	10	1000	10000		
Sets / releases	100 000 000			10 000 000	1 000 000	100 000

Table 4.5: Input parameters to the benchmark for setting and releasing mutexes

The average time used to set a mutex is then calculated using formula 4.5

$$X = \frac{1}{mn} \sum_{j=0}^{n} X_j \tag{4.5}$$

The average time used to release a mutex is calculated using formula 4.6

$$Y = \frac{1}{mn} \sum_{j=0}^{n} Y_j \tag{4.6}$$

When the benchmark is run, several parameters are varied as listed in figure 4.5.

In contrast with the other benchmarks not all combinations of input parameters are used. Only the parameters satisfying the expression $m \cdot n = 100\,000\,000$ are included.

Both the number of mutexes and the number of sets and releases per mutex are given as arguments directly to the `sr_mutex.c` program, whereas the entire benchmark is scheduled using a shell script. The shell script executed for this benchmark is presented in appendix B.5. The code for this benchmark is presented in appendix C.5.

4.1.6 Benchmark 6: Synchronizing using System V semaphores

As explained in section 4.1.5, it is important to point out the cost associated with synchronizing in the SE case. Whereas section 4.1.5 explains how this cost is benchmarked when mutexes are used as the synchronization primitive, this section explains how this is done when the benchmark uses semaphores.

To establish the cost associated with setting and releasing semaphores, four programs are written, namely `lseminit.c`, `lsemstart.c`, `lsemset.c` and `lsemrm.c`. The first program creates a semaphore `main`, and initializes its value to be equal to the number of process that will try to grab it. Thus, when several processes try to grab the semaphore simultaneously, they will not have to wait for the semaphore to be available. The processes will however need to wait for the built-in synchronization of the operations on the semaphore.

As illustrated in figure 4.6, the program `lsemset.c` contains a main loop that iterates as many times as the number of sets and releases of the semaphore. This number is denoted n. At each iteration, the semaphore is set and released once. Time stamps are taken before and after the set operation and before and after the release operation. In the figure these time stamps are named S_i^1 (before setting the i'th time), S_i^2 (after setting the i'th time), R_i^1 (before releasing the i'th time) and R_i^2 (after releasing the i'th time). At each iteration the differences between the before and after values are added to global variables keeping track of the total time spent setting and releasing.

The time used to set the semaphore the i'th time is thus calculated using the formula

$$X_i = S_i^2 - S_i^1$$

Similarly, the time used to release the semaphore the i'th time is calculated using the formula

$$Y_i = R_i^2 - R_i^1$$

The average time used to set a semaphore is calculated using formula 4.7

$$X = \frac{1}{n} \sum_{i=0}^{n} X_i \tag{4.7}$$

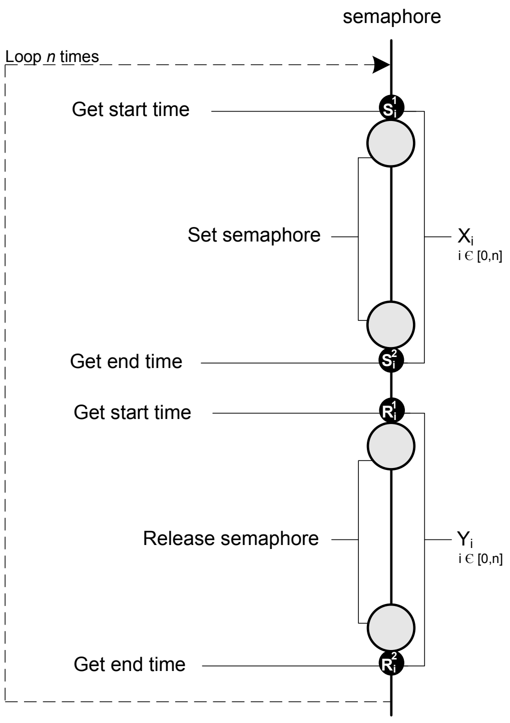


Figure 4.6: Establishing cost for setting and releasing semaphores. The program, `lsemset.c`, sets and releases a semaphore n times. Time stamps are taken before and after the set operation, and before and after the release operation. At each iteration the differences between the before and after time stamps are added to global variables keeping track of the total time spent setting and releasing.

The average time used to release a semaphore is calculated using formula 4.8

$$Y = \frac{1}{n}\sum_{j=0}^{n} Y_i \tag{4.8}$$

When this benchmark is run, the number of process pairs is varied as listed in table 4.6

This benchmark also uses semaphores to synchronize its execution as described in section 4.1 at page 33. Therefore, the program `lseminit.c` also creates the semaphores `start` and `finish`. These semaphores are only used for synchronization of the benchmark and should not be confused with the `main` semaphore. Other programs that also are used to provide such synchronization are `lsemstart.c` and `lsemrm.c`.

The shell scripts executed for this benchmark are presented in appendix B.6. The code for this benchmark is presented in appendix C.6.

Benchmark 6: Synchronizing using System V Semaphores							
Sets / releases	1 000 000						
Processes	2	4	6	8	16	32	64

Table 4.6: Input parameters to the benchmark for setting and releasing semaphores

4.2 Benchmark scheduling

This section describes how scripting is used to automate the scheduling of the benchmarks. The advantages of running the benchmarks in this manner compared to running them manually are clear. First, the parameters are easily given as input, reducing the chances of erroneous input parameters. Second, it is possible to schedule batches of benchmarks to be run, minimizing the human interaction. This allows for benchmarks to be run overnight. Third, scripting simplifies the task of maintaining consistent log file naming.

The following subsections describe how shell scripts are used to schedule each benchmark.

4.2.1 Benchmark 1: Message passing using TCP/IP sockets

Benchmark 1 is scheduled by using four `sh` scripts, namely `run_pong`, `run_ping`, `rpps` (short for `run ping-pong scheduler`) and `run_pp_benchmark`. Both scripts are presented in appendix B.1. The following sections describe each script.

`run_pong` and `run_ping`

The scripts `run_pong` and `run_ping` are responsible for starting a given number of *pong* and *ping* processes by executing the programs compiled from `pong.c` and `ping.c`, respectively. The scripts take three input parameters:

- Number of messages
- Message size
- Number of process pairs

Each script uses the **`pbind`** command to bind a process to a logical processor. The only difference between the scripts is that the calls to **`pbind`** made in `run_ping` are in the reverse order of the calls made in `run_pong`. This is done in order to ensure that *ping* and *pong* are indeed executed on different logical processors. All the started processes are executed as background processes.

`rpps`

The script `rpps` is a small script that takes the same three input parameters as `run_pong` and `run_ping`. This script is responsible for starting a given number of process pairs by executing `run_pong` and `run_ping`, passing on the input parameters.. The script starts with running the program compiled from `pingpong_init.c`. This program creates the semaphores `start` and `finish` used for synchronization, as explained in section 4.1.

Next, the script starts all the *pong* processes by executing the script `run_pong`. When all the *pong* processes are started, the script starts all the *ping* processes by executing `run_ping`. When all the *ping* processes are waiting for the `start` semaphore as explained in section 4.1, the program compiled from `pingpong_start.c` is executed. This program increases the value of the `start` semaphore, so that all the *ping* processes wakes up.

Next, the program compiled from `pingpong_finish.c` is executed. This process is used to remove the `start` and `finish` semaphores. However, in order to use this process as a way of telling when all the process pairs are done passing messages, the semaphores cannot be removed straight away. The process therefore waits for the `finish` semaphore to reach a value equal to the number of process pairs. When a process pair is done passing messages, the *ping* side increases the value of the `finish` semaphore. Thus, when all process pairs are done, the *pingpong_finish* process removes both the `start` and thr `finish` semaphores.

`run_pingpong_benchmark`

In order to minimize human intervention, the script `run_pingpong_benchmark` is responsible for scheduling the entire benchmark. This is done by serially executing `rpps` several times, giving different input parameters. The parameters are listed in table 4.1 in section 4.1.1 at page 36. Because `rpps` does not finish before the last process pair is finished (because of the *pingpong_finish* process), the next test is assured not to begin before the current test is finished. A five second sleep is placed between two tests to ensure that the system is not cluttered from the previous test.

4.2.2 Benchmark 2: Message passing using Solaris Doors

Benchmark 2 is scheduled by using two `sh` scripts, namely `run_doors` and `run_doors_benchmark`. Both scripts are presented in appendix B.2. The following sections describe each script.

`run_doors`

The script `run_doors` is responsible for starting a given number of *door_server* and *door_client* process pairs, by executing the programs compiled from *doorserver.c* and `doorclient.c`, respectively.

The approach is very similar to the approach taken for benchmark 1, but instead of using three scripts to start a given number of process pairs and to control synchronization, the benchmark uses one script. Thus, `run_doors` may be regarded as a concatenation of the three scripts, `run_pong`, `run_ping` and `rpps`, described in the previous section. Of course, *doors* process pairs are started, not *pingpong* process pairs. The script takes three input parameters:

- Message size
- Number of process pairs
- Number of messages

The script starts with executing the program compiled from `door_init.c`. As for benchmark 1, this program creates the semaphores `start` and `finish` used for synchronization, as explained in section 4.1. Both semaphores are initialized to have a value of zero.

Next, the desired number of *door_server* processes are started by executing the program compiled from *doorserver.c* several times. Each process is bound to a specific logical processor by issuing the command **`pbind`**.

Next, the script starts a number of *door_client* processes, equal to the number of *door_server*s. The calls to **`pbind`** made when starting the *door_client*s are in the reverse order of the calls made when starting the *door_server*s. This is done in order to ensure that matching pairs of servers and clients are indeed executed on different logical processors. All *door_server* and *door_client* processes are run as background processes.

The first thing a *door_client* does is to try to grab the `start` semaphore. Because the `start` semaphore is initialized to have a value of zero, all *door_client* processes halt their execution.

When all the *door_client* processes are waiting for the `start` semaphore as explained in section 4.1, the program compiled from `door_start.c` is executed. This program increases the value of the `start` semaphore, so that all the *door_client* processes wake up.

When the message passing has begun, the program compiled from `door_finish.c` is executed. This process tries to remove all the semaphores. However, as described for benchmark 1, this cannot be done before the process pairs are done passing messages. Thus, the `door_finish` process waits for the semaphore to reach a value equal to the number of process pairs.

When a *door_server* process has received the last message, it increases the `finish` semaphore by one before terminating. Thus, when all process pairs are done passing messages, the `door_finish` process wakes up, and removes both the `start` and the `finish` semaphores.

run_doors_benchmark

The script `run_doors_benchmark` is used to schedule the entire benchmark. This is done in order to minimize the amount of human intervention. The script serially executes `run_doors` several times, passing the parameters given in table 4.2 in section 4.1.2 at page 38. To ensure that resources used by the previous test are released before the next test commences, a five second sleep is placed between any two tests.

4.2.3 Benchmark 3: Building or interpreting messages and log posts

Benchmark 3 uses one `sh` script to schedule the entire benchmark, namely `run_build`. Because this benchmark does not include simultaneous processes, there is no need to synchronize using semaphores. The script is presented in appendix B.3. The following section describes `run_build`.

run_build

The script `run_build` is responsible for starting several processes of the program compiled from `build.c`. These processes are executed serially. The script does not take any parameters, but passes parameters to the build processes according to table 4.3 in section 4.1.3 at page 40.

4.2.4 Benchmark 4: Writing to log

Benchmark 4 is scheduled by using two `sh` scripts, namely `run_myLog` and `run_myLog_benchmark`. Like benchmark 3, this benchmark does not include simultaneous processes and is therefore not synchronized using semaphores. Both scripts are listed in appendix B.4. The following sections describe both scripts.

run_myLog

The script `run_myLog` is responsible for starting several processes of the program compiled from `myLog.c`. This is done serially. The script takes one input parameter:

- Buffer size

The program compiled from `myLog.c`, however, takes three parameters. These are

- Number of posts
- Buffer size
- Post size

The number of posts is held constant for all tests. The buffer size is held constant within the `run_myLog` script, whereas the post size is varied according to table 4.4 in section 4.1.4 at page 41.

`run_myLog_benchmark`

In order to vary the buffer size according to table 4.4 at page 41, `run_myLog` is executed several times, serially with different input parameters. This is done in the script `run_myLog_benchmark`. In order to ensure that the system is not cluttered from the previous test, a five second sleep is placed between any two tests.

4.2.5 Benchmark 5: Synchronizing using POSIX Mutexes

Benchmark 5 is scheduled by using the single `sh` script `run_sr_mutex`. Like benchmark 3 and 4, this benchmark is not synchronized by using semaphores. The script is presented in appendix B.5. The following section describes `run_sr_mutex`.

`run_sr_mutex`

The script is responsible of starting several processes of the program compiled from `sr_mutex.c`. This is done serially. The script does not take any input parameters, but passes parameters to the processes according to table 4.5 at page 42.

4.2.6 Benchmark 6: Synchronizing using System V Semaphores

Benchmark 6 is scheduled by using two `sh` script, namely `run_lem` and `run_sem_benchmark`. Like benchmark 1 and 2, this benchmark is synchronized using semaphores. Both scripts are presented in appendix B.6. The following sections describe both scripts.

`run_lsem`

The approach taken by this script is very similar to the one in `run_doors` described in section 4.2.2. The script takes two input parameters:

- Number of processes
- Number of sets and releases on the semaphore

The script starts with executing the program compiled from `lseminit.c`. This program creates two semaphores, `start` and `finish`, which are used for synchronization as described earlier. Both semaphores are initialized to zero. In addition, the program creates a third semaphore, `main`, which is used by the benchmark to measure the costs.

Next, the desired number of processes to set and release the semaphore are created by executing the program compiled from `lsemset.c`. The processes are bound to different logical processors by issuing the command **pbind**. Like the *ping* and *door_client* processes, described in sections 4.2.1 and 4.2.2, respectively, this process tries to grab the `start` semaphore. Because this semaphore is initialized to zero, the execution halts.

When all processes are waiting on the `start` semaphore, the program compiled from `lsemstart.c` is executed. This program sets the value of the `start` semaphore to be equal to the number of processes trying to perform operations on the `main` semaphore. Thus, all the processes wakes up.

When the setting and releasing of the `main` semaphore has begun, the program compiled from `lsemrm.c` is executed. This program tries to remove all three semaphore. However, this cannot be done before all of the *lsemset* processes have terminated. Thus, the *lsemrm* process waits for the `finish` semaphore to reach a value equal to the number of *lsemset* processes.

When an *lsemset* process has released the `main` semaphore the last time, it increases the value of the `finish` semaphore by one before the process terminates. In this way, when all the `lsemset` processes have terminated, the *lsemrm* process wakes up and removes all three semaphores.

run_sem_benchmark

The script `run_sem_benchmark` is used to schedule the entire benchmark. As for the other benchmarks, this is done in order to minimize the amount of human intervention. The script serially executes `run_lsem` several times, passing the parameters given in table 4.6 in section 4.1.6 at page 44. To ensure that the system is not cluttered from the previous test a five second sleep is placed between any two tests.

CHAPTER 5 RESULTS

This chapter presents the results obtained from the benchmarks listed below and described in chapter 4.

- Benchmark 1: Message passing using TCP/IP
- Benchmark 2: Message passing using Solaris Doors
- Benchmark 3: Building or interpreting messages and log posts
- Benchmark 4: Writing to log
- Benchmark 5: Synchronizing using POSIX Mutexes
- Benchmark 6: Synchronizing using System V Semaphores

The following sections describe and explain results obtained from running the benchmarks.

Inspecting the hardware configuration of the T1000 [1], it is apparent that the T1000 strength is parallelism. We suspect that it might not excel at tasks that do not involve parallelism. For comparison, we therefore ran Benchmarks 3-5 on a Sun Fire V890 server. The V890 server has four CPUs, each running at 1350 MHz. The total amount of main memory is 16GB. This server resides at Sun Microsystems' database development premises in Trondheim, Norway. However, it was not possible to obtain exclusive access to this server, and therefore not possible to control the server's total workload. Nonetheless, for comparison, the results from the V890 Server is presented in conjunction with the corresponding benchmark.

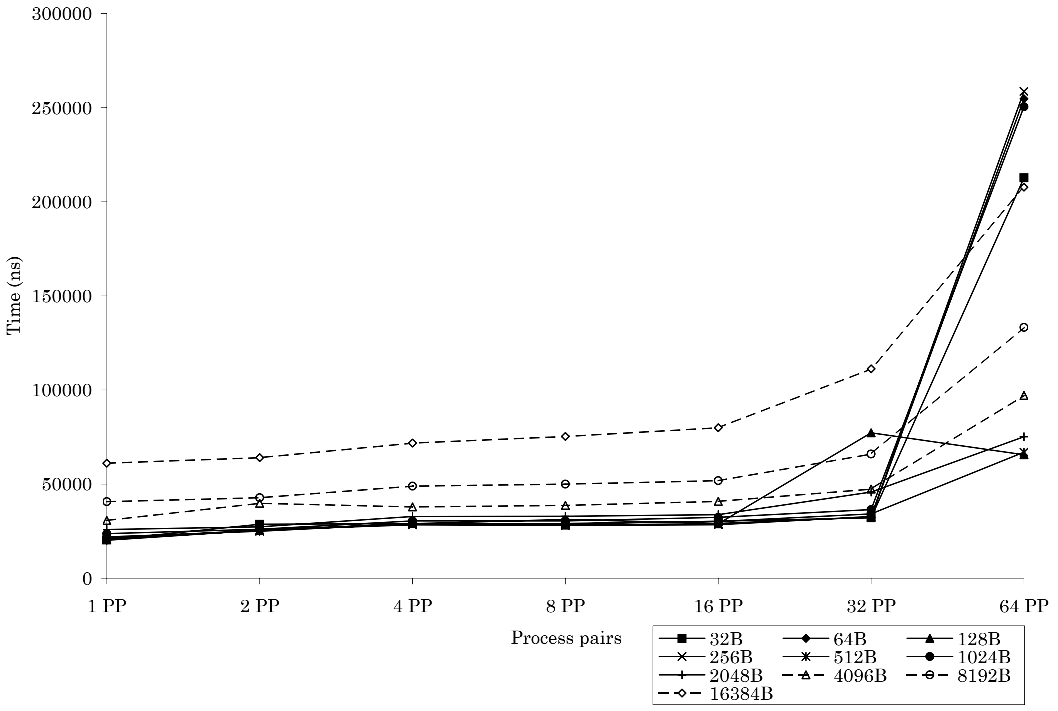
5.1 BENCHMARK 1: MESSAGE PASSING USING TCP/IP SOCKETS

This section describes the cost figures derived from running `ping.c` and `pong.c`, described in section 4.1.1.

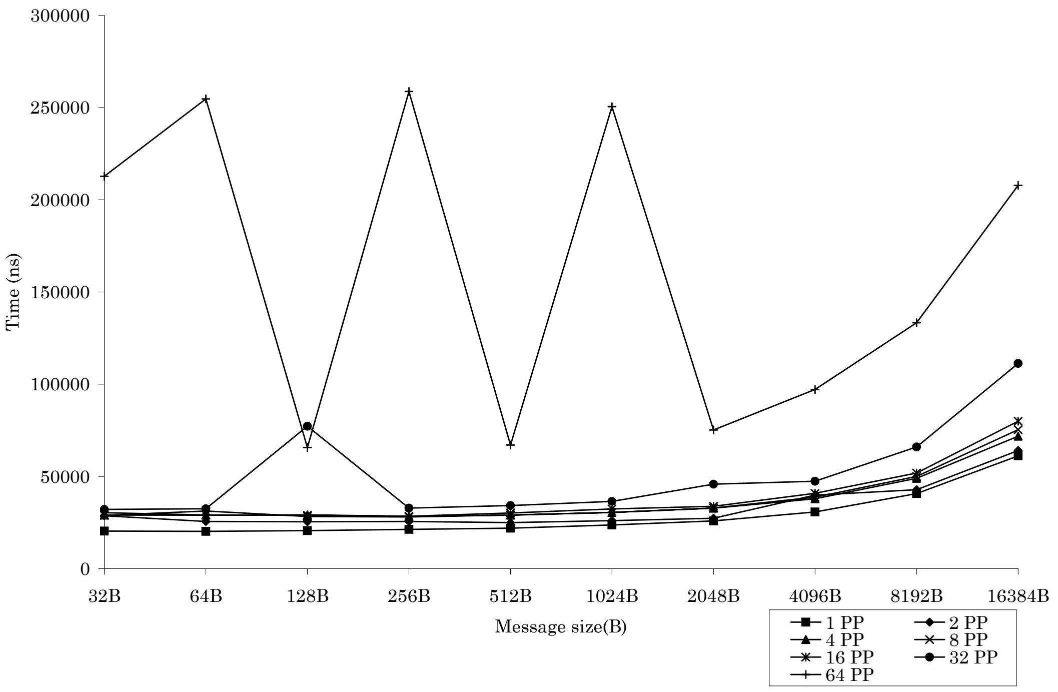
Figure 5.1(b) shows the costs associated with sending messages when the number of process pairs vary between 2 and 64. There are 10 curves in the figure, each representing a different message size, ranging from 32B to 16KB. As expected, the cost grows as the message size is increased. However, very little distinguishes the curves representing messages of size 32B up to and included 1024B. A possible explanation to this is that the maximum frame size for Ethernet is 1518B, giving an IP Maximum Transmission Unit (MTU)[2] of 1500B. This is further explained in [Hor84]. Messages smaller than 1500B are therefore all contained within a single Ethernet frame, and thus, the cost differences are indeed small. The messages of size 2KB,

[1]The T1000 hardware configuration is discussed in section 3.2 at page 24

[2]Size of the largest datagram that a given layer of a communications protocol can pass onwards.



(a)

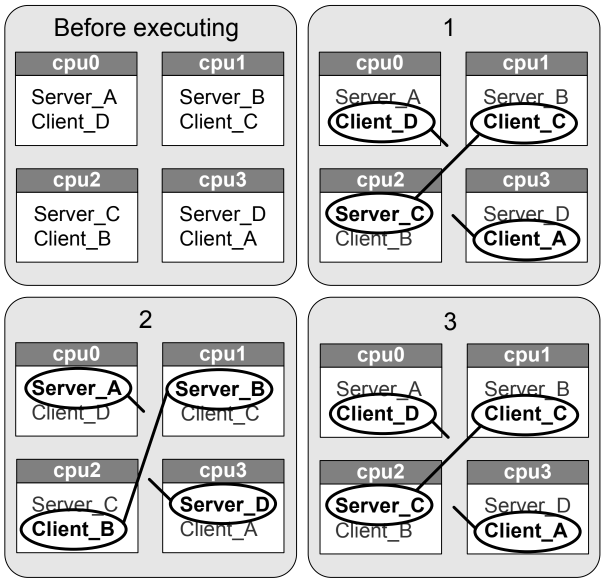


(b)

Figure 5.1: Cost figures for sending/receiving messages using TCP/IP sockets. In (a), the number of process pairs is varied. In (b), the message size is varied. The cost is the average time used to send/receive a message.

Process	Binds at
Server_A	cpu0
Server_B	cpu1
Server_C	cpu2
Server_D	cpu3
Client_A	cpu3
Client_B	cpu2
Client_C	cpu1
Client_D	cpu0

(a)



(b)

Figure 5.2: The interleaving of process execution. The processes bind to different processors, as illustrated in (a). Figure (b) illustrates that process pairs `A` and `D` have problems passing messages, because the server sides and the client sides are not scheduled to run simultaneously.

4KB, 8KB and 16KB, on the other hand, amounts to $\lceil \frac{2048B}{1500B} \rceil = 2$, $\lceil \frac{4096B}{1500B} \rceil = 3$, $\lceil \frac{8192B}{1500B} \rceil = 6$ and $\lceil \frac{16384B}{1500B} \rceil = 11$ frames, respectively. Therefore, there is a leap in costs associated with sending or receiving messages of these sizes.

Figure 5.1(b) also shows that until 16 process pairs are exceeded, the costs for sending or receiving messages of different sizes are fairly stable. There is a small increase in time used when going from 16 to 32 process pairs. At 32 process pairs, all nodes run two processes, one *ping* process, and one `pong` process. When the number of process pairs reaches 64, all nodes run four processes, and the cost of sending messages increases dramatically.

Figure 5.1(a) shows the costs associated with sending messages when the message size vary from 32B to 16KB. There are 7 curves in the figure, each representing a different number of concurrent process pairs. As suggested by figure 5.1(b), the cost associated with sending a message is fairly stable until the message size exceeds the size of a maximum IP MTU. It is also evident that running more than 32 concurrent process pairs results in a dramatic increase in the average cost per message.

The curve representing 64 process pairs is fluctuating frequently. There may be several reasons to this. One possible reason that might seem obvious, is the chance of that one test is not done finished before the next is started, and that this simply is a defect in the benchmark. However, this is not probable as the benchmark is synchronized using semaphores in order to ensure that the current test is done before the next begins.

Queuing

Queuing is probable for all message pairs. In order to simulate a real-world workload, semaphores are introduced in order to be certain that all message pairs indeed starts simultaneously. Because there are two (or more) processes per logical CPU, the processes are subject to queuing. Which process to execute first is decided by the OS. For 32 or less process pairs, there are two processes per logical CPU, and thus 50% probability for two corresponding processes to be executed simultaneously. For 64 process pairs, there are four processes per node, thus reducing the probability to 25% for the two corresponding messages to be executed simultaneously. An example of the race between two processes per node is illustrated in figure 5.2.

Consider the scenario where the processes are bound to CPUs as described in table 5.2(a). A CPU must execute parallel processes in an interleaved fashion. The process not currently executed is therefore queued. As illustrated in step 1 in figure 5.2(b), *Client_C* and *Server_C* are executed simultaneously at different CPUs. This process pair may therefore send and receive messages as normal. *Client_D* and *Client_A*, however, are not scheduled to run simultaneously as their respective server sides. Their time is therefore spent waiting for an answer from the servers. In step 2, the previous processes are switched. *Client_B* and *Server_B* are communicating normally, whereas process pairs *D* and *A* are still not communicating as both client side processes has been exchanged with their server side processes. Their execution is therefore still halting. A similar scenario is illustrated in step 3.

Queuing as described above may lead to unfortunate scenarios. If it is possible to control the execution of the benchmark one might achieve a curve for 64 process pairs similar to the curve one would get when excluding the upper extremes. This curve will indeed have the same trends as the curves representing fewer process pairs. However, because Solaris 11 is not yet well documented, to predict how this is solved by the OS is not straight forward.

5.2 BENCHMARK 2:MESSAGE PASSING USING SOLARIS DOORS

This section describes the cost figures derived from running `door_client.c` and `door_server.c`, described in section 4.1.2.

Figure 5.3(b) shows the costs associated with sending messages when the number of process pairs vary between 1 and 64.There are 12 curves in figure 5.3(b), each representing a different message size, ranging from 32B to 16KB.

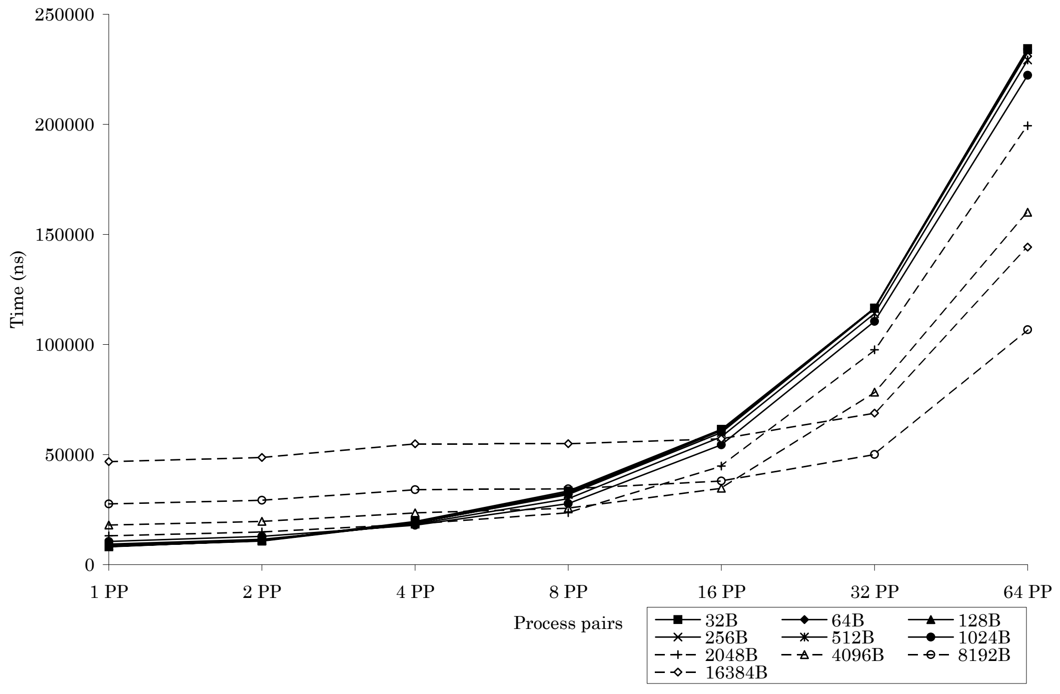
In the following, we short-name the processes of message passing with a message size ranging from 32B to 512B *small messages*. The message size range from 1KB to 16KB are denoted *large messages*.

The cost associated with sending a message is fairly stable until the message size exceeds the 1024B. From message size ranging from 32B – 1024B, the cost is virtually equal for each number of process pair(s). Indeed, these curves overlap for all process pairs. However, the four dashed curves, denoting message sizes from 2KB up to and included 16KB does not follow the same pattern. The curves for these message sizes have a surprising evolution as the number of process pairs increase.

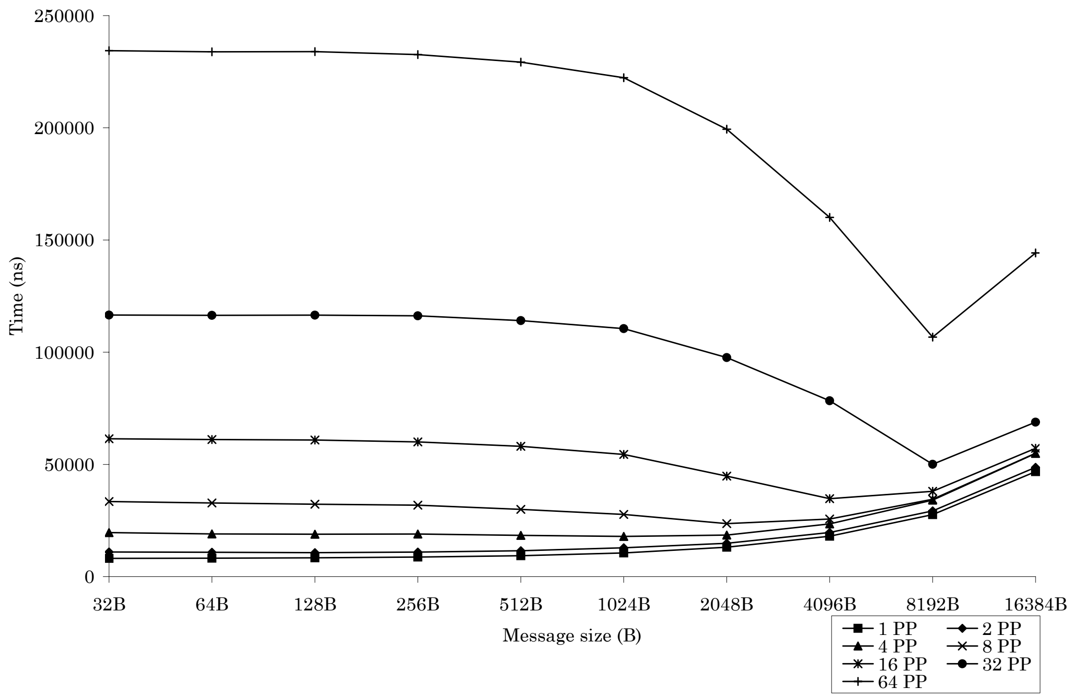
For up to 4 process pairs, the relative difference between these curves are maintained. 16KB has twice the cost of 8KB, which in turn has twice the cost of 4KB, which in turn has twice the cost of 2KB. Starting at 8 process pairs, the ranking begins to change. At 64 process pairs, the ranking is almost inversed: $2 > 4 > 16 > 8$. This surprising effect is perhaps more apparent when the data material is transposed in figure 5.3(a).

Figure 5.3(a) shows the costs associated with sending messages when the message size vary from 32b to 16Kb. There are 7 curves in the figure, each representing a different number of concurrent process pair.

Although the graph appear to lead to the conclusion that sending large messages cost lest than sending small messages, it is important to realize that this is indeed not the case. The graphs are visualizing the cost of message passing, excluding the time to build the message.



(a)



(b)

Figure 5.3: Cost figures for sending/receiving messages using Solaris Doors. In (a), the number of process pairs is varied. In (b), the message size is varied. The cost is the average time used to send/receive a message.

The time to build a message is also a cost that needs to be taken into account before concluding about the total cost.

We present the results from *Benchmark 3: Building or interpreting messages and log posts* in section 5.3. Benchmark 3 suggests that the cost of building a message nearly doubles as the message size increases. Indeed, the results from the Doors benchmark strengthens this suggestion. As explained in figure 4.2 at page 37, the time on the *door_server* includes the time to build the server-side reply message. Although there is more to the server side than building the message, the time used on the server side is two sizes of magnitude larger for the 16KB message size than the 32B message. All test result data from the Solaris Doors micro benchmark are listed in appendix D.2.

To sum up, if we were to send the maximum number of messages in a given time slot, we would end up with sending far more smaller messages than larger, due to the increasing cost of building messages of increasing larger messages. Still, the graph is surprising, in the way that the time actually used on transmitting the message drops as the message size increase. We have identified two possible explanations to this phenomena:

Thrashing

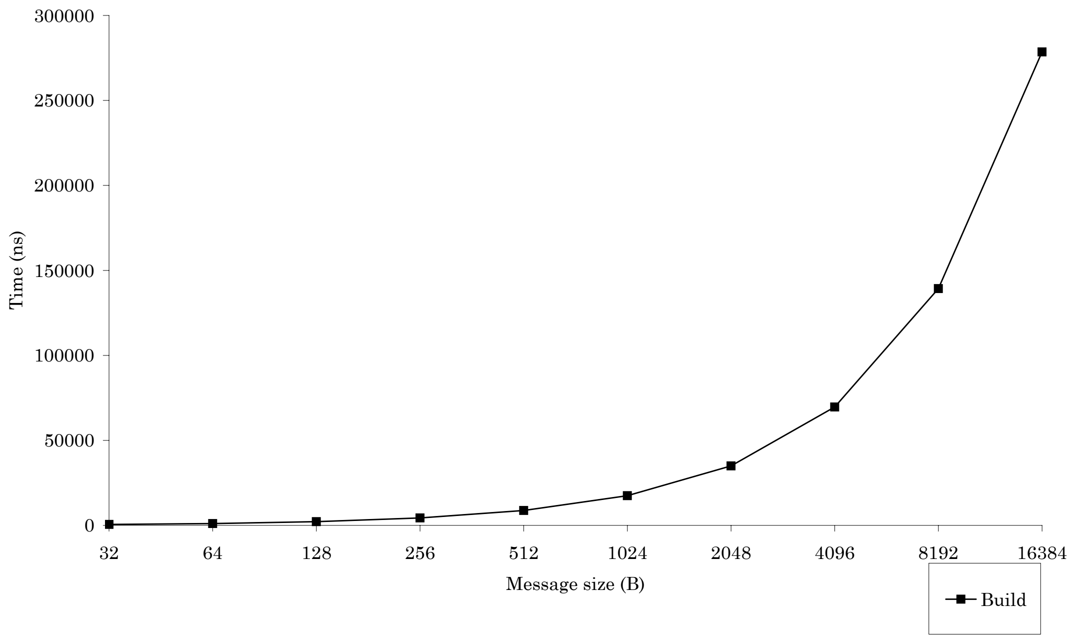
We suspect that Doors is subject to heavy trashing caused by the OS. As described in section 4.1.2, the Doors communication mechanism could be abstracted as a method in a server process that a calling process is able to execute. The server process receives the message as a parameter from the caller. In a OS provided shared memory space, the server process responds to the caller by sending the message as a parameter in the *door_return* call. Given that the shared memory space is shared for all process pairs, and not separate for each pair of processes, thrashing in this space would be more frequent the smaller the message.

To see why this is the case, it is necessary to look into the details of the sending and receiving mechanism.

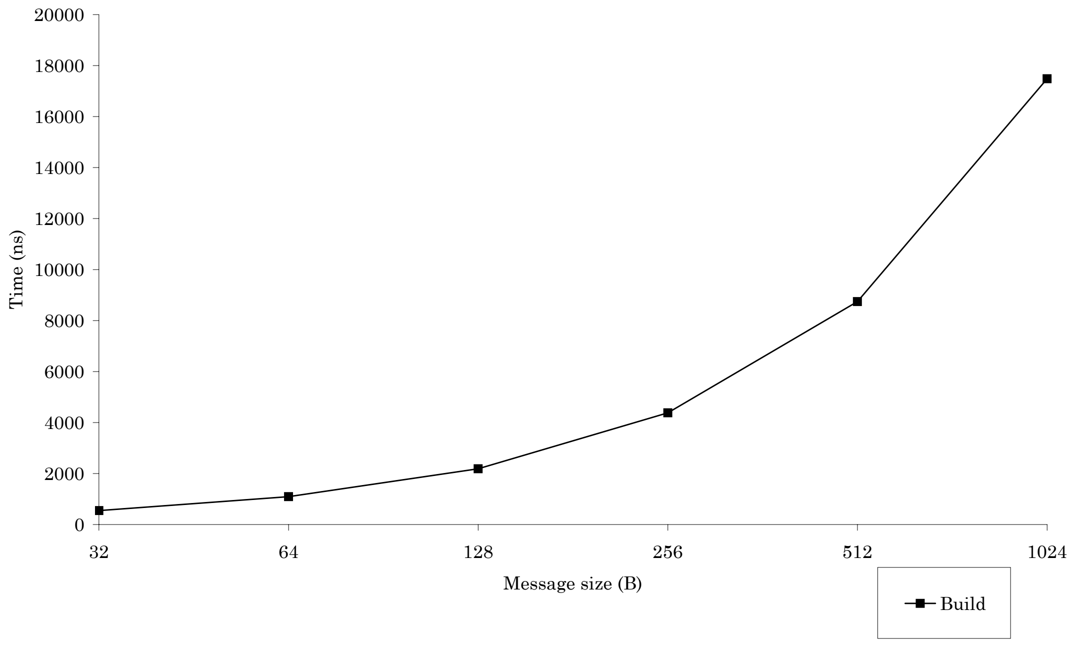
For small messages, building the message takes relative less of the total time used than for larger messages. Indeed, most of the time used for smaller messages is to actually execute *door_call*, whereas it is the other way around for larger messages – building the message is the dominant cost. This implies that smaller messages uses relative more time accessing the shared memory space than larger messages. In turn, there are more frequent accesses to the shared memory space for smaller messages than for larger. Because there is OS provided synchronization[3] in the shared memory space, only one process can access the space at a time. Hence, the more accesses to such a synchronized space, the more thrashing. For larger messages, there are less accesses because more time is spent building the message.

Queuing

Second, as for message passing over the TCP/IP stack, queuing is also probable for all message pairs. In order to simulate a real-world workload, semaphores are introduced in order to be certain that all message pairs indeed starts simultaneously. The race between the different processes is identical to the one illustrated for TCP/IP. This mechanism is illustrated in figure 5.2 and described in the corresponding section 5.1.



(a)



(b)

Figure 5.4: Cost figures for building a message or post. The cost is the average time used to build a post/message of a given size.Figure (b) is an enlarged version of figure (a).

5.3 Benchmark 3: Building or interpreting messages and log posts

This section describes the cost figures derived from running `build.c`, described in section 4.1.3. As illustrated in figure 5.4(a), message or post sizes vary between 32B and 16384KB. It is important to notice that the x-axis is logarithmic. Therefore, even though the cost at first sight seems to be increasing exponentially, it is actually increasing near linearly: Every time the message or post size is doubled, the cost associated with building the message or post is also approximately doubled.

In our previous work, we suggested that a message or post would be of size 32B. For clarity, we have included an enlarged version of figure 5.4(a). Figure 5.4(b) shows the cost associated with building messages/posts of sizes ranging from 32B to 1024B. To verify that these results are indeed accurate, consider the case when building a message or post of 32B. According to the results, this takes on average $546ns$. As explained en section 4.1.3, the benchmark uses a nested loop, filling out one byte of the message or post at a time. When examining the code for `build.c`, presented in appendix C.3, it is possible to perform an analysis of the code used to build a message or post. Each message or post built includes 1 condition + 1 value assignment + 1 pointer assignment + 32 $\times$ (1 condition + 1 value assignment + 1 pointer increment + 1 integer increment) + 1 integer increment. If we assume that this results in approximately 500 instructions of machine code, the time needed to execute this is

$$\frac{500inst}{0.93\frac{ns}{inst}} \approx 538ns$$

which complies with the benchmark results.

As mentioned in the introduction, this benchmark was also run at a Sun Fire V890 server. In comparison with the benchmark run at the T1000 server, the V890 on average uses $25\,884ns$ to build a message of size 16KB. That is one order of magnitude better than the T1000, which uses $278\,552ns$ to perform the same task. This suggests that the T1000 server indeed does not excel at tasks that do not involve parallelism.

5.4 Benchmark 4: Writing to log

This section describes the cost figures derived from running *myLog.c*, described in section 4.1.4. As illustrated in figure 5.5 the message size varies from 32B to 16KB, whereas the buffer size varies from 10 to 1280MB. Although not immediate apparent from the figure, it contains 8 curves, each representing a different buffer size. However, the time used to write a log post to buffer is clearly not much dependent on the buffer size. It is worth mentioning that all buffer sizes in the test set are larger than the L2 cache of 3MB. Writing to a buffer smaller than 3MB might therefore result in different costs.

It is important to notice that the x-axis is logarithmic. Therefore, even though the cost at first sight seems to be increasing exponentially, it is actually increasing near linearly. Every time the post size is doubled, the cost associated with writing the post to log is also approximately doubled.

This benchmark was also run at the Sun Fire V890 server. When writing a 16KB log post to a buffer of size 640MB, the V890 on average uses $29\,700ns$. That is one order of magnitude better than the T1000, which T1000 uses $279\,532ns$ to perform the same task.

[3] OS provided synchronization is controlled by the OS, and not the application programmer.

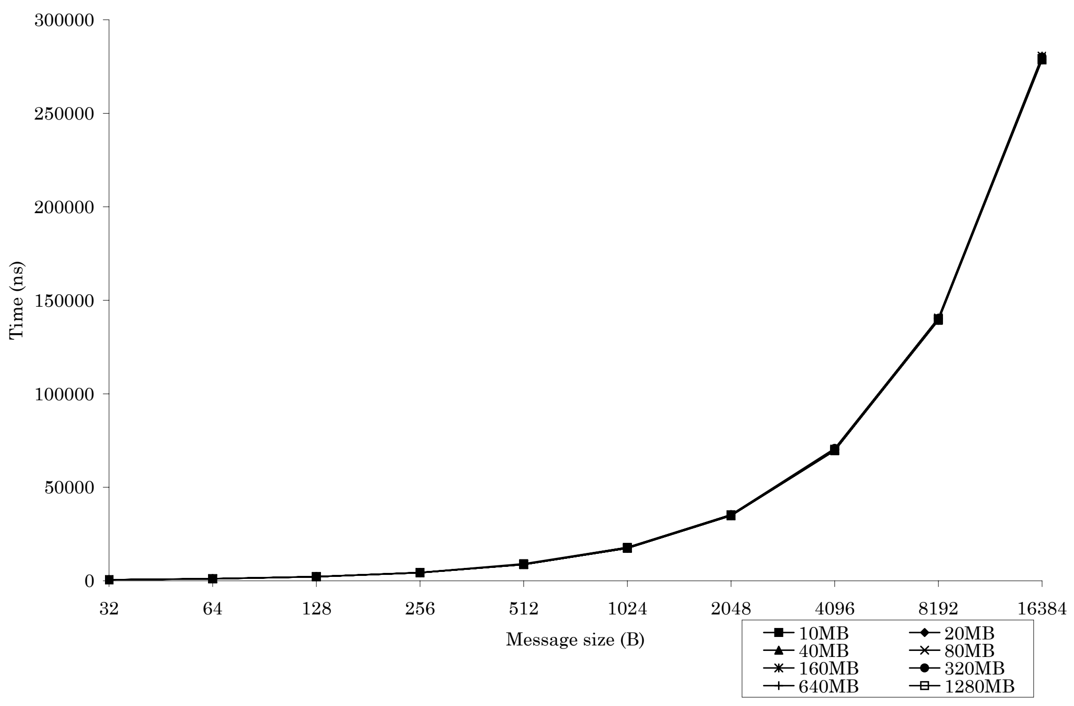


Figure 5.5: Cost figures for writing to log. Message size and buffer size are varied. The cost is the average time used to write a log post to buffer.

5.5 Benchmark 5: Synchronizing using POSIX mutexes

This section describes the cost figures achieved from running `sr_mutex.c`, described in section 4.1.5.

Figure 5.6 shows the costs associated with setting and releasing mutexes when both the number of mutexes in the mutex set, and the total number of sets and releases per mutex are varied. It is evident that compared to the costs of sending and receiving messages, mutexes are cheap both to set and release. As illustrated by the figure, setting a mutex is slightly more expensive than releasing it. This is the case for all samples, and the cost difference is fairly constant ranging 10 to 20 ns, approximately.

It is interesting to see the drop of costs when the number of mutexes in the mutex set is increased beyond one. However, increasing the number further does not result in any additional fall of costs.

As stated earlier, this micro benchmark synchronize threads within the same process. This is likely the reason to why mutexes are so cheap. Threads within the same process run at the same core and thus share L1 cache. Therefore, access to the mutex may be kept entirely in L1 cache, which has a much lower access time than the higher memory levels.

This benchmark was also run at the Sun Fire V890 server described in section 5.3. When using a mutex set of 10 mutexes and setting and releasing each mutex 10 000 000 times, the V890 on average uses 35 ns to set a mutex and $32ns$ to release a mutex. The T1000, on the other hand, uses $83ns$ to set a mutex and $71ns$ to perform the same task.

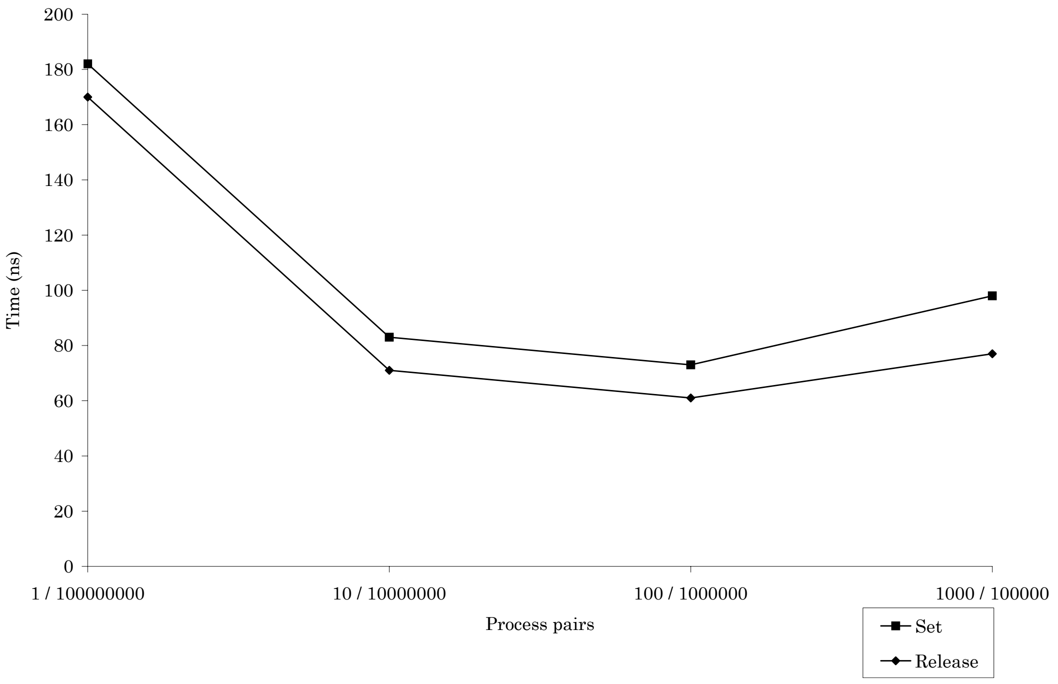


Figure 5.6: Cost figures for setting/releasing a mutex. The number of mutexes in the mutex set and the number of sets and releases are varied. The cost is the average time used to set a mutex.

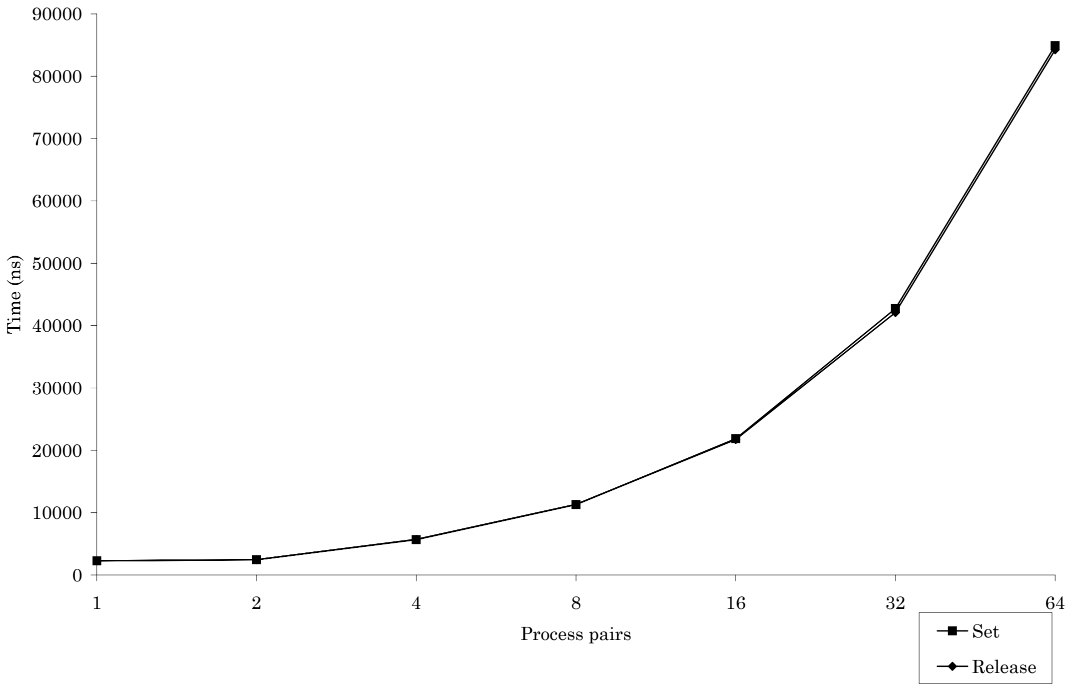


Figure 5.7: Cost figures for setting and releasing a semaphore, the number of processes are varied. The costs are the average times used to set and release a semaphore.

5.6 Benchmark 6: Synchronizing using System V semaphores

This section describes the cost figures derived from running `lseminit.c` and `lsemset.c`, described in section 4.1.6. As illustrated in figure 4.6, process pairs are varied from 2 to 64. There are two curves in the figure. One curve represents the cost associated with setting a semaphore, the other curve represents the cost associated with releasing the semaphore. It is important to notice that the scale of the x-axis is logarithmic. Therefore, even though the costs associated with setting and releasing semaphores look as if they are increasing exponentially, they are actually increasing near linearly. Except from the interval between 1 and 2 process pairs, the time used to set or release a semaphore is approximately doubled every time the number of process pairs is doubled.

Compared to Benchmark 5, it is obvious that semaphores are expensive. The average time used when setting 10 mutexes 10 000 000 times each, as described in 5.5, is $83ns$ per mutex. The average time used by a single process when setting a semaphore 1 000 000 times is $2269ns$. Semaphores are thus two orders of magnitude more expensive than mutexes. Clearly, a system that would keep the use of semaphores at an absolute minimum would be tractable.

One possible reason to why semaphores are much more expensive than mutexes, is their ability to synchronize across process boundaries. Because processes may run at different cores, semaphores must at least be as low in the memory hierarchy as the L2 cache. One might argue that in the case of one process, as in the example above, the semaphore could instead exist in L1 cache, and thus impose smaller costs. However, this is decided by the OS and is therefore not up to the application programmer.

The reason to why the costs increase with the number of concurrent processes, is that only one process at a time can update a semaphore. If several processes try to update the semaphore simultaneously, they will be performed in an arbitrary order. Processes therefore have to spend time waiting for the semaphore to be available for updates.

CHAPTER 6
COST FIGURES

In order to perform the calculation presented in chapters 7 and 8, several cost figures are needed. Having presented all results from the benchmarks in chapter 5, this chapter presents the typical cost figure for each benchmark. These figures are described in section 6.1.

Because it is not feasible to benchmark every single cost associated with a transaction, the remaining costs, derived from [BJ05], are adjusted to the Sun Fire T1000 hardware. This is presented in section 6.2.

Finally, section 6.3 lists all the costs associated with a transaction to clearly set them out.

6.1 BENCHMARKED COSTS

When the micro benchmarks described in section 4.1 were executed, several parameters were varied. In order to perform the calculations presented in the next chapters, it is necessary to decide upon a single cost for each benchmark. The following subsections present the cost figures derived for each of the most important tasks.

6.1.1 Benchmark 1: Message passing using TCP/IP sockets

In order to establish the typical message passing cost, it is necessary to examine how the transactions behave. In our previous work we assumed transactions to be small and not to involve many nodes. To assure that the complexities of the transactions are not underestimated, we assume that the CTU transaction is the most frequent transaction. We therefore anticipate the average transaction to involve four nodes.

To fully take advantage of the system, all nodes should at any time participate in at least one transaction. This results in 32 process pairs communicating simultaneously. This may not be correct in every case, but we are assured not to under estimate the number of concurrent processes communicating. In our previous work, we estimated a message to be of size 32B. The messages sent according to the 2PC protocol are indeed small, but to assure that the somewhat larger operation messages also are considered, we settle for a message size of 64B.

According to figure 5.1(a) on page 52, the time used to send or receive a message of 64B, when 32 process pairs are communicating simultaneously, is

$$32\ 457ns$$

6.1.2 Benchmark 2: Message passing using Solaris Doors

Following the deduction presented in section 6.1.1, messages are of size 64B. If all nodes are participating in at least one transaction at any time, 32 processes are communicating simultaneously.

According to the graph presented in figure 5.3(a) on page 55, it appears desirable to send messages of size 4096B rather than 64B. When looking exclusively on the time used to send or receive, this is correct. However, as discussed in section 5.2, when including the time used to build and interpret messages, this is no longer the case. Messages of 64B are therefore still desirable.

According to figure 5.3(a), the average time needed to send or receive a message of size 64B when 32 processes pairs are communicating simultaneously is

$$116\,422ns$$

It is obvious that when following this scenario, it is cheaper to communicate using TCP/IP sockets.

6.1.3 Benchmark 3: Building or interpreting messages and log posts

As already mentioned, the message size is 64B. However, it is also important to establish the size of a log post. In our previous work we suggested the size to be 32B. However, some log posts may be larger, i.e. the 2PC log posts containing all the participants of a distributed transaction. To assure that the log post size is not underestimated, we assume it to be 64B.

According to the graph presented in figure 5.4(a) on page 57, the time used to build a message or post of 64B is

$$1092ns$$

6.1.4 Benchmark 4: Writing to log

It is important for the log buffer to be sufficiently large to maintain the log while reviving a malfunctioning node, which may take several minutes. However, as illustrated by the graph in figure 5.5 on page 59, there is no noticeable difference in the cost associated with writing to buffer regardless of the buffer size. To be on the safe side, we choose the most expensive alternative, a buffer size of 640MB. The difference in cost between the cheapest and the most expensive alternatives is only 2.9%. The time used to write a 64B log post to a log buffer of size 640MB is

$$1120ns$$

6.1.5 Benchmark 5: Synchronizing using POSIX mutexes

When it comes to the cost for setting and releasing mutexes, figure 5.6 indicates that the greatest variation in cost is when going from 1 to 10 mutexes in the mutex set. As mentioned in the introduction to chapter 4, we will not use POSIX mutexes to synchronize data structures shared between different processes. Nevertheless, because mutexes are likely to be needed to synchronize within a process, the costs for setting and releasing mutexes are still included in this report.

Because we do not have an actual DBMS to our disposal, it is hard to predict the number of mutexes in an actual mutex set. However, because the variations in costs are minor when the set holds 10 or more mutexes, we assume the typical mutex set to contain 10 mutexes.

According to the graph presented in figure 5.6 on page 60, the average time used to set a mutex in this scenario is

$$83ns$$

The average time used to release a mutex is

$$71ns$$

6.1.6 Benchmark 6: Synchronizing using System V semaphores

A rough approximation of the cost associated with running the CTU transaction on the SE architecture, reveals that the tasks of setting and releasing semaphores predominates the total cost. This means that a great fraction of the time is spent setting and releasing semaphores. According to [BJ05] there are four shared data structures that need synchronization. These are

- Lock hierarchy
- File dictionary
- B-tree
- Log

Three of these data structures are synchronized using semaphores, whereas the fourth, the file dictionary, is synchronized using database locks. This is further explained in section 2.6.2 at page 11. According to [BJ05], the lock hierarchy semaphore is set and released 13 times, the b-tree semaphore is set and released 4 times, and the log semaphore is set and released 5 times during the execution of the CTU transaction. This means that a semaphore is on average set and released

$$\left\lceil \frac{13+4+5}{3} \right\rceil = 8$$

times during the CTU transaction. If all nodes are running at least one transaction at any time, and we (in order not to under estimate) assume that the average transaction is the CTU transaction, processes are bound to step on each other's toes.

If we simplify and round off the execution time, including only the time used to set or release semaphores, one third of the time is on average used to set and release one of the three semaphores. If 32 nodes are executing one transaction each at the same time, we can expect

$$\left\lceil 32 \times \frac{1}{3} \right\rceil = 11$$

processes to perform operations on a given semaphore at any given time. In order to be able to compare this with the results presented in section 5.6 on page 61, we assume that 8 processes are performing operations on a semaphore simultaneously. Because the average transaction most likely is not the CTU transaction, we do not expect this to affect the results. According to the graph presented in figure 4.6 on page 60, the time used to set a semaphore when 8 concurrent processes are performing operations on a semaphore simultaneously, is 11 295 ns. The time used to release a semaphore is 11 323 ns. The time spent waiting for the semaphore to be available is not included in these figures.

6.2 Other Costs

In addition to the costs described in the previous section, [BJ05] utilized several other basic costs to calculate the total time used to execute the benchmark transactions. As these basic costs originally were calculated for an assumed platform with a different hardware specification than the Niagara platform, they are recalculated. It is important to be aware of that these costs are not obtained by simulation or benchmarking, but are based on assumptions and may therefore be possible sources to inaccurate results.

Section 6.2.1 presents the organization of a data block before section 6.2.2 describes the costs used for both the SN and SE architectural approaches, the SN specific costs and the SE specific costs, based on the hardware described in section 3.2.1.

6.2.1 Block organization

For the reader to understand the costs associated with reading records, it is necessary to explain the block organization. This is illustrated and explained in figure 6.1. Data records are assumed to be 200B, whereas index records are assumed to be 20B. All blocks are assumed to be of size 8KB (8192B). This gives

$$\left\lfloor \frac{8192B}{200B} \right\rfloor = 40$$

data records per block and

$$\left\lfloor \frac{8192B}{20B} \right\rfloor = 409$$

index records per block.

An index vector is located at the start of each block, (this approach is also taken by System R and is described at page 120 in [ABC+76]). Each entry in the index vector contains a pointer to a record in the block. All records in the block are associated with a pointer in the index vector. Pointers are listed in the order of the primary key of the record to which they point, but there is no way of knowing the value of the key without following the pointer. The organization of the block is illustrated in figure 6.1. The block illustrated is an index block, but a data block is organized in the same way. The only difference is that the data block has fewer records, because data records are larger than index records. The number of entries in the index vector in a data block is therefore also smaller.

The index vectors of the index blocks need to be able to point to 409 different index records. We assume that the organization of records within a block is strict, and that if a pointer points to the number of a record within the block, i.e. record number 277, an offset from the start of the block is calculated. Therefore only logical pointers within a block are needed.

To uniquely identify 409 different index records, minimum 9 bits are needed per record. It is perhaps more common to operate with bit size in a power of 2. That is, even though only 9 bits per record are needed to identify 409 index records, one would allocate 16 bits instead of 9. However, we have chosen to operate with a minimum size. This is equal for both architectures, and hence it does not have significant impact on the relative difference between them.

The total size of the index vector in an index block is:

$$\left\lceil \frac{409 \times 9bits}{8} \right\rceil = 461B$$

The same assumptions are made for the data blocks. To uniquely identify 40 data records, minimum 6 bits are needed. This means that the total size of the index vector in a data block

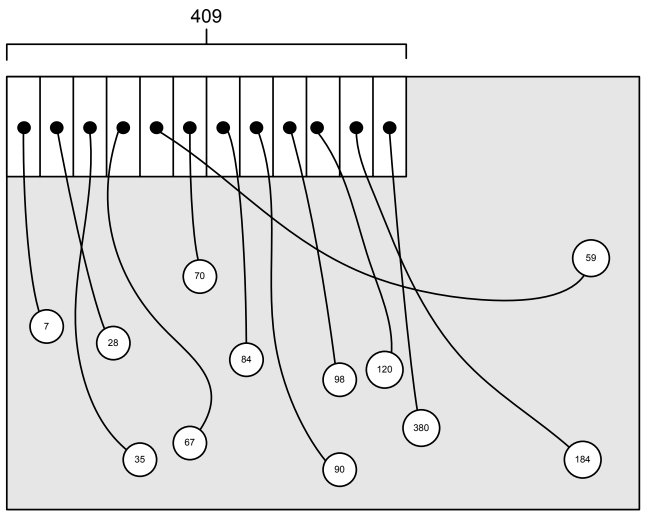


Figure 6.1: Block organization. Each block starts with an index vector. All records in the block has an entry in the index vector containing a pointer to the record. The pointers in the index vector are ordered by the keys of the records to which they point. A binary search for the record is then performed. For every pointer that is examined, it is necessary to follow the pointer and examine the record's key, because the pointer does not contain that information.

is:

$$\left\lceil \frac{40 \times 6bits}{8} \right\rceil = 30B$$

6.2.2 General costs

This section presents costs associated with both the SN and SE approach.

Write a cache word to / read a cache word from main memory

Because cache words between main memory and L2 cache are 64B, and only one cache word can be read or written per access, writing or reading 64B takes (see section 3.2.1 at page 24)

$$100ns$$

The consequence of several nodes trying to access the main memory simultaneously is not taken into consideration.

Write a cache word to / read a cache word from L2 cache

As stated in section 3.2.1 the latency for accessing the L2 cache is 23 cycles, or 21 ns. As for the main memory, a single cache word may be read or written per access. For data, the cache word size between the L2 and L1 cache is 16B. Writing or reading 16B thus takes

$$21ns$$

The consequence of several nodes trying to access the L2 cache simultaneously is not taken into consideration.

Write to / read from L1 cache

The access time for L1 cache is almost negligible, as the L1 caches reside very close to the processing unit. As explained in chapter 3.2.1, the execution of a single instruction takes

$$\frac{1}{1024^3}ns \approx 0.93ns$$

Reading or writing a cache word (16B) from L1 cache therefore takes

$$0.93ns \approx 1ns$$

The consequence of several nodes trying to access the L1 cache simultaneously is not taken into consideration.

Create transaction context

All nodes that are involved in the execution of a transaction, both NOC and NOS(es) in the SN case, and the node performing the transaction in the SE case, need to create a context for the transaction. The transaction context includes information needed to perform logging. The research report [HT93] estimates that 200 instructions are needed to create a transaction context at a single node. Executing 200 instructions takes:

$$0.93\frac{ns}{inst} \times 200inst = 186ns$$

Delete transaction context

When a transaction commits or aborts, the transaction context must be deleted in order to leave the DBMS in a consistent state. According to [HT93], this is estimated to take 200 instructions. Executing 200 instructions takes:

$$0.93\frac{ns}{inst} \times 200inst = 186ns$$

Check transaction context

It is sometimes necessary to check for the existence of a transaction context. This may be before writing a log post produced by a transaction. In [HT93], checking for the existence of a transaction context is estimated to take 100 instructions. Executing 100 instructions takes:

$$0.93\frac{ns}{inst} \times 100inst = 93ns$$

Read an index record from main memory

Reading an index record from main memory involves reading the index vector of the block in which the record reside, performing a binary search on the index vector and reading the record. Reading the index vector from main memory to L2 cache requires

$$\left\lceil \frac{461B}{64B/access} \right\rceil = 8$$

accesses to main memory. Reading the index vector from L2 cache requires

$$\left\lceil \frac{461B}{16B/access} \right\rceil = 29$$

accesses. Reading the index vector from L1 cache also requires

$$\left\lceil \frac{461B}{16B/access} \right\rceil = 29$$

accesses. The number of keys that need to be examined when performing a binary search on the index vector is in the worst case

$$\lceil \log_2 409 \rceil = 9$$

For each of the 9 keys examined during the binary search, the main memory, the L2 cache and the L1 cache must be accessed. Because index records are assumed to be of size 20 bytes, the whole record is contained within the 64B cache word when fetching from memory to L2 cache. Fetching the index record from L2 cache and L1 cache, on the other hand, requires 2 accesses ($\frac{20B}{16B/access} > 1acess$) However, we assume that the key is included in the first cache line, and thus only one access is needed during the binary search. When the correct record is found, one additional access is needed to the L2 and L1 cache in order to fetch the remainder of the record.

Reading an index record from main memory then takes:

$$\begin{aligned}
&\overbrace{(8 \times 100ns) + (29 \times 21ns) + (29 \times 0.93ns)}^{\text{read vector}} \\
+&\overbrace{(9 \times 100ns) + (9 \times 21ns) + (9 \times 0.93ns)}^{\text{binary search}} \\
+&\overbrace{21ns + 0.93ns}^{\text{fetch record}} \\
\approx&\underline{2555ns}
\end{aligned}$$

Read an index record from L2 cache

Reading an index record from L2 cache is very similar to reading it from main memory. The only difference is that the accesses to main memory are not included. Reading an index record from L2 cache then takes:

$$\overbrace{(29 \times 21ns) + (29 \times 0.93ns)}^{\text{read vector}} + \underbrace{(9 \times 21ns) + (9 \times 0.93ns)}_{\text{binary search}} + \overbrace{21ns + 0.93ns}^{\text{fetch record}} \approx \underline{855ns}$$

Read an index record from L1 cache

Reading an index record from L1 cache is very much like reading an index record from L2 cache. Even though the index vector of the b-tree root has fewer entries than an index vector for an ordinary index record, we assume that searching for the correct index record is performed in the same manner, that 9 keys need to be examined during the binary search. This cost figure will therefore be used also when reading an index record from the root. Reading the index vector requires

$$\left\lceil \frac{461B}{16B/access} \right\rceil = 29$$

accesses to the L1 cache. For each of the keys that need to be examined, one additional access is needed. Finally, one last access is made to the L1 cache in order to fetch the remainder of the record. Thus, reading and index record from L1 cache takes:

$$\overbrace{(29 \times 0.93ns)}^{\text{read vector}} + \underbrace{(9 \times 0.93ns)}_{\text{binary search}} + \overbrace{0.93ns}^{\text{fetch record}} \approx \underline{36ns}$$

Read a data record from main memory

Reading a data record from main memory is very similar to reading an index record. The only difference is the number of accesses needed. Because the index vectors for the data records are smaller than in the case of the index records, the number of accesses to the main memory is only

$$\left\lceil \frac{30B}{64B/access} \right\rceil = 1$$

The number of accesses to the L2 cache is

$$\left\lceil \frac{30B}{16B/access} \right\rceil = 2$$

and the number of accesses to the L1 cache is also

$$\left\lceil \frac{30B}{16B/access} \right\rceil = 2$$

in order to fetch the index vector. The number of accesses to the main memory, the L2 cache and the L1 cache during the binary search is in the worst case

$$\lceil \log_2 40 \rceil = 6$$

We assume that the entire key to be examined resides within the cache word that is fetched during the binary search. In this case, the data record is too big to be contained within a cache word when the last key is examined, therefore the remainder of the record needs to be read in addition. The number of accesses to the main memory to do this is

$$\left\lceil \frac{200B}{64B/access} \right\rceil - 1 = 3$$

The number of accesses needed to the L2 cache is

$$\left\lceil \frac{200B}{16B/access} \right\rceil - 1 = 12$$

The number of accesses needed to the L1 cache is also

$$\left\lceil \frac{200B}{16B/access} \right\rceil - 1 = 12$$

Reading a data record from main memory then takes:

$$\begin{aligned}
&\overbrace{(1 \times 100ns) + (2 \times 21ns) + (2 \times 0.93ns)}^{\text{read vector}} \\
+&\overbrace{(6 \times 100ns) + (6 \times 21ns) + (6 \times 0.93ns)}^{\text{binary search}} \\
+&\overbrace{(3 \times 100ns) + (12 \times 21ns) + (12 \times 0.93ns)}^{\text{fetch record}} \\
\approx&\underline{1439ns}
\end{aligned}$$

Read a data record from L2 cache

As for index records, reading a data record from L2 cache is very similar to reading it from main memory. The only difference is that the accesses to main memory are not included. Reading a data record from L2 cache then takes:

$$\overbrace{(2 \times 21ns) + (2 \times 0.93ns)}^{\text{read vector}} + \underbrace{(6 \times 21ns) + (6 \times 0.93ns)}_{\text{binary search}} + \overbrace{(12 \times 21ns) + (12 \times 0.93ns)}^{\text{fetch record}} \approx \underline{439ns}$$

Read a data record from L1 cache

Due to the extremely small size of the L1 cache compared to the database size, we find the probability of a data record residing in L1 cache to be negligible (opposed to reading an index record from L1 cache, in section 6.2.2).

Lock a resource

Locking a data resource is in [HT93] estimated to take about 200 instructions. We will use this estimate. Executing 200 instructions takes:

$$0.93\frac{ns}{inst} \times 200inst = 186ns$$

The time used to synchronize the lock hierarchy in the SE case is not included.

Unlock a resource

We assume that unlocking a resource costs the same as locking a resource. We will therefore estimate that 200 instructions are needed to unlock a resource. Executing 200 instructions takes:

$$0.93\frac{ns}{inst} \times 200inst = 186ns$$

The time used to synchronize the lock hierarchy in the SE case is not included.

6.2.3 SN specific costs

This section presents costs specific for the SN approach.

Decide which nodes that are slaves

Deciding which nodes that are to be slaves involves accessing the dictionary and performing a hash for each of the participants. In the transactions considered in our research, one or four slaves are used, depending on the transaction complexity. The research presented in [HT93] gives an estimate of how many instructions that are needed to do this. To decide a single slave, 150 instructions are estimated. Executing 150 instructions takes:

$$0.93\frac{ns}{inst} \times 150inst \approx 140ns$$

6.2.4 SE specific costs

This section presents costs specific for the SE approach.

Access dictionary

This operation is only performed as a standalone task in SE. For SN, dictionary access is included in the cost of deciding which nodes that are to be slaves in the transaction execution. The report [HT93] suggests that the cost measured to decide which nodes that are to be slaves takes 150 instructions. Under the assumption that the additional hash function that is applied for each participant is cheap, we treat the cost for dictionary access as equal to the cost in section 6.2.3. Accessing the dictionary then takes:

$$0.93\frac{ns}{inst} \times 150inst \approx 140ns$$

The time used to synchronize the dictionary in the SE case is not included.

6.3 Cost summary

All costs that will be used when performing the calculations in chapters 7 and 8 are summarized in table 6.1.

[1]Regarding the field *Send a message* in table 6.1, TCP/IP sockets are used to realize message passing because this is the cheaper alternative given the message size and the number of concurrent processes

Description	**Cost**
Read a cache word (64B) from memory	100 ns
Write a cache word (64B) to memory	100 ns
Read a cache word (16B) from L2 cache	21 ns
Write a cache word (16B) to L2 cache	21 ns
Read a cache word (16B) from L1 cache	0.93 ns
Write a cache word (16B) to L1 cache	0.93 ns
Create transaction context	186 ns
Delete transaction context	186 ns
Check transaction context	93 ns
Read an index record from main memory	2555 ns
Read an index record from L2 cache	855 ns
Read an index record from L1 cache	36 ns
Read a data record from main memory	1439 ns
Read a data record from L2 cache	439 ns
Lock a resource	186 ns
Unlock a resource	186 ns
Build log post	1092 ns
Write to log	1120 ns
Build a message	1092 ns
Send a message[1]	32 457 ns
Receive a message	32 457 ns
Decide which nodes that are slaves	140 ns
Interpret a message	1092 ns
Access dictionary	140 ns
Set mutex	83 ns
Release mutex	71 ns
Set semaphore	11 295 ns
Release semaphore	11 323 ns

Table 6.1: Cost figures

CHAPTER 7
SN WORKLOAD ESTIMATE

This chapter investigates the total cost of performing the four different transaction types defined in section 2.2 on page 6 in an SN context.

Albeit the structure of this chapter is indeed similar to an analog chapter in our previous work, the costs of performing the transactions are recalculated, as all the cost figures are different. It is also necessary to be familiar with the transactions' sub tasks to make the most of the analysis presented in chapter 9.

Figure 7.1 illustrates the execution path for a transaction, with the critical path drawn[1] with bold lines. The transaction is decomposed into groups of operations[2], called tasks. This figure is valid for all the transactions defined in section 2.1, although the contents of each task may differ.

This chapter examines each task for each transaction. Each task should be seen in conjunction with the 2PC presumed commit protocol explained in section 2.4.2 at page 8.

A NOC and a NOS may very well be on the same node, but this report only consider the worst-case scenario, when the NOC and NOSes reside on different nodes. For the two complex transactions CTR and CTU, we assume that the four tuples reside at different NOSes.

All cost estimates used in this chapter are explained in detail in chapter 6, and summarized in table 6.1.

The next section presents a calculation of the height of the database b-tree, and the probability of a cache hit when fetching a record. The following sections in turn presents the costs associated with each transaction type. Corresponding calculations for the SE DBMS architecture are presented in chapter 8.

7.1 CALCULATING THE B-TREE HEIGHT FOR SN

This section calculates the b-tree height in the SE case. In our previous work, we assumed the database to be of size 10GB, and that the entire database is kept in main memory. However, as explained in section 3.2.1, the Sun Fire T1000 Server is only equipped with 8GB of main memory. Nevertheless, we continue to assume a database size of 10GB, as the T1000 Server may be extended to have 16GB of memory, which is more than necessary for the types of databases we examine.

In the SN case, the main memory area is split into subsets of $\frac{1}{32}$ size. We assume that data is partitioned using an approximately perfect hash function and therefore is distributed evenly among all nodes. Although the usual approach is to have a b-tree for every table, we assume that each node operates on a single b-tree containing the entire subset of the database.

[1]The critical path is the sum of all operations needed to be performed before a response can be given back to the transaction service requester.

[2]The term operation as used here should not be mixed up with the term operation as used for an update or read operation earlier in this report. Although this is still valid, the term expands to contain operations such as creating a transaction context, setting a lock etc.

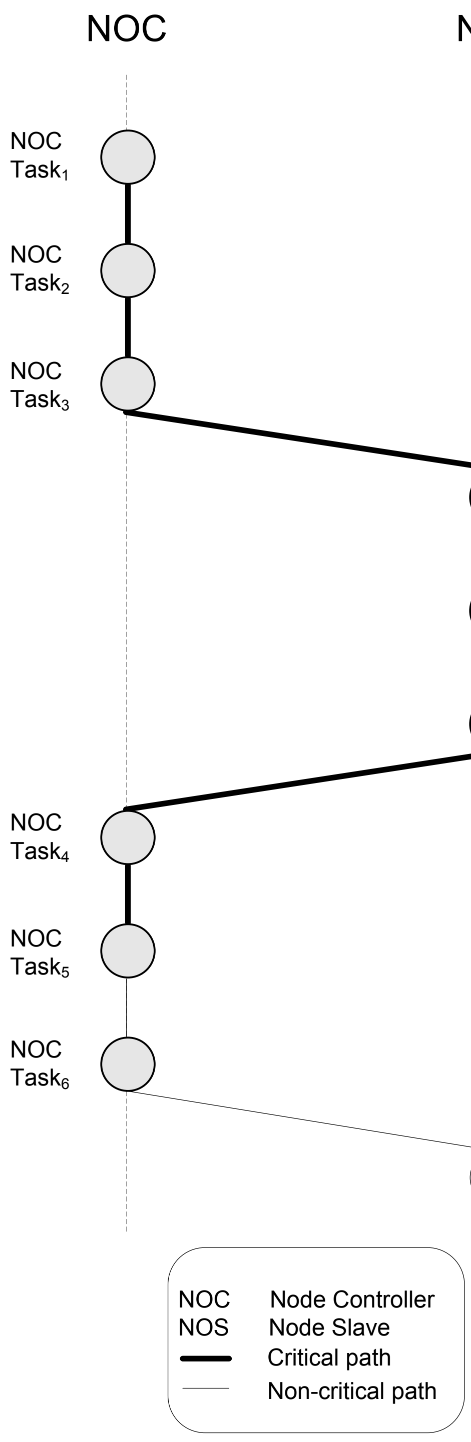


Figure 7.1: The execution path in a SN architecture. The figure for simplicity shows the scenario where only one NOS is involved. For the CTU and CTR transactions, however, up to four NOSes could be involved.

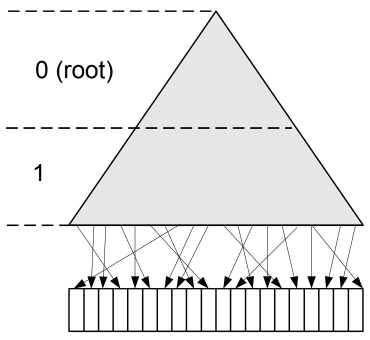


Figure 7.2: The SN b-tree. The b-tree consists of two levels of index records and one level of data records. The height of the b-tree is 3.

The number of data blocks per node is:

$$\left\lceil \frac{10GB}{32 \times 8KB} \right\rceil = \left\lceil \frac{10737418240B}{32 \times 8192B} \right\rceil = 40960$$

The number of index blocks on the first level is:

$$\left\lceil \frac{40960}{409} \right\rceil = 101$$

The number of index blocks on the second level is:

$$\left\lceil \frac{101}{409} \right\rceil = 1$$

This results in a b-tree of height 3, with two levels of index records, and one level of data records. This is illustrated in figure 7.2. The entire b-tree consists of

$$(1 + 101 + 40960) \times 32 = 1\ 313\ 984$$

blocks. The total size of the index pages per node is:

$$(1 + 101) \times 8KB = 839KB$$

As described in section 3.2.1, the L2 cache size is 3MB. This results in a L2 cache size of

$$\frac{3MB}{32} = 96KB$$

per node. The L1 data cache size, on the other hand is only 16KB per core. This results in a L1 cache size of

$$\frac{16KB}{4} = 4KB$$

per node. Because the L1 cache size per node indeed is small, we assume that only index records that reside in the b-tree root will have cache hits. We assume that no other index records or data records will have a hit in the L1 cache. This means that

$$\frac{96KB}{839KB} \approx 11.4\%$$

of the index blocks at the second level are available from L2 cache. The remaining 88.6% must be fetched from main memory.

7.2 The Simple Transaction Update (STU)

This section gives the estimated time used to execute the STU transaction. The STU transaction contains a single update and may therefore involve up to two nodes, The transaction is decomposed into groups of operations. The following is a description of these operations together with a cost estimate measured in time for each task. The relationships between tasks are illustrated in figure 7.1.

7.2.1 Node Controller (NOC)

This section gives the estimated workload for the part of the transaction executed at the node running the node controller.

NOC Task$_1$

This task receives and interprets the transaction. It also creates a transaction context. The costs are described in table 7.1.

Description	**Cost** (ns)
Read transaction from main memory:	100
Interpret transaction (message):	1092
Create transaction context:	186

Table 7.1: NOC Task$_1$ cost description for STU in SN

This gives:

$$100ns + 1092ns + 186ns = \underline{\underline{1378ns}}$$

This task is in the critical path.

NOC Task$_2$

This task decides which node slave participates in the execution of the transaction, and writes the result to log. In order to preserve consistency, the log is written to main memory. The costs are described in table 7.2.

Description	**Cost** (ns)
Decide participant:	140
Check transaction context:	93
Build start log post:	1092
Write start log post:	1120

Table 7.2: NOC Task$_2$ cost description for STU in SN

This gives:

$$140ns + 93ns + 1092ns + 1120ns = \underline{\underline{2445ns}}$$

This task is in the critical path.

NOC Task$_3$

This task builds the message containing which update that is to be done, and sends it to the slave. The message has a *prepare*-message piggybacked. The costs are described in table 7.3.

Description	**Cost** (ns)
Build operation message:	1092
Send operation message:	32 457

Table 7.3: NOC Task$_3$ cost description for STU in SN

This gives:

$$1092ns + 32\,457ns = \underline{\underline{33\,549ns}}$$

This task is in the critical path.

NOC Task$_4$

This task receives a *ready*-message from the slave when the slave has performed the update. It interprets the message and logs the commit. The log is written to main memory. The costs are described in table 7.4.

Description	**Cost** (ns)
Receive *ready*-message:	32 457
Interpret *ready*-message:	1092
Check transaction context:	93
Build commit log post:	1092
Log commit:	1120

Table 7.4: NOC Task$_4$ cost description for STU in SN

This gives:

$$32\,457ns + 1092ns + 93ns + 1092ns + 1120ns = \underline{\underline{35\,854ns}}$$

This task is in the critical path.

NOC Task$_5$

This task builds an *OK*-message and sends this to the calling application. This means that the message is written to main memory. The costs are described in table 7.5.

This gives:

Description	**Cost** (ns)
Build *OK* message:	1092
Write *OK* messages to main memory:	100

Table 7.5: NOC Task$_5$ cost description for STU in SN

$$1092ns + 100ns = \underline{\underline{1192ns}}$$

This task is in the critical path.

NOC Task$_6$

This task builds a *commit*-message and send it to the node slave. It also deletes the transaction context. The costs are described in table 7.6.

Description	**Cost** (ns)
Build *commit* message:	1092
Send *commit* message:	32 457
Delete transaction context:	186

Table 7.6: NOC Task$_6$ cost description for STU in SN

This gives:

$$1092ns + 32\,457ns + 186ns = \underline{\underline{33\,735ns}}$$

This task is <u>not</u> in the critical path, because it is not a part of the sum of all operations needed to be performed before a response can be given back to the transaction service requester.

Total costs

The total costs for the NOC when the STU transaction is executed is:

$$NOC\ Task_1 + NOC\ Task_2 + NOC\ Task_3 + NOC\ Task_4 + NOC\ Task_5 + NOC\ Task_6 =$$

$$1378ns + 2445ns + 33\,549ns + 35\,854ns + 1192ns + 33\,735ns = \underline{\underline{108\,153ns}}$$

7.2.2 Node Slave (NOS)

This section gives the estimated workload for the part of the transaction executed at the node running as the NOS.

Description	**Cost** (ns)
Receive operation message:	32 457
Interpret operation message:	1092
Create transaction context:	186

Table 7.7: NOS Task$_1$ cost description for STU in SN

NOS Task$_1$

This task receives and interprets the message containing which update that is to be done. It also creates a transaction context. The costs are described in table 7.7.

This gives:

$$32\,457ns + 1092ns + 186ns = \underline{\underline{33\,735ns}}$$

at each node. This task is in the critical path.

NOS Task$_2$

This task locks the data record and performs the update. It also logs the update and a prepare-post according to the 2PC protocol described in section 2.4.2. The log is written to main memory. The costs are described in table 7.8.

Description	**Cost** (ns)
Lock data record:	186
Read index record from index root in L1 cache:	36
Read first level index record:	$11.4\% \times 855 + 88.6\% \times 2555 \approx 2361$
Read data record (including locating the record):	1439
Write data record:	$\lceil \frac{200B}{64B/access} \rceil \times 21ns/access = 84$
Check transaction context:	93
Build update log post:	1092
Build prepare log post:	1092
Log update:	1120
Log prepare:	1120

Table 7.8: NOS Task$_2$ cost description for STU in SN

This gives:

$$186ns + 36ns + 2361ns + 1439ns + 84ns + 93ns + 1092ns + 1092ns + 1120ns + 1120ns = \underline{\underline{8623ns}}$$

This task is in the critical path.

NOS Task$_3$

This task creates and sends the *ready*-message according to the 2PC protocol. The costs are described in table 7.9.

Description	**Cost** (ns)
Build *ready*-message:	1092
Send *ready*-message:	32 457

Table 7.9: NOS Task$_3$ cost description for STU in SN

This gives:

$$1092ns + 32\,457ns = \underline{\underline{33\,549ns}}$$

This task is in the critical path.

NOS Task$_4$

This task receives and interprets the *commit*-message from the node controller and logs the commit. The log is written to main memory. It also unlocks the data record and deletes the transaction context. The costs are described in table 7.10.

Description	**Cost** (ns)
Receive *commit*-message:	32 457
Interpret *commit*-message:	1092
Check transaction context:	93
Build commit log post:	1092
Log commit post:	1120
Unlock data record:	186
Delete transaction context:	186

Table 7.10: NOS Task$_4$ cost description for STU in SN

$$32\,457ns + 1092ns + 93ns + 1092ns + 1120ns + 186ns + 186ns = \underline{\underline{36\,226ns}}$$

This task is <u>not</u> in the critical path, because it is not a part of the sum of all operations needed to be performed before a response can be given back to the transaction service requester.

Total costs

The total cost for the NOS when the STU transaction is executed is:

$$NOS\ Task_1 + NOS\ Task_2 + NOS\ Task_3 + NOS\ Task_4 =$$

$$33\,735ns + 8623ns + 33\,549ns + 36\,226ns = \underline{\underline{112\,133ns}}$$

7.3 THE SIMPLE TRANSACTION READ (STR)

This section gives the estimated time used to execute the STR transaction. The STR transaction contains a single read, and will therefore, as in the case of the STU, in the worst-case only involve two nodes. The transaction is decomposed into groups of operations. The following is a description of these operations together with a cost estimate measured in time for each task. Our assumed DBMS does not log read-only transactions, therefore the tasks NOC Task$_8$ and NOS Task $_4$ are not executed. The relationships between tasks are illustrated in figure 7.1.

7.3.1 Node Controller (NOC)

This section gives the estimated workload for the part of the transaction performed at the node running the node controller.

NOC Task$_1$

This task receives and interprets the transaction. It also creates a transaction context. The costs are described in table 7.11.

Description	**Cost** (ns)
Read transaction from main memory:	100
Interpret transaction:	1092
Create transaction context:	186

Table 7.11: NOC Task$_1$ cost description for STR in SN

This gives:

$$100ns + 1092ns + 186ns = \underline{\underline{1378ns}}$$

This task is in the critical path.

NOC Task$_2$

This task decides which node to participate in the execution of the transaction. The costs are described in table 7.12.

Description	**Cost** (ns)
Decide participants:	140

Table 7.12: NOC Task$_2$ cost description for STR in SN

This gives:

$$\underline{\underline{140ns}}$$

This task is in the critical path.

NOC Task$_3$

This task builds the message containing which record to read, and sends it to the node slave. The message has a *prepare*-message piggybacked. The costs are described in table 7.13.

Description	**Cost** (ns)
Build operation message:	1092
Send operation message:	32 457

Table 7.13: NOC Task$_3$ cost description for STR in SN

This gives:

$$1092ns + 32\,457ns = \underline{\underline{33\,549ns}}$$

This task is in the critical path.

NOC Task$_4$

This task receives a *read*-message from the node slave when it has performed the read. It also interprets the message. The costs are described in table 7.14.

Description	**Cost** (ns)
Receive *read*-message:	32 457
Interpret *read*-message:	1092

Table 7.14: NOC Task$_4$ cost description for STR in SN

This gives:

$$32\,457ns + 1092ns = \underline{\underline{33\,549ns}}$$

This task is in the critical path.

NOC Task$_5$

This task builds a data message containing the data read, and sends this to the calling application. This means that the message is written to main memory. The task also deletes the transaction context. The costs are described in table 7.15.

Description	**Cost** (ns)
Build *data* message:	1092
Write *data* message to main memory:	100
Delete transaction context:	186

Table 7.15: NOC Task$_5$ cost description for STR in SN

This gives:

$$1092ns + 100ns + 186ns = \underline{\underline{1378ns}}$$

This task is in the critical path.

NOC Task$_6$

This task is not performed when the transaction is a read-only transaction. This is because no logging is performed, and therefore the node performing a read, is not needed to receive a *commit*-message. If the transaction performed contained one or more updates, *commit*-messages would be sent to all nodes performing these updates.

Total costs

The total costs for the NOC when the STR transaction is executed is:

$$NOC\ Task_1 + NOC\ Task_2 + NOC\ Task_3 + NOC\ Task_4 + NOC\ Task_5 =$$

$$1378ns + 140ns + 33\ 549ns + 33\ 549ns + 1378ns = \underline{\underline{69\ 994ns}}$$

7.3.2 Node Slave (NOS)

This section gives the estimated workload for the part of the transaction performed at the node running as the node slave.

NOS Task$_1$

This task receives and interprets the message containing which record to read. It also creates a transaction context if one does not already exist. The costs are described in table 7.16.

Description	**Cost** (ns)
Receive operation message:	32 457
Interpret operation message:	1092
Create transaction context:	186

Table 7.16: NOS Task$_1$ cost description for STR in SN

This gives:

$$32\ 457ns + 1092ns + 186ns = \underline{\underline{33\ 735ns}}$$

at each node. This task is in the critical path.

Description	**Cost** (ns)
Lock data record:	186
Read index record from index root in L1 cache:	36
Read first level index record:	$11.4\% \times 855 + 88.6\% \times 2555 \approx 2361$
Read data record (including locating the record):	1439

Table 7.17: NOS Task$_2$ cost description for STR in SN

NOS Task$_2$

This task locks the data record and performs the read. The costs are described in table 7.17.

This gives:

$$186ns + 36ns + 2361ns + 1439ns = \underline{\underline{4022ns}}$$

This task is in the critical path.

NOS Task$_3$

This task creates and sends the *read*-message according to the 2PC protocol. It also unlocks the data record and deletes the transaction context. The costs are described in table 7.18.

Description	**Cost** (ns)
Build *read*-message:	1092
Send *read*-message:	32 457
Unlock data record:	186
Delete transaction context:	186

Table 7.18: NOS Task$_3$ cost description for STR in SN

This gives:

$$1092ns + 32\,457ns + 186ns + 186ns = \underline{\underline{33\,921ns}}$$

This task is in the critical path.

NOS Task$_4$

For the same reasons as the reasons described in NOC Task$_6$, this task is not performed.

Total costs

The total costs for the NOS when the STR transaction is executed is:

$$NOS\ Task_1 + NOS\ Task_2 + NOS\ Task_3 =$$

$$33\,735ns + 4022ns + 33\,921ns = \underline{\underline{71\,678ns}}$$

7.4 The Complex Transaction Update (CTU)

This section gives the estimated time used to execute the CTU transaction. The CTU transaction contains four updates. We consider the case when the records to be updated reside at four different nodes. The workload estimated in this section is therefore based on the use of one NOC and four NOSes. The transaction is decomposed into groups of operations called tasks. The following is a description of these operations together with a time estimate of each. The relationships between tasks are illustrated in figure 7.1.

7.4.1 Node Controller (NOC)

This section gives the estimated workload for the part of the transaction performed at the node running the node controller.

NOC Task$_1$

This task receives and interprets the transaction. It also creates a transaction context. The costs are described in table 7.19.

Description	**Cost** (ns)
Read transaction from main memory:	100
Interpret transaction:	1092
Create transaction context:	186

Table 7.19: NOC Task$_1$ cost description for CTU in SN

This gives:

$$100ns + 1092ns + 186ns = \underline{\underline{1378ns}}$$

This task is in the critical path.

NOC Task$_2$

This task decides which node slaves that are to participate in the transaction, and writes this to log. The log is written to main memory. The costs are described in table 7.20.

Description	**Cost** (ns)
Decide participants:	$4 \times 140 = 560$
Check transaction context:	93
Build start log post:	1092
Write start log post:	1120

Table 7.20: NOC Task$_2$ cost description for CTU in SN

This gives:

$$560ns + 93ns + 1092ns + 1120ns = \underline{\underline{2865ns}}$$

This task is in the critical path.

NOC Task$_3$

This task builds the messages containing which updates that are to be done, and sends these. These messages have a *prepare*-message piggybacked. There are four of these messages, one for each NOS. The costs are described in table 7.21.

Description	**Cost** (ns)
Build operation messages:	4 × 1092 = 4368
Send operation messages:	4 × 32 457 = 129 828

Table 7.21: NOC Task$_3$ cost description for CTU in SN

This gives:

$$4368ns + 129\ 828ns = \underline{\underline{134\ 196ns}}$$

This task is in the critical path.

NOC Task$_4$

This task receives *ready*-messages from the node slaves, when they have performed the update. It interprets the messages and logs the commit. The log is written to main memory. There are four of these messages. The costs are described in table 7.22.

Description	**Cost** (ns)
Receive *ready*-messages:	4 × 32 457 = 129 828
Interpret *ready*-messages:	4 × 1092 = 4368
Check transaction context:	93
Build commit log post:	1092
Log commit:	1120

Table 7.22: NOC Task$_4$ cost description for CTU in SN

This gives:

$$129\ 828ns + 4368ns + 93ns + 1092ns + 1120ns = \underline{\underline{136\ 501ns}}$$

This task is in the critical path.

Description	**Cost** (ns)
Build *OK* message:	1092
Write *OK* message to main memory:	100

Table 7.23: NOC Task$_5$ cost description for CTU in SN

NOC Task$_5$

This task builds an *OK*-message and sends this to the calling application. This means that the message is written to main memory. The costs are described in table 7.23.

This gives:

$$1092ns + 100ns = \underline{\underline{1192ns}}$$

This task is in the critical path.

NOC Task$_6$

This task builds *commit*-messages and send these to all participants. It also deletes the transaction context. The costs are described in table 7.24.

Description	**Cost** (ns)
Build *commit* messages:	$4 \times 1092 = 4368$
Send *commit* messages:	$4 \times 32\,457 = 129\,828$
Delete transaction context:	186

Table 7.24: NOC Task$_6$ cost description for CTU in SN

This gives:

$$4368ns + 129\,828ns + 186ns = \underline{\underline{134\,382ns}}$$

This task is <u>not</u> in the critical path, because it is not a part of the sum of all operations needed to be performed before a response can be given back to the transaction service requester.

Total costs

The total costs for the NOC when the CTU transaction is executed is:

$$NOC\ Task_1 + NOC\ Task_2 + NOC\ Task_3 + NOC\ Task_4 + NOC\ Task_5 + NOC\ Task_6 =$$

$$1378ns + 2865ns + 134\,196ns + 136\,501ns + 1192ns + 134\,382ns = \underline{\underline{410\,514ns}}$$

7.4.2 Node Slave (NOS)

This section will give the estimated workload for the part of the transaction performed at one of the four nodes running as node slaves.

NOS Task$_1$

This task receives and interprets the message containing which update that is to be done. It also creates a transaction context. The costs are described in table 7.25.

Description	**Cost** (ns)
Receive operation message:	32 457
Interpret operation messages:	1092
Create transaction context:	186

Table 7.25: NOS Task$_1$ cost description for CTU in SN

This gives:

$$32\,457ns + 1092ns + 186ns = \underline{\underline{33\,735ns}}$$

at each node.

This task is in the critical path.

NOS Task$_2$

This task locks the data record and performs the update. It also logs the update and a prepare-post according to the 2PC protocol described in section 2.4.2. The log is written to main memory. The costs are described in table 7.26.

Description	**Cost** (ns)
Lock data record:	186
Read index record from index root in L1 cache:	36
Read first level index record:	$11.4\% \times 855 + 88.6\% \times 2555 \approx 2361$
Read data record (including locating the record):	1439
Write data record:	$\lceil \frac{200B}{64B/access} \rceil \times 21ns/access = 84$
Check transaction context:	93
Build update log post:	1092
Build prepare log post:	1092
Log update:	1120
Log prepare:	1120

Table 7.26: NOS Task$_2$ cost description for CTU in SN

This gives:

$$186ns + 36ns + 2361ns + 1439ns + 84ns + 93ns + 1092ns + 1092ns + 1120ns + 1120ns = \underline{\underline{8623ns}}$$

This task is in the critical path.

NOS Task$_3$

This task creates and sends the *ready*-message according to the 2PC protocol. The costs are described in table 7.27.

Description	**Cost** (ns)
Build *ready*-message:	1092
Send *ready*-message:	32 457

Table 7.27: NOS Task$_3$ cost description for CTU in SN

This gives:

$$1092ns + 32\ 457ns = \underline{\underline{33\ 549ns}}$$

This task is in the critical path.

NOS Task$_4$

This task receives and interprets the *commit*-message from the node controller and logs the commit. The log is written to main memory. The task also deletes the transaction context. The costs are described in table 7.28.

Description	**Cost** (ns)
Receive *commit*-message:	32 457
Interpret *commit*-message:	1092
Check transaction context:	93
Build commit log post:	1092
Log commit post:	1120
Unlock data record:	186
Delete transaction context:	186

Table 7.28: NOS Task$_4$ cost description for CTU in SN

$$32\ 457ns + 1092ns + 93ns + 1092ns + 1120ns + 186ns + 186ns = \underline{\underline{36\ 226ns}}$$

This task is <u>not</u> in the critical path, because it is not a part of the sum of all operations needed to be performed before a response can be given back to the transaction service requester.

Total costs

The total costs for each NOS when the CTU transaction is executed is:

$$NOS\ Task_1 + NOS\ Task_2 + NOS\ Task_3 + NOS\ Task_4 =$$

$$33\ 735ns + 8623ns + 33\ 549ns + 36\ 226ns = \underline{\underline{112\ 133ns}}$$

7.5 The Complex Transaction Read (CTR)

This section gives the estimated time used to execute the CTR transaction. Opposed to the CTU transaction, the CTR transaction contains pure read operations. The transaction will therefore involve up to five different nodes, one NOC and four NOSes. The transaction is decomposed into groups of operations called tasks. The following is a description of these tasks together with a time estimate of each. Our assumed DBMS does not log read-only transactions, therefore the tasks NOC Task_8 and NOS Task_4 are not executed. The relationships between tasks are illustrated in figure 7.1.

7.5.1 Node Controller (NOC)

This section gives the estimated workload for the part of the transaction performed at the node running as node controller.

NOC Task_1

This task receives and interprets the transaction. It also creates a transaction context. The costs are described in table 7.29.

Description	**Cost** (ns)
Read transaction from main memory:	100
Interpret transaction:	1092
Create transaction context:	186

Table 7.29: NOC Task_1 cost description for CTR in SN

This gives:

$$100ns + 1092ns + 186ns = \underline{\underline{1378ns}}$$

This task is in the critical path.

NOC Task_2

This task decides which nodes that are to participate in the execution of the transaction. The costs are described in table 7.30.

Description	**Cost** (ns)
Decide participants:	$4 \times 140 = 560$

Table 7.30: NOC Task_2 cost description for CTR in SN

This gives:

$$\underline{\underline{560ns}}$$

This task is in the critical path.

NOC Task$_3$

This task builds the messages containing which reads that are to be done, and sends these to the slaves. The messages have a *prepare*-message piggybacked. The costs are described in table 7.31.

Description	**Cost** (ns)
Build operation messages:	$4 \times 1092 = 4368$
Send operation messages:	$4 \times 32\,457 = 129\,828$

Table 7.31: NOC Task$_3$ cost description for CTR in SN

This gives:

$$4368ns + 129\ 828ns = \underline{\underline{134\ 196ns}}$$

This task is in the critical path.

NOC Task$_4$

This task receives *read*-messages from the slaves, when they have performed the reads. It also interprets the messages. The costs are described in table 7.32.

Description	**Cost** (ns)
Receive *read*-messages:	$4 \times 32\,457 = 129\,828$
Interpret *read*-messages:	$4 \times 1092 = 4368$

Table 7.32: NOC Task$_4$ cost description for CTR in SN

This gives:

$$129\ 828ns + 4368ns = \underline{\underline{134\ 196ns}}$$

This task is in the critical path.

NOC Task$_5$

This task builds a data message containing the data read, and sends this to the calling application. This means that the message is written to main memory. The task also deletes the transaction context. The costs are described in table 7.33.

Description	**Cost** (ns)
Build *data* message:	1092
Write *data* message to main memory:	100
Delete transaction context:	186

Table 7.33: NOC Task$_5$ cost description for CTR in SN

This gives:

$$1092ns + 100ns + 186ns = \underline{\underline{1378ns}}$$

This task is in the critical path.

NOC Task$_6$

This task is not performed when the transaction is a read-only transaction. This is because no logging is performed, and therefore the node performing a read, is not needed to receive a *commit*-message. If the transaction performed contained one or more updates, *commit*-messages would be sent to all nodes performing these updates.

Total costs

The total costs for the NOC when the CTR transaction is executed is:

$$NOC\ Task_1 + NOC\ Task_2 + NOC\ Task_3 + NOC\ Task_4 + NOC\ Task_5 =$$

$$1378ns + 560ns + 134\ 196ns + 134\ 196ns + 1378ns = \underline{\underline{271\ 708ns}}$$

7.5.2 Node Slave (NOS)

This section gives the estimated workload for the part of the transaction performed at one of the four nodes running as node slaves.

NOS Task$_1$

This task receives and interprets the message containing which record to read. It also creates a transaction context. The costs are described in table 7.34.

Description	**Cost** (ns)
Receive operation message:	32 457
Interpret operation messages:	1092
Create transaction context:	186

Table 7.34: NOS Task$_1$ cost description for CTR in SN

This gives:

$$32\ 457ns + 1092ns + 186ns = \underline{\underline{33\ 735ns}}$$

at each node.

This task is in the critical path.

Description	**Cost** (ns)
Lock data record:	186
Read index record from index root in L1 cache:	36
Read first level index record:	$11.4\% \times 855 + 88.6\% \times 2555 \approx 2361$
Read data record (including locating the record):	1439

Table 7.35: NOS Task$_2$ cost description for CTR in SN

NOS Task$_2$

This task locks the data record and performs the read. The costs are described in table 7.35.

This gives:

$$186ns + 36ns + 2361ns + 1439ns = \underline{\underline{4022ns}}$$

This task is in the critical path.

NOS Task$_3$

This task creates and sends the *read*-message according to the 2PC protocol. It also unlocks the data resource and deletes the transaction context. The costs are described in table 7.36.

Description	**Cost** (ns)
Build *read*-message:	1092
Send *read*-message:	32 457
Unlock data record:	186
Delete transaction context:	186

Table 7.36: NOS Task$_3$ cost description for CTR in SN

This gives:

$$1092ns + 32\,457ns + 186ns + 186ns = \underline{\underline{33\,921ns}}$$

This task is in the critical path.

NOS Task$_4$

For the same reasons as described in NOC Task$_6$, this task is not performed.

Total costs

The total costs for the NOS when the CTR transaction is executed is:

$$NOS\ Task_1 + NOS\ Task_2 + NOS\ Task_3 =$$

$$33\,735ns + 4022ns + 33\,921ns = \underline{\underline{71\,678ns}}$$

CHAPTER 8 SE WORKLOAD ESTIMATE

This chapter investigates the total cost of performing the four different transaction types defined in section 2.2 at page 6 in an SE context. We will do that by performing a decomposition of each element presented in figure 8.1. The cost figures presented in the following sections are taken from chapter 6, and are summarized in table 6.1.

As for chapter 7, the structure of this chapter is indeed similar to an analog chapter in our previous work [BJ05]. However, as all the cost figures are different, the costs of performing the different transactions are recalculated. It is also necessary to be familiar with the transactions' sub tasks to make the most of the analysis presented in chapter 9.

Figure 8.1 illustrates the execution path for a transaction. This figure is quite different from the corresponding figure presented in chapter 7. The reason is that there is no node controller or slaves in SE, because all nodes have access to the same shared, global memory. However, as mentioned in section 2.6 at page 11, the main challenge with the SE approach is the need for heavy synchronization to ensure that nodes do not interfere with each other.

Figure 8.1 illustrates the different tasks each node has to go through to ensure serializable execution. For $Task_1$, the node needs to set a lock on the file dictionary in order to locate the tuple that is the target for the current transaction operation. This sub task can be further decomposed: It first need to acquire a semaphore, then set a lock on the file dictionary before releasing the semaphore again. The first time $Task_1$ is executed, the transaction is also read from memory and interpreted. A transaction context is also created. The node can then move on to performing the dictionary operation, before releasing the lock again.

After performing the actual operation ($Task_2$), the node has to start over with $Task_1$ as long as the transaction has more operations to be executed. When the operation queue is empty, the node proceeds to $Task_3$. If the transaction was of type STU or CTU, the node logs a commit and responds to the calling application. If the transaction consisted of pure read operations, such as STR or CTR, this task only consists of responding to the calling application. The transaction then moves on to $Task_4$, which is the only task that is outside the critical path. This task consist of releasing all locks the transaction has acquired during the operation execution. This is done by accessing the lock data structure by hashing on the transaction ID. All the locks the transaction has acquired are linked together in a linked list fashion, and the node releases all locks in one batch by following these links.

The next section presents a calculation of the height of the database b-tree, and the probability of a cache hit when fetching a record. The following sections in turn presents the costs associated with each transaction type.

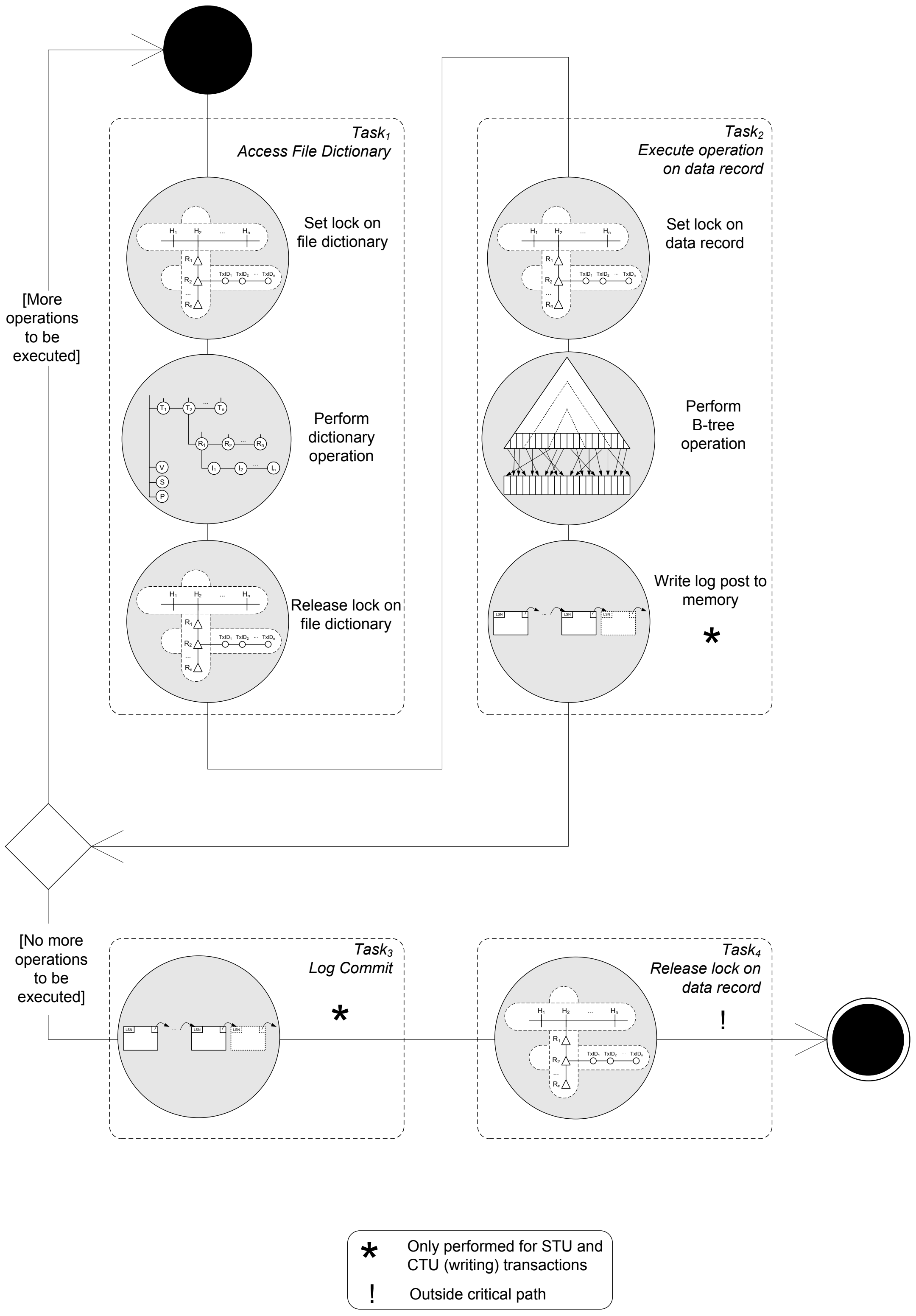

Task$_1$
Access File Dictionary
Set lock on file dictionary
Perform dictionary operation
Release lock on file dictionary
Task$_2$
Execute operation on data record
Set lock on data record
Perform B-tree operation
Write log post to memory
*
[More operations to be executed]
[No more operations to be executed]
Task$_3$
Log Commit
*
Task$_4$
Release lock on data record
!
* Only performed for STU and CTU (writing) transactions
! Outside critical path

Figure 8.1: The execution path in a SE architecture. The transaction performs four tasks during the execution with respect to synchronizing. There are several sub tasks within each task. The circular figures next to the description of the different sub tasks has the purpose of illustrating the typical data structures each task needs to synchronize [1]The details within each task is explained roughly in the introduction of this chapter, and more detailed in the following sections.

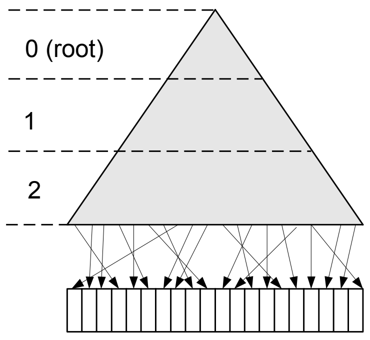


Figure 8.2: The SE b-tree. The b-tree consists of three levels of index records and one level of data records. The height of the b-tree is 4.

8.1 Calculating the b-tree height for SE

This section calculates the b-tree height in the SE case. As explained in section 7.1, we assume the database size to be 10 GB. Although the usual approach is to have a b-tree for every table, we assume that the entire database is contained within a single b-tree.

The number of data blocks in the SE case is:

$$\left\lceil \frac{10GB}{8KB} \right\rceil = \left\lceil \frac{10\ 737\ 418\ 240B}{8192B} \right\rceil = 1\ 310\ 720$$

The number of index blocks on the lowest level is:

$$\left\lceil \frac{1\ 310\ 720}{409} \right\rceil = 3205$$

The number of index blocks on the next level is:

$$\left\lceil \frac{3205}{409} \right\rceil = 8$$

The number of index blocks on the highest level is:

$$\left\lceil \frac{8}{409} \right\rceil = 1$$

This results in a four level b-tree, with three levels of index records, and one level of data records. This is illustrated in figure 8.2. The entire b-tree consists of:

$$1 + 8 + 3205 + 1\ 310\ 720 = 1\ 313\ 934$$

blocks. The total size of the index pages are:

$$(1 + 8 + 3205) \times 8KB \approx 25MB$$

The total amount of L2 cache available in our system is 3MB. This allows us to assume that the two upper levels of index records always reside in cache, and that approximately

$$\frac{3MB}{25MB} = 12\%$$

[1]The details of each data structure is not included here, because it is outside the scope of this report. However, it is described in [BJ05]

of the index blocks at the lowest level can be fetched from L2 cache. The remaining 88% must be fetched from main memory.

The total amount of L1 cache is

$$16KB \times 8 = 128KB$$

However, a node may only access the L1 cache within its core, thus we operate with a L1 cache size of 16KB.

If we assume that the root always resides in L1 cache, there is 8KB left[2]. We assume that records at the lowest index level never reside in L1 cache, as the chance of a record doing this is very small. However, because the first level of index records are accessed frequently, as there are only 8 such blocks, it makes sense to assume that there is a good chance that some of these blocks reside in L1 cache. If we anticipate the remaining 8KB of L1 cache to be available, there is a

$$\frac{8KB}{8 \times 8KB} = 12.5\%$$

probability of a L1 cache hit when fetching first level index records. The remaining 87.5% must be fetched from L2 cache.

8.2 THE SIMPLE TRANSACTION UPDATE (STU)

This section investigates the total cost of performing a Simple Transaction Update (STU) in SE. We will to that by decomposing each task into sub tasks.

8.2.1 $Task_1$: Access file dictionary

This task investigates the cost for accessing the file directory by decomposing the task in the sub tasks indicated in figure 8.1. The tasks are:

- Set lock on file dictionary
- Perform dictionary operation
- Release lock on file dictionary

The following subsections will describe the sub tasks in detail.

Set lock on file dictionary

This sub task reads the transaction from memory, interprets it and creates a transaction context. The task also sets a lock on the file dictionary. This involves setting and releasing a semaphore for the lock hierarchy and setting a lock on the file dictionary.

Using the values from 8.1, this yields a total cost of:

$$100ns + 1092ns + 186ns + 11\ 295ns + 186ns + 11\ 323ns = \underline{24\ 182ns}$$

[2]In reality, the root indices would not make up as much as 8KB.

Description	**Cost** (ns)
Read transaction from memory:	100
Interpret transaction:	1092
Create transaction context:	186
Set semaphore:	11 295
Set file dictionary lock:	186
Release semaphore:	11 323

Table 8.1: $Task_1$, cost description for setting a lock on file dictionary by STU in SE

Perform dictionary operation

This sub task performs a dictionary operation. This can be a read operation or a write operation. Because the number of read operations by far exceeds the number of writes, we only estimate the costs for the read case. This task therefore involves the single sub task of accessing the dictionary.

Description	**Cost** (ns)
Access dictionary:	140

Table 8.2: $Task_1$, cost description for reading file dictionary by STU in SE

Using the values from 8.2, this yields a total cost of:

$$\underline{140ns}$$

Release lock on file dictionary

This sub task releases the lock set on the file dictionary. This involves setting and releasing a semaphore for the lock hierarchy and releasing the lock held for the file dictionary.

Description	**Cost** (ns)
Set semaphore:	11 295
Release file dictionary lock:	186
Release semaphore:	11 323

Table 8.3: $Task_1$, cost description for releasing a lock on file dictionary by STU in SE

Using the values from 8.3, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns = \underline{22\ 804ns}$$

$Task_1$ **cost**

The total cost for $Task_1$ yields:

$$24\ 182ns + 140ns + 22\ 804ns = \underline{47\ 126ns}$$

8.2.2 $Task_2$: Execute operation on data record

This task includes all sub tasks needed to execute an operation on a data record in the b-tree. The sub tasks are:

- Set lock on record
- Perform b-tree operation
- Write log post to memory

The following subsections will describe the sub tasks in detail.

Set lock on data record

This sub task sets a lock on a record. This involves setting and releasing a semaphore for the lock hierarchy and setting a lock.

Description	**Cost** (ns)
Set semaphore:	11 295
Set lock on record:	186
Release semaphore:	11 323

Table 8.4: $Task_2$, cost description for setting a lock by STU in SE

Using the values from 8.4, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns = \underline{22\ 804ns}$$

Perform b-tree operation

This sub task performs an operation on the b-tree. This involves setting and releasing a semaphore for the entire b-tree, reading three index records, reading one data record and writing the data record back to memory.

Description	**Cost** (ns)
Set semaphore	11 295
Read index record from index root in L1 cache:	36
Read first level index record:	$12.5\% \times 36 + 87.5\% \times 855 \approx 753$
Read second level index record:	$12\% \times 855 + 88\% \times 2555 = 2351$
Read data record:	1439
Write data record:	$\lceil \frac{200B}{64B/access} \rceil \times 21 = 84$
Release semaphore:	11 323

Table 8.5: $Task_2$, cost description for performing a b-tree operation by STU in SE

Using the values from 8.5, this yields a total cost of:

$$11\ 295ns + 36ns + 753ns + 2351ns + 1439ns + 84ns + 11\ 323ns = \underline{27\ 281ns}$$

Write log post to memory

This sub task writes a log post to memory. This involves building the log post, checking the transaction context, setting and releasing a semaphore and writing the log post to main memory.

Description	**Cost** (ns)
Build log post:	1092
Check transaction context:	93
Set semaphore:	11 295
Write log post:	1120
Release semaphore:	11 323

Table 8.6: $Task_2$, cost description for writing a log post to memory by STU in SE

Using the values from table 8.6, this yields a total cost of:

$$1092ns + 93ns + 11\ 295ns + 1120ns + 11\ 323ns = \underline{24\ 923ns}$$

$Task_2$ **cost**

The total cost for $Task_2$ yields:

$$22\ 804ns + 27\ 281ns + 24\ 923ns = \underline{75\ 008ns}$$

8.2.3 $Task_3$: Log commit and respond to calling application

This task includes a single sub task, which is to write a commit log post to memory and to write the response to main memory.

Write commit log post and response to memory

This sub task involves building the log post, checking the transaction context, setting and releasing a semaphore and writing the log post and the response to main memory.

Description	**Cost** (ns)
Build commit log post:	1092
Check transaction context:	93
Set semaphore:	11 295
Write commit log post:	1120
Release semaphore:	11 323
Write response to main memory:	100

Table 8.7: $Task_3$, cost description for writing a commit post to memory by STU in SE

Using the values from table 8.7, this yields a total cost of:

$$1092ns + 93ns + 11\ 295ns + 1120ns + 11\ 323ns + 100ns = \underline{25\ 023ns}$$

8.2.4 $Task_4$: Release lock

This task includes a single sub task which is to release the lock on the data record.

Release lock on data record

This sub task involves setting and releasing a semaphore and unlocking the data record. The task also involves deleting the transaction context.

Description	**Cost** (ns)
Set semaphore:	11 295
Release lock on record:	186
Release semaphore:	11 323
Delete transaction context:	186

Table 8.8: $Task_4$, cost description for releasing a lock by STU in SE

Using the values from table 8.8, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns + 186ns = \underline{22\ 990ns}$$

8.2.5 Total costs for the STU transaction

The total cost when the STU transaction is executed is:

$$Task_1 + Task_2 + Task_3 + Task_4 =$$

$$47\ 126ns + 75\ 008 + 25\ 023ns + 22\ 990ns = \underline{\underline{170\ 147ns}}$$

8.3 The Simple Transaction Read (STR)

This section investigates the total cost of performing a STR in SE. We will to that by decomposing each task into sub elements.

8.3.1 $Task_1$: Access file dictionary

This task investigates the cost for accessing the file directory by decomposing the task in sub elements The tasks are:

- Set lock on file dictionary
- Perform dictionary operation
- Release lock on file dictionary

The following subsections will describe the sub tasks in detail.

Set lock on file dictionary

This sub task reads the transaction from memory, interprets it, and creates a transaction context. The task also sets a lock on the file dictionary. This involves setting and releasing a semaphore for the lock hierarchy and setting a lock on the file dictionary.

Description	**Cost** (ns)
Read transaction from memory:	100
Interpret transaction:	1092
Create transaction context:	186
Set semaphore:	11 295
Set file dictionary lock:	186
Release semaphore:	11 323

Table 8.9: $Task_1$, cost description for setting a lock on file dictionary by STR in SE

Using the values from table 8.9, this yields a total cost of:

$$100ns + 1092ns + 186ns + 11\ 295ns + 186ns + 11\ 323ns = \underline{47\ 126ns}$$

Perform dictionary operation

This sub task performs a dictionary operation. This could be a read operation or a write operation. Because the number of read operations by far exceeds the number of writes, we only estimate the costs for the read case. This task therefore involves the single sub task of accessing the dictionary.

Description	**Cost** (ns)
Access dictionary:	140

Table 8.10: $Task_1$, cost description for reading file dictionary by STR in SE

Using the values from table 8.10, this yields a total cost of:

$$\underline{140ns}$$

Release lock on file dictionary

This sub task releases the lock set on the file dictionary. This involves setting and releasing a semaphore for the lock hierarchy and releasing the lock held for the file dictionary.

Description	**Cost** (ns)
Set semaphore:	11 295
Release file dictionary lock:	186
Release semaphore:	11 323

Table 8.11: $Task_1$, cost description for releasing a lock on file dictionary by STR in SE

Using the values from table 8.11, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns = \underline{22\ 804ns}$$

$Task_1$ cost

The total cost for $Task_1$ yields:

$$24\ 182ns + 140ns + 22\ 804ns = \underline{47\ 126ns}$$

8.3.2 $Task_2$: Execute read operation on data record

This task includes all sub tasks needed to execute an operation on a data record in the b-tree. The sub tasks are:

- Set lock on record
- Perform b-tree operation

The following subsections will describe the sub tasks in detail.

Set lock on record

This sub task sets a lock on a record. This involves setting and releasing a semaphore for the lock hierarchy and setting a lock.

Description	**Cost** (ns)
Set semaphore:	11 295
Set lock on record:	186
Release semaphore:	11 323

Table 8.12: $Task_2$, cost description for setting a lock by STR in SE

Using the values from table 8.12, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns = \underline{22\ 804ns}$$

Perform b-tree operation

This sub tasks performs an operation on the b-tree. This involves setting and releasing a semaphore for the entire b-tree, reading three index records and reading one data record.

Using the values from table 8.13, this yields a total cost of:

$$11\ 295ns + 36ns + 753ns + 2351ns + 1439ns + 11\ 323ns = \underline{27\ 197ns}$$

$Task_2$ cost

The total cost for $Task_2$ yields:

$$22\ 804ns + 27\ 197ns = \underline{50\ 001ns}$$

Description	**Cost** (ns)
Set semaphore	11 295
Read index record from index root in L1 cache:	36
Read first level index record:	$12.5\% \times 36 + 88.5\% \times 855 \approx 753$
Read second level index record:	$12\% \times 855 + 88\% \times 2555 = 2351$
Read data record:	1439
Release semaphore:	11 323

Table 8.13: $Task_2$, cost description for performing a b-tree operation by STR in SE

8.3.3 $Task_3$: Log commit and respond to calling application

Reading does not require writing record or log, therefore only the response is written to main memory. The total cost for $Task_3$ is thus:

$$\underline{100ns}$$

8.3.4 $Task_4$: Release lock

This task includes a single sub task which is to release the lock on the data record.

Release lock on data record

This sub involves setting and releasing a semaphore and unlocking the data record. The task also involves deleting the transaction context.

Description	**Cost** (ns)
Set semaphore:	11 295
Release lock on record:	186
Release semaphore:	11 323
Delete transaction context:	186

Table 8.14: $Task_4$, cost description for releasing a lock by STR in SE

Using the values from table 8.14, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns + 186ns = \underline{22\ 990ns}$$

8.3.5 Total costs for the STR transaction

The total cost when the STR transaction is executed is:

$$Task_1 + Task_2 + Task_3 + Task_4 =$$

$$47\ 126ns + 50\ 001ns + 100ns + 22\ 990ns = \underline{\underline{120\ 217ns}}$$

8.4 THE COMPLEX TRANSACTION UPDATE (CTU)

This section investigates the total cost of performing a CTU in SE. We will to that by decomposing each task into sub elements. There are four tuples to be updated during the CTU execution. Referring to figure 8.1, this means that $Task_1$ and $Task_2$ loops four times until the execution queue is empty.

8.4.1 $Task_1$: Access file dictionary

This task investigates the cost for accessing the file directory by decomposing the task in the sub elements indicated in figure 8.1. The tasks are:

- Set lock on file dictionary
- Perform dictionary operation
- Release lock on file dictionary

The following subsections will describe the sub tasks in detail.

Set lock on file dictionary

This sub task reads the transaction from memory, interprets it and creates a transaction context. The task also sets a lock on the file dictionary. This involves setting and releasing a semaphore for the lock hierarchy and setting a lock on the file dictionary. In the execution of the CTU transaction, this sub task is executed four times. However, the work of reading and interpreting the transaction, and creating a transaction context, is only performed once. Therefore we operate with two distinct costs for this task.

Description	**Cost** (ns)
Read transaction from memory (performed once):	100
Interpret transaction (performed once):	1092
Create transaction context (performed once):	186
Set semaphore:	11 295
Set file dictionary lock:	186
Release semaphore:	11 323

Table 8.15: $Task_1$, cost description for setting a lock on file dictionary by CTU in SE

Using the values from table 8.15, this yields a total cost of:

$$100ns + 1092ns + 186ns + 11\ 295ns + 186ns + 11\ 323ns = \underline{24\ 182ns}$$

the first time the sub task is executed, and

$$11\ 295ns + 186ns + 11\ 323ns = \underline{22\ 804ns}$$

all other times.

Perform dictionary operation

This sub tasks performs a dictionary operation. This could be a read operation or a write operation. Because the number of read operations by far exceeds the number of writes, we only estimate the costs for the read case. This task therefore involves the single sub task of accessing the dictionary

Description	**Cost** (ns)
Access dictionary:	140

Table 8.16: $Task_1$, cost description for reading file dictionary by CTU in SE

Using the values from table 8.16, this yields a total cost of:

$$\underline{140ns}$$

Release lock on file dictionary

This sub task releases the lock set on the file dictionary. This involves setting and releasing a semaphore for the lock hierarchy and releasing the lock held for the file dictionary.

Description	**Cost** (ns)
Set semaphore:	11 295
Release file dictionary lock:	186
Release semaphore:	11 323

Table 8.17: $Task_1$, cost description for releasing a lock on file dictionary by CTU in SE

Using the values from table 8.17, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns = \underline{22\ 804ns}$$

$Task_1$ **cost**

The total cost for $Task_1$ yields:

$$24\ 182ns + 140ns + 22\ 804ns = \underline{47\ 126ns}$$

the first time the task is executed, and

$$22\ 804ns + 140ns + 22\ 804ns = \underline{45\ 748ns}$$

all other times. These are referred to as $Task_1start$ and $Task_1rest$, respectively.

8.4.2 $Task_2$: Execute operation on data record

This task includes all sub tasks needed to execute an operation on a data record in the b-tree. The sub tasks are:

- Set lock on record
- Perform b-tree operation

- Write log post to memory

The following subsections will describe the sub tasks in detail.

Set lock on record

This sub task sets a lock on a record. This involves setting and releasing a semaphore for the lock hierarchy and setting a lock.

Description	**Cost** (ns)
Set semaphore:	11 295
Set lock on record:	186
Release semaphore:	11 323

Table 8.18: $Task_2$, cost description for setting a lock by CTU in SE

Using the values from table 8.18, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns = \underline{22\ 804ns}$$

Perform b-tree operation

This sub tasks performs an operation on the b-tree. This involves setting and releasing a semaphore for the entire b-tree, reading three index records, reading one data record and writing the data record back to memory.

Description	**Cost** (ns)
Set semaphore:	11 295
Read index record from index root in L1 cache:	36
Read first level index record:	$12.5\% \times 36 + 87.5\% \times 855 \approx 753$
Read second level index record:	$12\% \times 855 + 88\% \times 2555 = 2351$
Read data record:	1439
Write data record:	$\lceil \frac{200B}{64B/access} \rceil \times 21ns = 84ns$
Release semaphore:	11 323

Table 8.19: $Task_2$, cost description for performing a b-tree operation by CTU in SE

Using the values from table 8.19, this yields a total cost of:

$$11\ 295ns + 36ns + 753ns + 2351ns + 1439ns + 84ns + 11\ 323ns = \underline{27\ 281ns}$$

Write log post to memory

This sub task writes a log post to memory. This involves building the log post, checking the transaction context, setting and releasing a semaphore and writing the log post to main memory.

Using the values from table 8.20, this yields a total cost of:

Description	**Cost** (ns)
Build log post:	1092
Check transaction context:	93
Set semaphore:	11 295
Write log post:	1120
Release semaphore:	11 323

Table 8.20: $Task_2$, cost description for writing a log post to memory by CTU in SE

$$1092ns + 93ns + 11\ 295ns + 1120ns + 11\ 323ns = \underline{24\ 923ns}$$

$Task_2$ **cost**

The total cost for $Task_2$ yields:

$$22\ 804ns + 27\ 281ns + 24\ 923ns = \underline{75\ 008ns}$$

for each operation.

8.4.3 $Task_3$: Log commit and respond to calling application

This task includes a single sub task. This is to write a commit log post to memory, and to write the response to main memory.

Write commit log post and response to memory

This sub task involves building the log post, checking the transaction context, setting and releasing a semaphore and writing the log post and the response to main memory.

Description	**Cost** (ns)
Build update log post:	1092
Check transaction context:	93
Set semaphore:	11 295
Write commit log post:	1120
Release semaphore:	11 323
Write response to main memory:	100

Table 8.21: $Task_3$, cost description for writing a commit post to memory by CTU in SE

Using the values from table 8.21, this yields a total cost of:

$$1092ns + 93ns + 11\ 295ns + 1120ns + 11\ 323ns + 100ns = \underline{25\ 023ns}$$

8.4.4 $Task_4$: Release locks

This task includes a single sub task which is to release the locks on the data records.

Release lock on data records

This sub task involves setting and releasing a semaphore and unlocking the data records. It also involves deleting the transaction context.

Description	**Cost** (ns)
Set semaphore:	11 295
Release lock on record:	186
Release semaphore:	11 323
Delete transaction context:	186

Table 8.22: $Task_4$, cost description for releasing a lock by CTU in SE

Using the values from table 8.22, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns + 186ns = \underline{22\ 990ns}$$

8.4.5 Total costs for the CTU transaction

$Task_1$ is split into $Task_1start$ and $Task_1rest$. $Task_1start$ is only performed once, while $Task_1rest$ is performed three times and is thus multiplied by 3. $Task_2$ is multiplied by 4, because it is executed for each operation.

The total cost when the CTU transaction is executed is:

$$Task_1start + (3 \times Task_1rest) + (4 \times Task_2) + Task_3 + Task_4 =$$

$$47\ 126ns + (3 \times 45\ 748ns) + (4 \times 75\ 008ns) + 25\ 023ns + 22\ 990ns = \underline{\underline{532\ 414ns}}$$

8.5 The Complex Transaction Read (CTR)

This section investigates the total cost of performing a CTR in SE. We will to that by decomposing each task into sub elements. There are four tuples to be updated during the CTR execution. Referring to figure 8.1, this means that $Task_1$ and $Task_2$ loops four times until the execution queue is empty.

8.5.1 $Task_1$: Access file dictionary

This task investigates the cost for accessing the file directory by decomposing the task in sub elements The tasks are:

- Set lock on file dictionary
- Perform dictionary operation
- Release lock on file dictionary

The following subsections will describe the sub tasks in detail.

Set lock on file dictionary

This sub task reads the transaction from memory, interprets it and creates a transaction context. The task also sets a lock on the file dictionary. This involves setting and releasing a semaphore for the lock hierarchy and setting a lock on the file dictionary. In the execution of the CTR transaction, this sub task is executed four times. However, the work of reading and interpreting the transaction, and creating a transaction context, is only performed once. Therefore we operate with two distinct costs for this task.

Description	**Cost** (ns)
Read transaction from memory (performed once):	100
Interpret transaction (performed once):	1092
Create transaction context (performed once):	186
Set semaphore:	11 295
Set file dictionary lock:	186
Release semaphore:	11 323

Table 8.23: $Task_1$, cost description for setting a lock on file dictionary by CTR in SE

Using the values from table 8.23, this yields a total cost of:

$$100ns + 1092ns + 186ns + 11\ 295ns + 186ns + 11\ 323ns = \underline{24\ 182ns}$$

the first time the sub task is executed, and

$$11\ 295ns + 186ns + 11\ 323ns = \underline{22\ 804ns}$$

all other times.

Perform dictionary operation

This sub task performs a dictionary operation. This could be a read operation or a write operation. Because the number of read operations by far exceeds the number of writes, we only estimate the costs for the read case. This task therefore involves the single sub task of accessing the dictionary.

Description	**Cost** (ns)
Access dictionary:	140

Table 8.24: $Task_1$, cost description for reading file dictionary by CTR in SE

Using the values from table 8.24, this yields a total cost of:

$$\underline{140ns}$$

Release lock on file dictionary

This sub task releases the lock set on the file dictionary. This involves setting and releasing a semaphore for the lock hierarchy and releasing the lock held for the file dictionary.

Description	**Cost** (ns)
Set semaphore:	11 295
Release file dictionary lock:	186
Release semaphore:	11 323

Table 8.25: $Task_1$, cost description for releasing a lock on file dictionary by CTR in SE

Using the values from table 8.25, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns = \underline{22\ 804ns}$$

$Task_1$ cost

The total cost for $Task_1$ yields:

$$24\ 182ns + 140ns + 22\ 804ns = \underline{47\ 126ns}$$

the first time the task is executed, and

$$22\ 804ns + 140ns + 22\ 804ns = \underline{45\ 748ns}$$

all other times. These are referred to as $Task_1start$ and $Task_1rest$, respectively.

8.5.2 $Task_2$: Execute read operation on data record

This task includes all sub tasks needed to execute an operation on a data record in the b-tree. The sub tasks are:

- Set lock on record
- Perform b-tree operation

The following subsections will describe the sub tasks in detail.

Set lock on record

This sub task sets a lock on a record. This involves setting and releasing a semaphore for the lock hierarchy and setting a lock.

Description	**Cost** (ns)
Set semaphore:	11 295
Set lock on record:	186
Release semaphore:	11 323

Table 8.26: $Task_2$, cost description for setting a lock by CTR in SE

Using the values from table 8.26, this yields a total cost of:

$$11\ 295ns + 186ns + 11\ 323ns = \underline{22\ 804ns}$$

Perform b-tree operation

This sub tasks performs an operation on the b-tree. This involves setting and releasing a semaphore for the entire b-tree, reading three index records, and reading one data record.

Description	**Cost** (ns)
Set semaphore:	11 295
Read index record from index root in L1 cache:	36
Read first level index record:	$12.5\% \times 36 + 87.5\% \times 855 \approx 753$
Read second level index record:	$12\% \times 855 + 88\% \times 2555 \approx 2351$
Read data record:	1439
Release semaphore:	11 323

Table 8.27: $Task_2$, cost description for performing a B-tree operation by CTR in SE

Using the values from table 8.27, this yields a total cost of:

$$11\,295ns + 36ns + 753ns + 2351ns + 1439ns + 11\,323ns = \underline{27\,197ns}$$

$Task_2$ **cost**

The total cost for $Task_2$ yields:

$$22\,804ns + 27\,197ns = \underline{50\,001ns}$$

8.5.3 $Task_3$: Log commit and respond to calling application

Reading does not require writing record or log, therefore only the response is written to main memory. The total cost for $Task_3$ is thus:

$$\underline{100ns}$$

8.5.4 $Task_4$: Release lock

This task includes a single sub task which is to release the lock on the data record.

Release lock on data record

This sub task involves setting and releasing a semaphore and unlocking the data record. It also involves deleting the transaction context.

Using the values from table 8.28, this yields a total cost of:

$$11\,295ns + 186ns + 11\,323ns + 186ns = \underline{22\,990ns}$$

Description	Cost
Set semaphore:	11 295
Release lock on record:	186
Release semaphore:	11 323
Delete transaction context:	186

Table 8.28: $Task_4$, cost description for releasing a lock by CTR in SE

8.5.5 Total costs for the CTR transaction

$Task_1$ is split into $Task_1start$ and $Task_1rest$. $Task_1start$ is only performed once, while $Task_1rest$ is performed three times and is thus multiplied by 3. $Task_2$ is multiplied by 4, because it is executed for each operation.

The total cost when the CTR transaction is executed is:

$$Task_1start + (3 \times Task_1rest) + (4 \times Task_2) + Task_3 + Task_4 =$$

$$47\ 126ns + (3 \times 45\ 748ns) + (4 \times 50\ 001ns) + 100ns + 22\ 990ns = \underline{\underline{407\ 463ns}}$$

Chapter 9 Analysis

This chapter presents an analysis based on the calculations presented in chapter 7 and chapter 8. The chapter starts with presenting a workload summary for SN and SE in sections 9.1 and 9.2, respectively. Next, section 9.3 presents a comparison between the results obtained for the two approaches. The results are also compared with the results in our previous work, [BJ05]. Finally, section 9.4 presents a sensitivity analysis. The ratio between the costs for message passing and synchronizing is varied. The number of expected processes is also varied.

9.1 SN workload summary

This section gives a brief summary and presentation of the costs calculated in chapter 7. Table 9.1 summarizes the costs associated with each of the four transactions (STU, STR, CTU and CTR), when utilizing SN.

As stated earlier, we assume that all records touched by the complex transactions reside at different nodes. We also assume that a NOC is not at the same node as any NOSes of the same transaction, and that there is no queuing of operations. These assumptions allow us to anticipate full parallelism in the SN case. All NOSes are executed in parallel, whereas the NOSes and the NOC are executed in an interleaved fashion. The latter is illustrated in figure 7.1 at page 76. This makes an impact on the costs described in table 9.1. The time used to execute the NOSes is equal for the simple and complex transactions of the different types. The total time used to execute the different transactions is therefore simply the sum of the NOC and one of the NOSes. As we shall see, this is not the case when using SE, where all operations are executed at a single node.

9.1.1 Increasing the number of operations

The ratio between the complex and the simple transactions is

$$\frac{CTU}{STU} = \frac{522\ 647 ns}{220\ 286 ns} \approx 2.4$$

Transaction	**NOC** (ns)	**NOS** (ns)	**Sum** (ns)
STU:	108 153	112 133	220 286
STR:	69 994	71 678	141 672
CTU:	410 514	112 133	522 647
CTR:	271 708	71 678	343 386

Table 9.1: Cost summary for SN

Task	**Description**	**Cost** (ns)
NOC Task$_1$	Decide participants	140
NOC Task$_3$	Build operation message	1092
	Send operation message	32 457
NOC Task$_4$	Receive *ready* message	32 457
	Interpret *ready* message	1092
NOC Task$_6$	Build *commit* message	1092
	Send *commit* message	32 457

Table 9.2: The additional NOC cost for an additional operation

in the case of updates, and

$$\frac{CTR}{STR} = \frac{343\ 386ns}{141\ 672ns} \approx 2.4$$

in the case of reads

Because all NOSes are executed in parallel, the extra work done by the NOC is the only cost added to the total transaction execution time when performing an additional operation. If we examine the calculation of the CTU transaction presented in section 7.4, it is apparent that the extra work imposed on the NOC by an additional operation is as presented in table 9.2.

This yields a total additional cost of

$$140ns + 1092ns + 32\ 457ns + 32\ 457ns + 1092ns + 1092ns + 32\ 457ns = 100\ 787ns$$

per added operation.

If we compare this additional cost to the total cost of executing the NOC in a transaction where a single NOS is involved (the STU transaction), the ratio is

$$\frac{100\ 787ns}{108\ 153ns} \approx 0.932$$

This means that when going from one to two operations, the cost associated with executing the NOC is nearly doubled.

The same phenomena is also apparent when examining the ratio between the time needed to run the NOC in the case of an STU transaction and in the case of a CTU transaction. This ratio is

$$\frac{NOC_{CTU}}{NOC_{STU}} = \frac{410\ 514ns}{108\ 153ns} \approx 3.8$$

This is not surprising as the CTU transaction involves four times as many operations as the STU transaction.

9.1.2 Workload

The cost associated with message passing is unquestionably the most expensive single cost. It is therefore interesting to examine the cost for message passing compared to the total costs of the different transactions. This is presented in table 9.3.

It is obvious that the time used to execute the transactions depends on the cost associated with message passing.

When examining table 9.3, it may look like much time is spent waiting for messages. As an example, let us again consider the CTU transaction. When the NOC is finished sending

Transaction	Percentage
STU:	$\frac{6\times 32\ 457ns}{220\ 286ns} \approx 88\%$
STR:	$\frac{4\times 32\ 457ns}{141\ 672ns} \approx 92\%$
CTU:	$\frac{15\times 32\ 457ns}{522\ 647ns} \approx 93\%$
CTR:	$\frac{10\times 32\ 457}{343\ 386ns} \approx 95\%$

Table 9.3: The percentage of the total transaction execution time used for message passing

the final operation message to the NOSes, which is done serially, the NOS that received the first operation message have already responded. Therefore, the NOC does not waste valuable processing time waiting for response from the NOSes.

On the NOS side, however, more time is spent waiting. According to figure 7.1, when a NOS has sent a *ready* message to the NOC, it has to wait for the NOC to perform NOC Task$_4$, NOC Task$_5$ and parts of NOC Task$_6$ before receiving a *commit* message. If the NOSes receive *commit* messages in the same order as they sent *ready* messages, then according to section 7.4, the NOS has been idle for

$$\overbrace{136\ 501ns}^{\text{Task}_4} + \underbrace{1192ns}_{\text{Task}_5} + \overbrace{4368ns + 32\ 457ns}^{\text{Task}_6} = 174\ 518ns$$

According to section 7.4, the total time spent executing the NOS is 112 133ns. This means that a node executing a single NOS is idle

$$\frac{174\ 518}{112\ 133ns + 174\ 518ns} \approx 61\%$$

of the time. Clearly, a node may run more than a single CTU NOS without swamping the system. However, an actual DBMS trying to meet high availability demands would never run at 100% of the maximum capacity in case of any unforeseen conditions. It is on the other hand common to keep the average load at 70% of the maximum.

9.2 SE WORKLOAD SUMMARY

This section describes and discusses the results from the cost calculations presented in chapter 8. The results are presented in table 9.4.

Contrasting SN, there is no parallelism in SE. Somewhat surprising, this is no obvious drawback in the execution of small transactions such as STR and STU. The reason is of course that the NOC overhead that SN carries in order to be able to run operations in parallel is not apparent in SE.

As illustrated in table 9.4, there is a significant leap in costs when moving from transactions containing a single operation to transactions containing four operations. Due to the lack of

Transaction	**Cost** (ns)
STU:	170 147
STR:	120 217
CTU:	532 414
CTR:	407 463

Table 9.4: Cost summary for SE

parallelism, adding a single operation to a transaction implies adding a considerable cost to the total transaction execution cost.

9.2.1 Increasing the number of operations

When inspecting the relative cost ratio between single and complex transactions, the ratio is

$$\frac{CTU}{STU} = \frac{532\ 414}{170\ 147} \approx 3.1$$

in the case of updates, and

$$\frac{CTR}{STR} = \frac{407\ 463}{120\ 217} \approx 3.4$$

in the case of reads.

The performance gain of executing four operations in a transaction opposed to serially executing four transactions each containing a single operation, is attributable to the decomposition of the sub tasks that only need to be executed once per transaction. Consulting figure 8.1 at page 98, these sub tasks are $Task_3$ and $Task_4$. In addition (but not apparent in the figure), parts of $Task_1$ only need to be executed once. The decomposition of $Task_1$ is described in section 8.4.1 at page 108. The decomposition shows that the cost difference between the first and the consecutive executions of $Task_1$ is indeed small:

$$\begin{aligned}&Task_1start - Task_1rest = \\ &\quad 47\ 126ns - 45\ 748ns = 1378ns\end{aligned}$$

Thus, while $Task_3$ and $Task_4$ only need to be performed once per transaction, the most significant parts of $Task_1$ and $Task_2$ in its entirety still need to be performed for each operation. $Task_1$ and $Task_2$ yield indeed the most significant costs. A closer look at the cost figures for the CTU transaction illustrates this. The first operation execution yields a cost of

$$\begin{aligned}&Task_1start + Task_2 = \\ &47\ 126ns + 75\ 008ns = 122\ 134ns\end{aligned}$$

The 3 consecutive operation executions each yields a cost of

$$\begin{aligned}&Task_1rest + Task_2 = \\ &45\ 748ns + 75\ 008ns = 120\ 756ns\end{aligned}$$

which yields a total of

Transaction	**Cost** (ns)
$Task_1start$	$\frac{45\ 236ns}{47\ 126} \approx 96\%$
$Task_1rest$	$\frac{45\ 236ns}{45\ 748} \approx 99\%$
$Task_2$:	$\frac{67\ 854ns}{75\ 008} \approx 90\%$
$Task_3$:	$\frac{22\ 618ns}{25\ 023} \approx 90\%$
$Task_4$	$\frac{22\ 618ns}{22\ 990} \approx 98\%$

Table 9.5: Synchronization percentage cost in SE for the CTU transaction

$$3 \times 120\ 756ns = 362\ 267ns$$

Before completion, $Task_3$ and $Task_4$ is run once, with a total cost of

$$Task_3 + Task_4 =$$
$$25\ 023ns + 22\ 990ns = 48\ 013ns$$

Clearly, the biggest cost in SE is associated with the execution of the first two tasks. The next question to answer is what mechanism makes the biggest cost in SE.

9.2.2 Workload

Not surprising, the most significant costs is associated with the synchronization using System V Semaphores. To realize this, let us again consider the CTU transaction. Inspecting the cost figures, the percentage of the cost associated with synchronization using semaphores, is for each sub task illustrated in table 9.5. Although only listing these figures for CTU, the same percentage also applies for the other classes of transactions, with the exception of the percentage of $Task_3$. Because there is no synchronization in $Task_3$ in the read cases, the percentage yields 0% for transactions CTR and STR. Obviously, $Task_1rest$ is only executed for CTU and CTR.

It is apparent that the most expensive mechanism in use is the synchronization primitives. A lot of time could be saved if a less expensive synchronization primitive could substitute the System V semaphores.

9.3 COMPARISON

This section presents a comparison between the results obtained for SN and SE, calculated in chapters 7 and 8, respectively. We also draw the lines back to our previous work, and state the main differences.

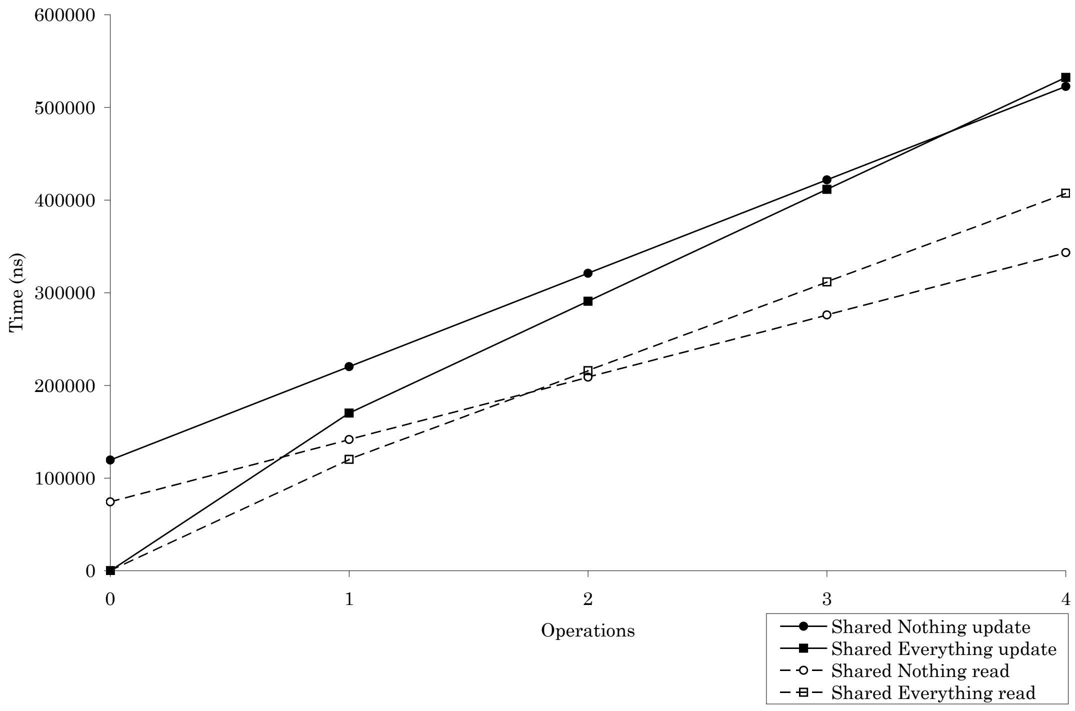


Figure 9.1: Transaction execution cost for up to four transactions. There are four curves in the graph. The dashed lines illustrate the costs associated with running pure read transactions, whereas the solid lines illustrate the costs associated with running pure update operations. It is apparent that SE is the cheaper alternative when performing one operation. SN read is chapter than SE for operations > 1, whereas SN update is preferable for operations > 3.

The costs for the different architectural approaches are already summarized in sections 9.1 and 9.2. A recapitulation of the costs presented in these sections is shown in table 9.6. These cost figures are used to present the following graphs.

The graph shown in figure 9.1 illustrates the cost associated with executing transactions containing up to four operations. As before, we assume that different operations in the SN case reside at different nodes. There are four curves in the graph. The dashed lines illustrate the costs associated with running pure read transactions, whereas the solid lines illustrate the costs associated with running pure update operations.

According to the graph, SE is the cheaper alternative when the transactions are only performing one operation. SN is more expensive, because the NOC component of SN is pure overhead. As explained earlier, the NOC is used to coordinate distributed transactions and the distributed commit. Because transactions involving a single operation cannot be distributed,

Transaction	**SN**			**SE**
	NOC	NOS	Sum	
STU:	108 153	112 133	220 286	170 147
STR:	69 994	71 678	141 672	120 217
CTU:	410 514	112 133	522 647	532 414
CTR:	271 708	71 678	343 386	407 463

Table 9.6: Cost summary for SN and SE

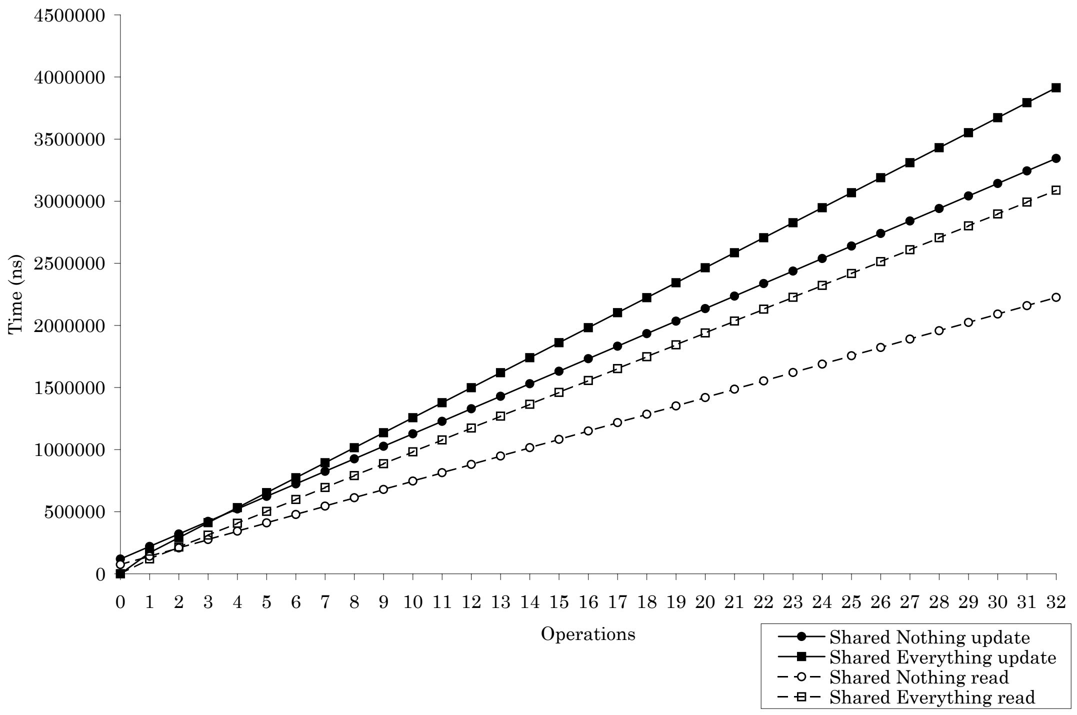


Figure 9.2: Transaction execution cost for up to 32 operations. The trend suggested in 9.1 continues. At 32 operations, SN performs substantially better than SE.

the work performed by the NOC is superfluous. Because there is no such controller component in SE, both the read and the update single operation transactions are cheaper when using SE. This was also the case in our previous work presented in [BJ05].

When increasing the number of operations, the curves for SN and SE intersect in both the read and the update case. This means that when performing additional operations, the SN approach is eventually cheaper.

In our previous work, SN was cheapest for transactions performing two or more transactions. This was the case in both the read and the update case. When examining the graph presented in figure 9.1, the number of operations needed to make SN the cheaper alternative deviates somewhat from our previous work. In the read case, the number of operations needed is still two. In the update case, however, the number of operations needed has increased to four. This is because the minimum, inaccurate cost estimates used in our previous work appears to have a greater impact on SN than SE.

This is perhaps not too surprising. The message passing cost in this report is

$$\frac{Msg\_passing_{new}}{Msg\_passing_{old}} = \frac{32\,457ns}{75ns} \approx 433$$

times greater than in the previous, whereas the cost for synchronization is

$$\frac{Synchronization_{new}}{Synchronization_{old}} = \frac{11\,295ns}{30ns} \approx 377$$

times greater than in the previous report.

The graph shown in figure 9.2 illustrates the costs associated with executing transactions involving up to 32 operations. The part of the graph up to four operations is therefore identical to the graph presented in figure 9.1. There are four curves in the graph. The dashed curves illustrate the read case, whereas the solid lines illustrate the update case. According to the graph, SN outperforms SE in both the update and the read case as the number of operations increases. Because we are assuming full parallelism in the SN case, all operations are running at different nodes. This means that when 32 operations are reached, these operations are running at 32 different nodes. This is not plausible, and this scenario would rarely take place in the real world.

In our previous work, SN outperformed SE by almost one order of magnitude when running 32 operations. According to the graph presented in figure 9.2, this is no longer the case. When the transactions involve 32 operations SN does not even outperform SE by a factor of two.

9.4 SENSITIVITY ANALYSIS

This section presents a sensitivity analysis for the costs of message passing and synchronization. These costs are chosen because they are the most significant isolated cost elements for each architectural approach.

Section 9.4.1 examines the performance when executing pure read transactions, whereas section 9.4.2 examines the performance when running pure update transactions. In both sections the ratio between the costs of message passing and synchronization is varied.

Because the choice of costs presented in chapter 6 is highly depending on the expected number of processes, this is an element that may have great impact on the relative performance of SN and SE. Section 9.4.3 presents an analysis by varying the number of expected processes.

9.4.1 Varying the cost ratio for read transactions

In this section we perform a sensitivity analysis by varying the ratio between the costs for message passing in SN and synchronization in SE. This is performed for the pure read transactions STR and CTR. It is also performed for a general read transaction where the number of operations is not predetermined. The objective of the latter is to determine the ratio that marks a crossing between the performance of SN and SE in the read case. Having benchmarked these costs, this ratio could yield a suggestion on which memory architecture to implement on a DBMS.

The following sections present an analysis for each of the transactions listed above.

The STR transaction

This section presents an analysis based on the STR transaction. The ratio between message passing in SN and synchronization in SE is varied, as illustrated by in figure 9.3. There are three horizontal curves in the graph. These curves illustrate the cost of running the STR transaction on SE when the cost for synchronization is the original cost, two times the original cost and three times the original cost. The three other curves illustrate the cost of executing the STR transaction on SN in the different cases.

In order to explain this more thoroughly, let us consider the two dashed curves. The horizontal curve illustrates the cost associated with running the STR transaction on SE when the cost for synchronization is equal to the original cost. The cost is not varied, and the time used to execute the transaction is therefore constant. The other dashed curve illustrates the cost associated with running the STR transaction on SN.

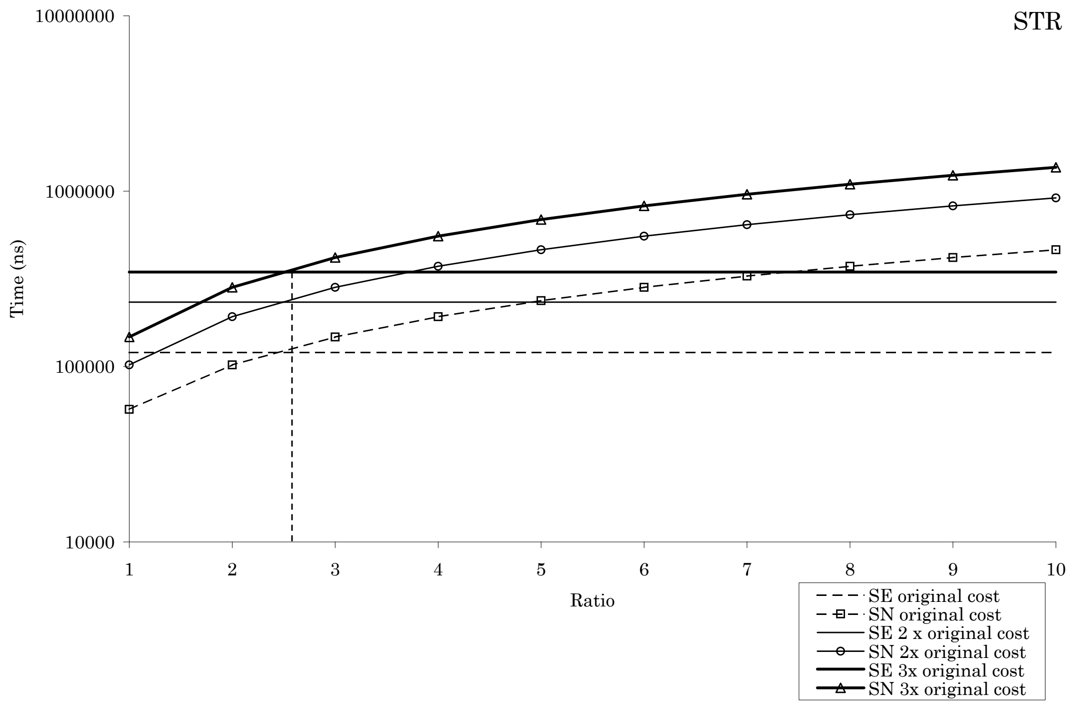


Figure 9.3: Varying the cost ratio for the STR transaction. There are three horizontal curves in the graph. These curves illustrate the costs associated with running the STR transaction on SE when the cost for synchronization is the original cost, two times the original cost and three times the original cost. The x-axis represents the ratio between the cost for message passing and the cost for synchronization. There are three pairs of curves. These are illustrated with dashed, solid and bold lines. The arched curves represent the costs associated with running the STR transaction on SN. The costs used for message passing when creating this curves are x times the cost used for synchronization in the matching SE curve. All pairs of curves intersect at $x = 2.5$. This suggests that if the cost for message passing is 2.5 times the cost for synchronization, SN and SE perform equally good when running STR transactions.

Opposed to the cost for synchronization, the cost for message passing is not held constant. The x-axis represents the ratio between these two costs. Thus at $x = 1$, the cost for message passing is set equal to the original cost for synchronization. At $x = 2$, the cost for message passing is set to two times the original cost for synchronization, and so forth. The intersection between the two curves illustrates the ratio that makes SN and SE perform equally well.

The other pairs of curves behave in the same manner. The only difference is that the starting points are shifted from the original cost for synchronization, to two and three times the original cost.

As illustrated in the graph, all pairs of curves intersect at approximately $x = 2.5$. This suggests that if the cost for message passing is approximately 2.5 times the cost for synchronization, SN and SE will perform equally good when running the STR transaction.

Another way to see this is to examine the number of times messages are sent or received in SN and compare this with the number of times a semaphore is set and released in SE. These numbers are for the STR case already described in sections 7.3 and 8.3, respectively. According to these sections, messages are sent or received four times, whereas semaphores are set or released 10 times. The ratio between these numbers is

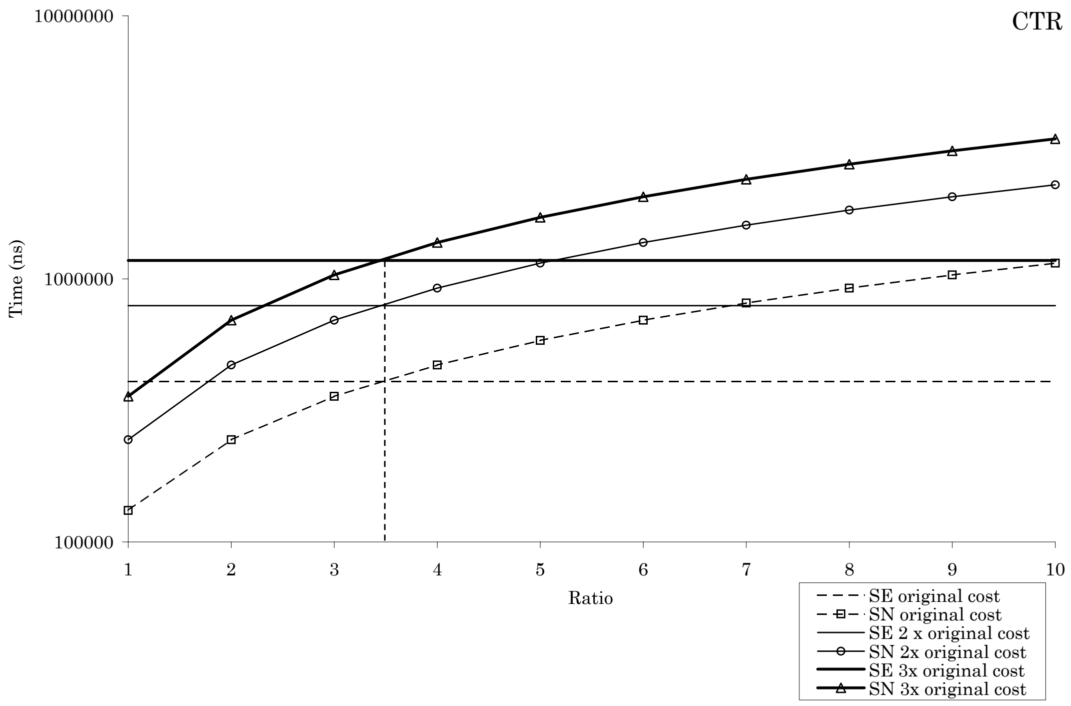


Figure 9.4: Varying the cost ratio for the CTR transaction. There are three horizontal curves in the graph. These curves illustrate the costs associated with running the CTR transaction on SE when the cost for synchronization is the original cost, two times the original cost and three times the original cost. The x-axis represents the ratio between the cost for message passing and the cost for synchronization. There are three pairs of curves. These are illustrated with dashed, solid and bold lines. The arched curves represent the costs associated with running the CTR transaction on SN. The costs used for message passing when creating this curves are x times the cost used for synchronization in the matching SE curve. All pairs of curves intersect at $x \approx 3.5$. This suggests that if the cost for message passing is 3.5 times the cost for synchronization, SN and SE perform equally good when running CTR transactions.

$$\frac{10}{4} = 2.5$$

Because the cost for message passing (see section 9.1) and synchronization (see section 9.2) dominates the total, the remaining costs do not make a significant impact on the ratio. As the costs of message passing and synchronization grows, the ratio to make SN and SE perform equally good when executing the STR transaction approaches 2.5. That is, given that the approaches perform equally well, then

$$\lim_{m,s \to \infty} \frac{m}{s} = 2.5$$

where m is the cost for message passing and s is the cost for synchronization.

This means that if the cost of passing messages is greater than 2.5 times the cost for synchronization, SE will perform best for a system running STR transactions. It is also valid the other way around. If message passing is less than 2.5 times the cost for synchronization, SN is the better alternative. It is important to notice that this ratio is only valid for the STR transaction. As we shall see, read transactions involving a different number of operations will have different ratios.

Task	**Description**	**Cost** (ns)
NOC $Task_2$	Decide participants	140
NOC $Task_3$	Build operation message	1092
	Send operation message	32 457
NOC $Task_4$	Receive *read* message	32 457
	Interpret *read* message	1092

Table 9.7: The additional cost for an additional read operation in SN

However, if the percentages of the total transaction execution time used for message passing and synchronization decrease, these costs will not longer predominate the total costs, and the ratio will deviate.

The CTR transaction

This section presents an analysis based on the CTR transaction. The ratio between message passing in SN and synchronization in SE is varied. This is illustrated by the graph in figure 9.4, which is analogous to the graph described in the previous section.

There are three horizontal curves in the graph. These illustrate the costs associated with running the CTR transaction on SE when the cost for synchronization is the original cost, two times the original cost and three times the original cost, respectively. The three remaining curves illustrate the costs associated with running the CTR transaction on SN in the different cases, as described in the previous section. The x-axis represents the ratio between the different costs used.

As already explained, the curves intersect at approximately $x = 2.5$ when running the STR transaction. This is no longer the case when running the CTR transaction. According to the graph, all pairs of curves intersect at approximately $x \approx 3.5$. This is because the ratio between the number of times a message is sent or received in SN and the number of times a semaphore is set or released in SE has changed. According to sections 7.5 and 8.5, these numbers are 10[1] and 34, respectively. The ratio between these numbers is

$$\frac{34}{10} = 3.4$$

For the same reasons as explained in the previous section, given that the approaches perform equally good, this yields

$$\lim_{m,s\to\infty} \frac{m}{s} = 3.4$$

where m is the cost for message passing and s is the cost for synchronization.

This means that if the cost of passing messages is greater than 3.4 times the cost for synchronization, SE will perform best for a system running CTR transactions. On the other hand, if message passing is cheaper than 3.4 times the cost for synchronization, SN is preferable. It is important to notice that this ratio is only valid for the CTR transaction, read transactions involving a different number of operations will have different ratios.

[1]Although there are four NOSes involved in a CTR transaction, we assume that all NOSes are executed in parallel. The numbers of messages sent is therefore calculated by summing up the numbers sent from NOC and one NOS.

Task	**Description**	**Cost** (ns)
$Task_1 rest$	Set semaphore	11 295
	Set file dictionary lock	186
	Release semaphore	11 323
	Access dictionary	140
	Set semaphore	11 295
	Release file dictionary lock	186
	Release semaphore	11 323
$Task_2$	Set semaphore	11 295
	Set lock on record	186
	Release semaphore	11 323
	Set semaphore	11 295
	Read index record from index root in L1 cache	36
	Read first level index record	753
	Read second level index record	2351
	Read data record	1439
	Release semaphore	11 323

Table 9.8: The additional cost for an additional read operation in SE

General read transaction

This section presents an analysis based on a general read transaction. The purpose of this section is to establish a ratio between the cost for message passing and the cost for synchronization that can be used to tell which approach yields best results regardless of how many operations that are added to the transaction. Thus, we are not limiting ourselves to transactions with a predefined number of operations such as the STR and CTR transactions described above. In order to do this, we have to examine the sub tasks of a read transaction when running on SN and SE.

As illustrated by the graphs in figures 9.1 and 9.2 in section 9.3, the curves for the costs of executing a read operation on SN and SE are increasing linearly with the number of operations. Thus, the costs for the two architectural approaches may be described by the following equations.

$$Cost_{SN}^{read} = z \cdot a_{SN}^{read} + b_{SN}^{read} \tag{9.1}$$

$$Cost_{SE}^{read} = z \cdot a_{SE}^{read} + b_{SE}^{read} \tag{9.2}$$

The z denotes the number of operations.[2] The constants a_{SN}^{read} and a_{SE}^{read} are the costs associated with performing one additional read operation. The constants b_{SN}^{read} and b_{SE}^{read} are the costs that are not dependent on the number of operations. Our goal is to find the cost ratio that satisfies

$$a_{SN}^{read} = a_{SE}^{read}$$

That is, a ratio between the cost for message passing and the cost for synchronization that makes the slopes of the curves representing the total costs of running a read transaction on SN and SE equal. In order to do this, we must establish the costs associated with running an additional operation on SN and SE.

[2]The reason that z is used in stead of the more common variable x, is to ensure that it is not mixed up with the x of the x-axis in the graph presented in figure 9.5.

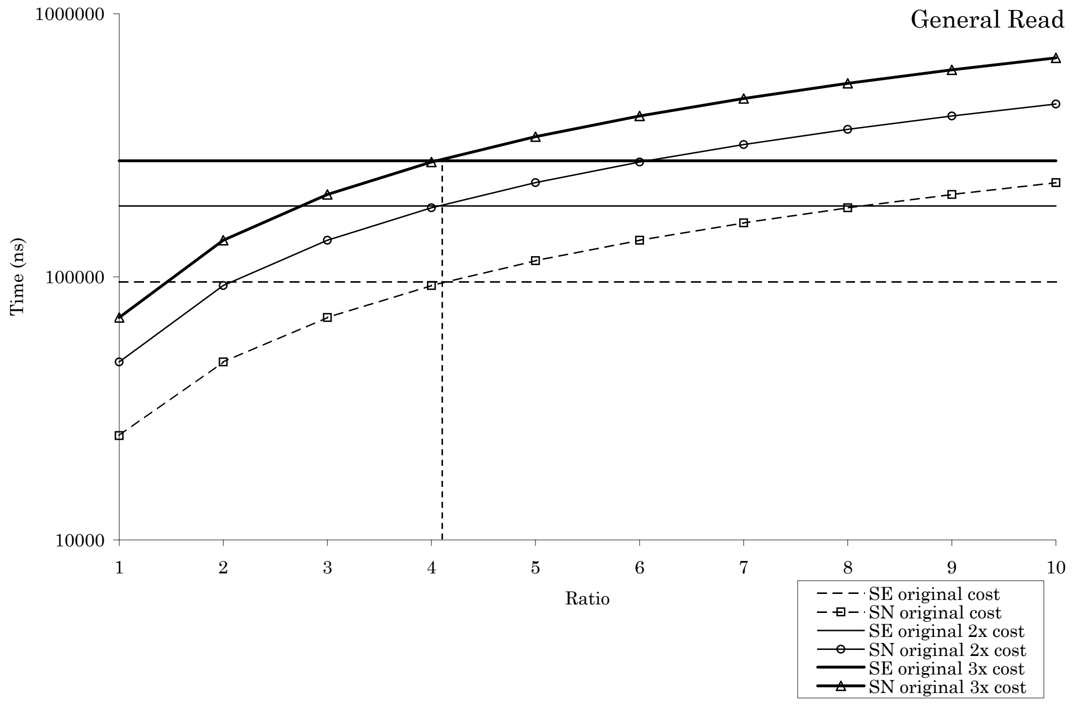


Figure 9.5: Varying the cost ratio for a general read transaction. There are three horizontal curves in the graph. These curves illustrate the costs associated with running an additional read operation in SE when the cost for synchronization is the original cost, two times the original cost and three times the original cost. The x-axis represents the ratio between the cost for message passing and the cost for synchronization. There are three pairs of curves. These are illustrated with dashed, solid and bold lines. The arched curves represent the costs associated with running an additional read operation in SN. The costs used for message passing when creating these curves are x times the cost used for synchronization in the matching SE curve. All pairs of curves intersect at $x \approx 4$. This suggests that if the cost for message passing is 4 times the cost for synchronization, SN and SE perform equally good when increasing the number of read operations.

The cost associated with performing an additional read operation in SN is presented in table 9.7. These figures are taken from section 7.5. The original cost for message passing is used in the table.

This makes a total of

$$140ns + 1092ns + 32\,457ns + 32\,457ns + 1092ns = 67\,238ns$$

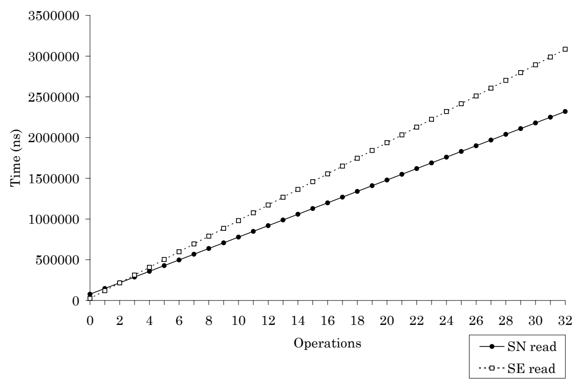
According to section 8.5, the additional costs imposed on SE by an additional read operation are as listed in table 9.8.

This makes a total of

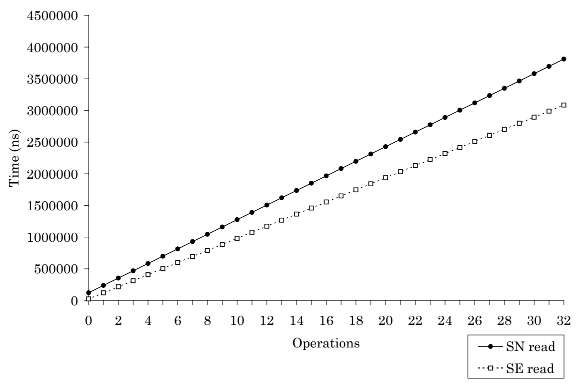
$$Task_1rest + Task_2 = 95\,749ns$$

It is clearly more expensive to perform an additional read operation in the SE case than in the SN case when using original costs.

The graph illustrated in figure 9.5 represents the cost of running an additional read operation in SN and SE. As for the graphs presented in the two previous sections, the ratio between the cost for message passing in SN and the cost for synchronization in SE is varied. The x-axis represents this ratio. There are three horizontal curves in the graph. These curves represent



(a)



(b)

Figure 9.6: Varying the cost ratio for the general read transaction. Graph (a) illustrates the case where the cost for message passing is 3 times the cost for synchronization. Graph (b) illustrates the case where the cost for message passing is 5 times the cost for synchronization. The difference in relative performance is clear.

the costs associated with running an additional read operation on SE when the cost for synchronization is the original cost, two times the original cost and three times the original cost, respectively.

All pairs of curves in the graph intersect at $x \approx 4.1$. Following the lines of the previous sections, this may also be seen by examining the additional number of times messages are sent or received for an extra read operation in SN, and the additional number of times semaphores are set or released for an extra read operation in SE. According to table 9.7 this two in the case of SN. For SE, table 9.8 tells us that the number of additional sets and releases of semaphores is 8. The ratio between these two numbers is

$$\frac{8}{2} = 4$$

This means that when the cost for message passing is 4 times the cost for synchronization, an additional read operation imposes approximately equal additional costs to SN and SE. Given that the additional costs are equal for SN and SE, this yields

$$\lim_{m,s \to \infty} \frac{m}{s} = 4$$

where the cost for message passing is denoted m and the cost for synchronization is denoted s.

If the cost for message passing is greater than 4 times the cost for synchronization, SE will perform best when increasing the number of read operations. On the other hand, if message passing is cheaper than 4 times the cost for synchronization, SN is preferable. This ratio is valid for all number of operations large enough to wipe out the impact of the initial costs, b_{SN}^{read} and b_{SE}^{read}. This is illustrated in figure 9.6. The graph shown in figure 9.6(a) illustrates the case where the cost for message passing is 3 times the cost for synchronization. In this scenario SN clearly performs better. Figure 9.6(b), on the other hand, illustrates the case where the cost for message passing is 5 times the cost for synchronization. In this scenario, SE is preferable.

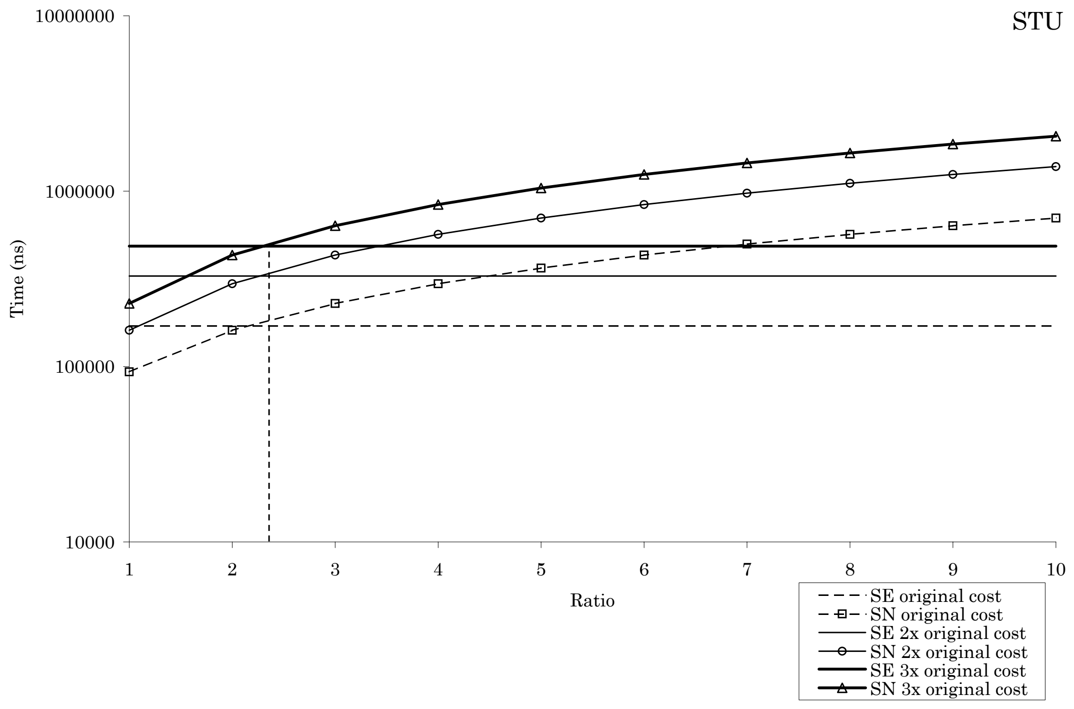


Figure 9.7: Varying the cost ratio for the STU transaction. There are three horizontal curves in the graph. These curves illustrate the costs associated with running the STU transaction on SE when the cost for synchronization is the original cost, two times the original cost and three times the original cost. The x-axis represents the ratio between the cost for message passing and the cost for synchronization. There are three pairs of curves. These are illustrated with dashed, solid and bold lines. The arched curves represent the costs associated with running the STU transaction on SN. The costs used for message passing when creating this curves are x times the cost used for synchronization in the matching SE curve. All pairs of curves intersect at $x = 2.33$. This suggests that if the cost for message passing is 2.33 times the cost for synchronization, SN and SE perform equally good when running STU transactions.

9.4.2 Varying the cost ratio for update transactions

In this section we perform a sensitivity analysis by varying the ratio between the costs for message passing in SN and synchronization in SE. This section is following the structure of section 9.4.1, but we now perform the analysis for the update transactions, STU and CTU. Also, an analysis for a general update transaction is performed. This is a transaction where the number of operations is not predetermined. As explained in section 9.4.1, our mission is to establish a ratio that marks a crossing in the performance of SN and SE. The following sections present an analysis for the transactions described above.

The STU transaction

This section presents an analysis based on the STU transaction. The ratio between the cost for message passing and the cost for synchronization is varied. Because a similar analysis is performed and thoroughly explained in section 9.4.1, this section briefly summarizes the results.

In the graph presented in figure 9.7, there are three horizontal curves. These curves illustrate the cost of executing the STU transaction on SE when the cost for synchronization is the

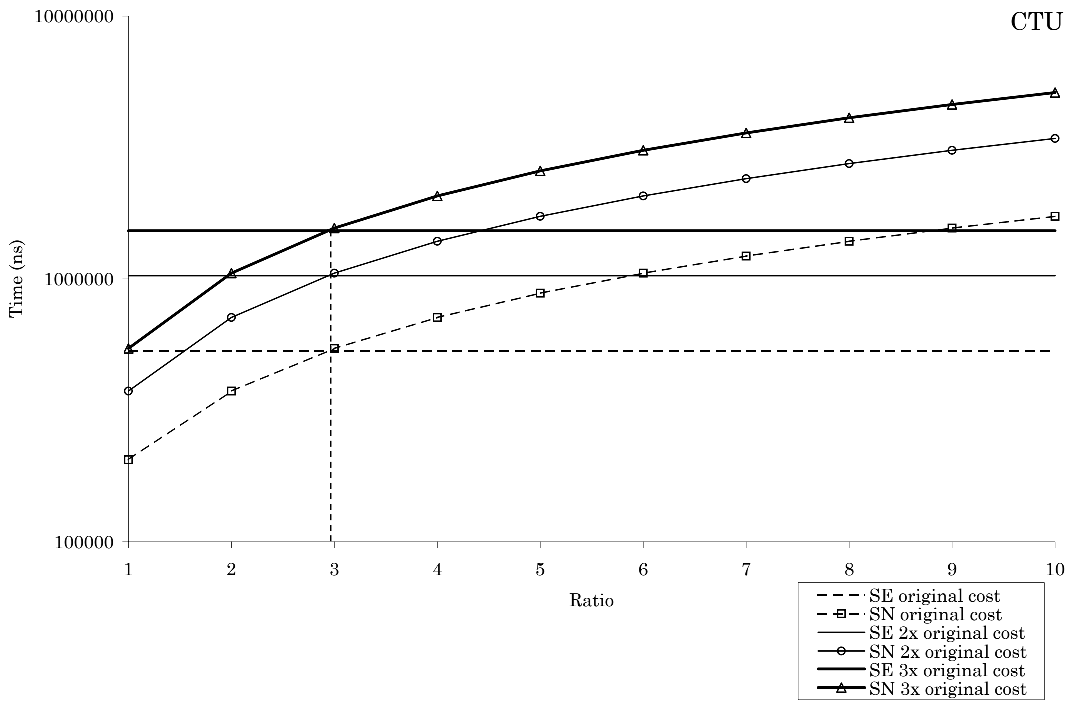


Figure 9.8: Varying the cost ratio for the CTU transaction. There are three horizontal curves in the graph. These curves illustrate the costs associated with running the CTU transaction on SE when the cost for synchronization is the original cost, two times the original cost and three times the original cost. The x-axis represents the ratio between the cost for message passing and the cost for synchronization. There are three pairs of curves. These are illustrated with dashed, solid and bold lines. The arched curves represent the costs associated with running the CTU transaction on SN. The costs used for message passing when creating this curves are x times the cost used for synchronization in the matching SE curve. All pairs of curves intersect at $x = 2.93$. This suggests that if the cost for message passing is 2.93 times the cost for synchronization, SN and SE perform equally good when running CTU transactions.

original cost, two times the original cost and three times the original cost, respectively. The three remaining curves illustrate the cost of executing the STR transaction on SN in the different cases, as explained thoroughly in section 9.4.1. The x-axis represents the ratio between the cost for message passing and the cost for synchronization. When running the STR transaction, the curves intersect at approximately $x = 2.5$. However, because the STU transaction is an update transaction, this point of intersection is shifted somewhat to the left. The ratio that marks a crossing between the performance of SN and SE is approximately $x \approx 2.3$.

We may also decide this crossing by comparing the number of times messages are sent or received in SN to the number of times semaphores are set or released in SE, during the execution of the STU transaction. According to sections 7.2 and 8.2 these numbers are 6 and 14, respectively. Thus, the ratio is

$$\frac{14}{6} = 2.33$$

For the same reasons as explained in section 9.4.2, given that the approaches perform equally good, this yields

$$\lim_{m,s\to\infty} \frac{m}{s} = 2.33$$

Task	**Description**	**Cost** (ns)
NOC Task$_2$	Decide participants	140
NOC Task$_3$	Build operation message	1092
	Send operation message	32 457
NOC Task$_4$	Receive *ready* message	32 457
	Interpret *ready* message	1092
NOC Task$_6$	Build *commit* message	1092
	Send commit message	32 457

Table 9.9: The additional cost for an additional update operation in SN

where m is the cost for message passing and s is the cost for synchronization. This means that if the cost for message passing is greater than 2.33 times the cost for synchronization, SE will perform best for a system running STU transactions. On the other hand, if the cost for message passing is less than 2.33 times the cost for synchronization, SN is preferable.

The CTU transaction

This section presents a sensitivity analysis based on the CTU transaction. As for the previous section, the ratio between the cost for message passing and the cost for synchronization is varied. This is illustrated in figure 9.8.

The graph in the figure contains three horizontal curves. These illustrate the cost of executing the CTU transaction on SE when the cost for message passing is equal the original cost of synchronization, two times the original cost and three times the original cost. The three remaining curves illustrate the cost of executing the CTU transaction on SN in the different cases. The x-axis represents the ratio between the cost message passing and the cost synchronization.

All the pairs of curves in the graph intersect at approximately $x \approx 3$. This suggests that when the ratio between the costs is approximately 3, SN and SE perform equally good. As explained earlier, another way to see this is to examine the number of times messages are sent and received in SN and the number of times semaphores are set or released in SE during the execution of the CTU transaction. According to sections 7.4 and 8.4, these number are 15 and 44,respectively. The ratio thus is

$$\frac{44}{15} = 2.93$$

For the same reasons as explained in section 9.4.2, given that the approaches perform equally good, this yields

$$\lim_{m,s\to\infty} \frac{m}{s} = 2.93$$

where m is the cost for message passing and s is the cost for synchronization. This means that if the cost for message passing is greater than 2.33 times the cost for synchronization, SE will perform best for a system running STU transactions. On the other hand, if the cost for message passing is less than 2.33 times the cost for synchronization, SN is preferable.

General update transaction

This section presents an analysis based on a general update transaction. The purpose of this section is to establish a ratio between the cost for message passing and the cost for synchronization that can be used to tell whether SN or SE yields best results regardless of how many

Task	**Description**	**Cost** (ns)
$Task_1rest$	Set semaphore	11 295
	Set file dictionary lock	186
	Release semaphore	11 323
	Access dictionary	140
	Set semaphore	11 295
	Release file dictionary lock	186
	Release semaphore	11 323
$Task_2$	Set semaphore	11 295
	Set lock on record	186
	Release semaphore	11 323
	Set semaphore	11 295
	Read index record from index root in L1 cache	36
	Read first level index record	753
	Read second level index record	2351
	Read data record	1439
	Write data record	84
	Release semaphore	11 323
	Build log post	1092
	Check transaction context	93
	Set semaphore	11 295
	Write log post	1120
	Release semaphore	11 323

Table 9.10: The additional cost for an additional update operation in SE

operations that are added to the transaction. This section is therefore analogous to section 9.4.2 which presents an analysis for the general read transaction.

As illustrated by the graphs in figures 9.1 and 9.2 in section 9.3, the costs of executing an update operation on SN and SE are increasing linearly with the number of operations. The costs for the two architectural approaches may therefore be described by two equations similar to equations 9.1 and 9.2 in section 9.4.1.

$$Cost_{SN}^{update} = z \cdot a_{SN}^{update} + b_{SN}^{update} \tag{9.3}$$

$$Cost_{SE}^{update} = z \cdot a_{SE}^{update} + b_{SE}^{update} \tag{9.4}$$

The z denotes the number of operations. The constants a_{SN}^{update} and a_{SE}^{update} are the costs associated with performing an additional update operation. The constants b_{SN}^{update} and b_{SE}^{update} are the parts of the total costs that are not depending on the number of operations. As described for the general read transaction in section 9.4.1, our goal is to find a ratio between the cost for message passing and the cost for synchronization that satisfies

$$a_{SN}^{update} = a_{SE}^{update}$$

To find this ratio, it is necessary to examine the extra work imposed by an additional update operation in SN and SE. According to section 7.4, the extra work imposed on SN is as described in table 9.9. The original cost is used for message passing.

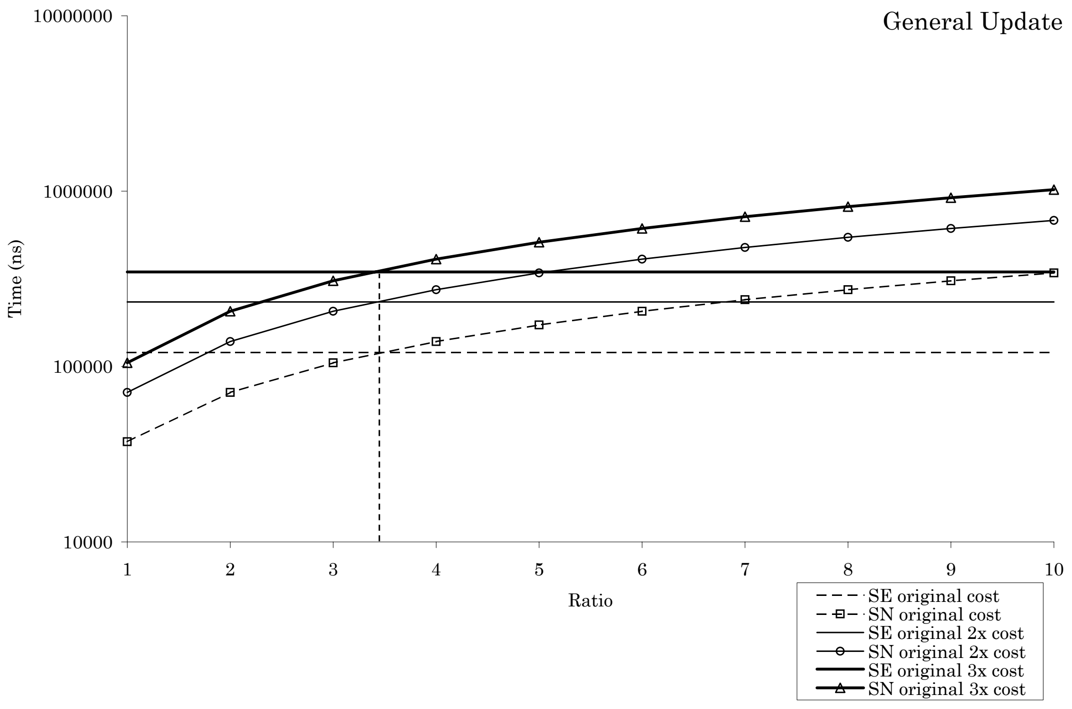


Figure 9.9: Varying the cost ratio for a general update transaction. There are three horizontal curves in the graph. These curves illustrate the costs associated with running an additional update operation in SE when the cost for synchronization is the original cost, two times the original cost and three times the original cost. The x-axis represents the ratio between the cost for message passing and the cost for synchronization. There are three pairs of curves. These are illustrated with dashed, solid and bold lines. The arched curves represent the costs associated with running an additional update operation in SN. The costs used for message passing when creating these curves are x times the cost used for synchronization in the matching SE curve. All pairs of curves intersect at $x = 3.33$. This suggests that if the cost for message passing is 3.33 times the cost for synchronization, SN and SE perform equally good when increasing the number of update operations.

This makes a total of

$$140ns + 1092ns + 32\,457ns + 32\,457ns + 1092ns + 1092ns + 32\,457ns = 100\,787ns$$

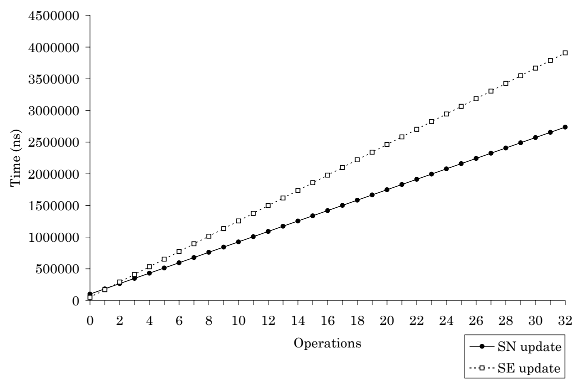
The costs associated with performing an additional update operation in SE are presented in table 9.10. These cost figures are taken from section 8.4.

This makes a total of

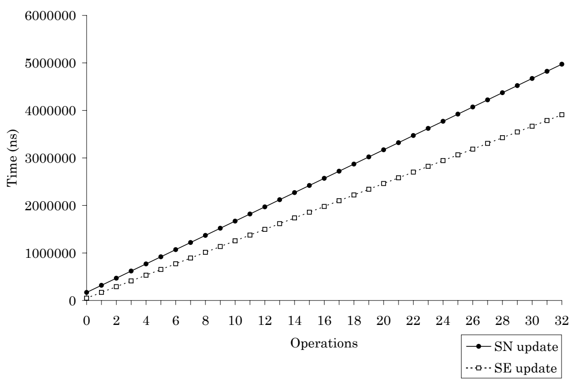
$$Task_1rest + Task_2 = 120\,756$$

When using original costs, it is considerably more expensive to perform an additional update operation in SE than in SN.

The graph in figure 9.9 represents the cost of running an additional operation for SN and SE. The cost ratio between the cost for message passing and the cost for synchronization is varied. There are three horizontal curves in the graph. These curves illustrate the cost of executing an additional operation on SE when the cost for synchronization is the original cost, two times the original cost and three times the original cost, respectively. The x-axis represents the ratio.



(a)



(b)

Figure 9.10: Varying the cost ratio for general update transaction. Graph (a) illustrates the case where the cost for message passing is 2.33 times the cost for synchronization. Graph (b) illustrates the case where the cost for message passing is 4.33 times the cost for synchronization. The difference in relative performance is clear.

All the pairs of curves in the graph intersect at approximately $x \approx 3.5$. This suggests that if the cost for message passing is 3 times the cost for synchronization, it is equally expensive to perform an additional update operation in SN and SE. Another way to see this is to examine the number of times messages are sent or received in SN and the number of times semaphores are set or released in SE for an additional update operation. According to tables 9.9 and 9.10 these numbers are 3 and 10 respectively. The ratio between these two numbers is

$$\frac{10}{3} = 3.33$$

Given that the additional costs are equal for SN and SE, this yields

$$\lim_{m,s \to \infty} \frac{m}{s} = 3.33$$

where m is the cost for message passing and s is the cost for synchronization.

If the cost for message passing is greater than 3.33 times the cost for synchronization, SE will perform best when increasing the number of update operations. If the cost for message passing is less than 3.33 times the cost for synchronization, SN is preferable. As for the general update transaction presented in section 9.4.1, this is valid for all numbers of operations large enough to wipe out the impact of the initial costs b_{SN}^{update} and b_{SE}^{update}. This is illustrated in figure 9.10. The graph shown in figure 9.10(a) illustrates the case where the cost for message passing is 2.33 times the cost for synchronization. In this scenario SN clearly performs better. Figure 9.10(b), on the other hand, illustrates the case where the cost for message passing is 4.33 times the cost for synchronization. In this scenario, SE is preferable.

9.4.3 Varying the number of process pairs

This section provides a sensitivity analysis by varying the number of process pairs. First, we examine the SN case, then SE.

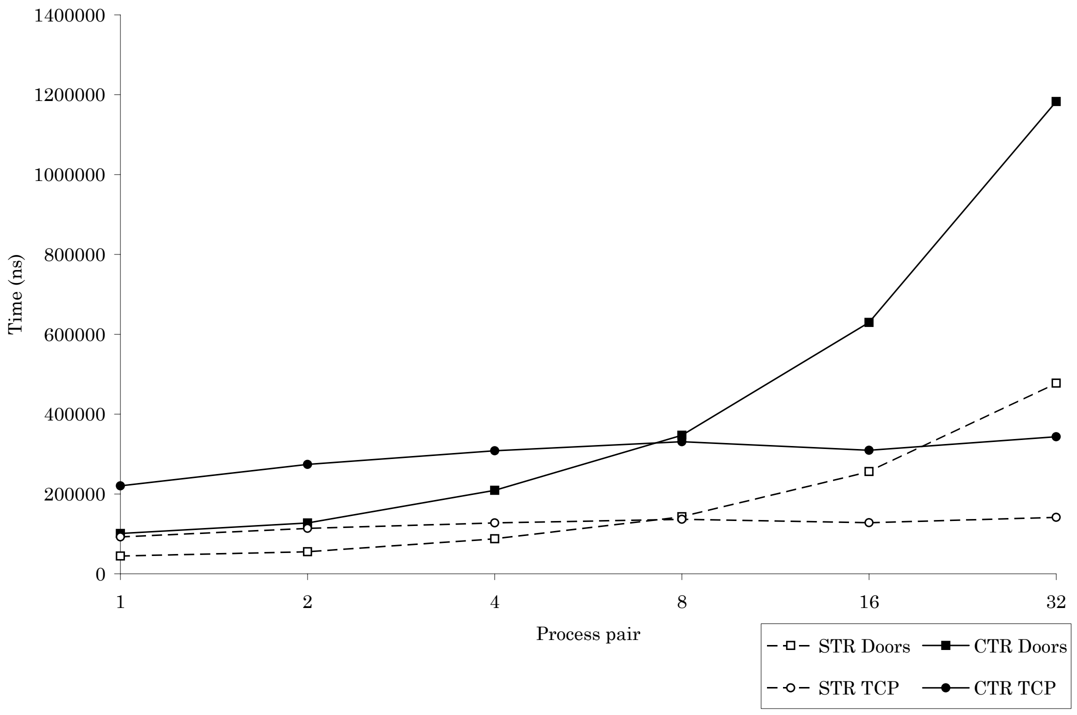


Figure 9.11: Varying the process pairs for read transactions in SN. The graph has four curves, representing the cost curves for the read transactions applied on both message passing techniques. The dashed curves represent the STR transaction, the solid curves the CTR transaction. The cost curves utilizing Solaris Doors is denoted with squares, TCP/IP is denoted with circles. Considering the STR and CTR transactions using Solaris Doors, they are both clearly cheaper compared to the TCP/IP equivalent for a number of process pairs < 8. However, when the numbers of process pairs is > 8, TCP/IP scales very well. Reaching 32 process pairs, TCP/IP outperforms Solaris Doors by a a factor of ≈ 3.45 in the CTR case, and ≈ 3.37 in the STR case.

Varying the number of process pairs in SN

This section expands on the most significant cost in SN, namely message passing. When we in section 6.1 at page 63 established the costs to use in the calculation chapters, we had two candidate techniques to choose from with regards to message passing, namely TCP/IP and Solaris Doors. After settling for a message size of 64B, we argued that in order to fully take advantage of the system, all nodes should at any time participate in at least one transaction. In other words, we considered the cost of 32 process pairs. Because the TCP/IP candidate was far cheaper in this case, we settled for this technique. This section seeks to examine which technique that is preferable when the number of process pairs is varied. Indeed, because Niagara has eight cores, it could also be interesting to examine which technique is cheaper when only one process is applied per core (compared to four processes per core).

Figure 9.11 illustrates how the total costs changes when the number of process pairs involved vary. The graph has four curves, representing the cost curves for the read transactions applied on both message passing techniques. The dashed curves represent the STR transaction, the solid curves represent the CTR transaction. The cost curves utilizing Solaris Doors is denoted with squares, TCP/IP is denoted with circles. The update case is illustrated in figure 9.12. Albeit the costs are higher in the latter graph, the graphs are practically identical. We will therefore combine the analysis of these graphs in the following.

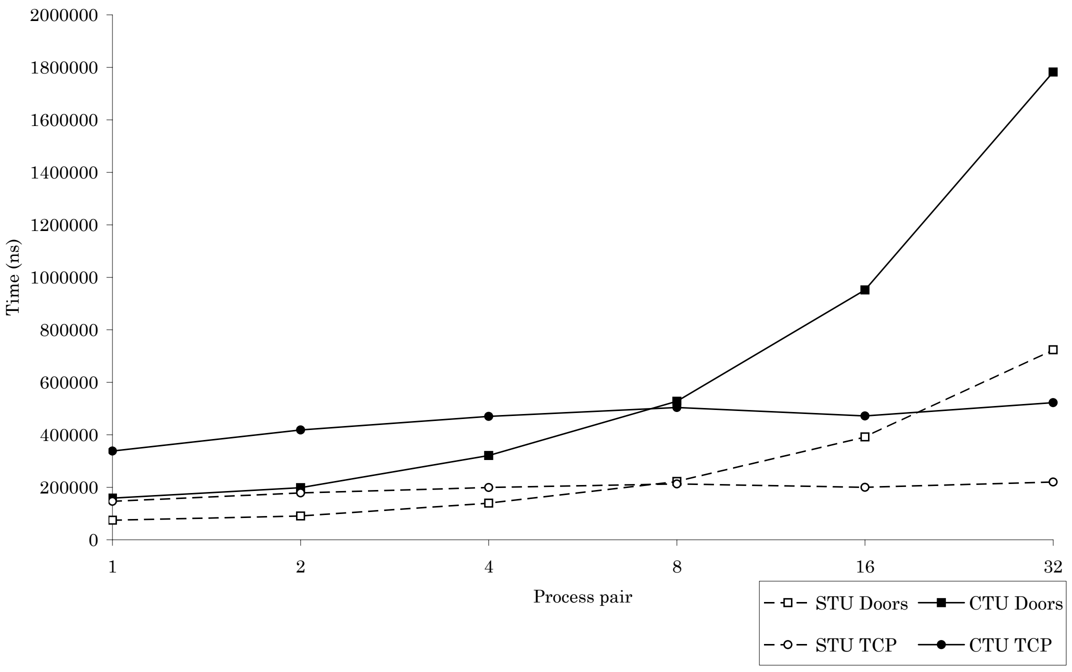


Figure 9.12: Varying the process pairs for update transactions in SN. As the previous graph (figure 9.11), the graph has four curves, representing the cost curves for the update transactions applied on both message passing techniques. The dashed curves represent the STU transaction, the solid curves the CTU transaction. The cost curves utilizing Solaris Doors is denoted with squares, TCP/IP is denoted with circles. Considering the STU and CTU transactions using Solaris Doors, they are both clearly cheaper compared to the TCP/IP equivalent for a number of process pairs < 8. However, when the numbers of process pairs is > 8, TCP/IP scales very well. Reaching 32 process pairs, TCP/IP outperforms Solaris Doors by a a factor of ≈ 3.41 in the CTU case, and ≈ 3.29 in the STU case.

Figure 9.11 illustrates that although message passing using TCP/IP has a significantly higher costs for a low number of process pairs, it scales very well, as there is literally no growth with an increasing number of process pairs. In contrast, Solaris Doors starts off much cheaper, but it does not scale well compared to TCP/IP. Indeed, when the number of process pairs is below 8, it is apparent that Solaris Doors is the cheaper technique. At 8 process pairs, however, there is practically no difference between the techniques. This is interesting, as noted above, in the case where there are one process per core. For all process pairs above 8, TCP/IP outperforms Solaris Doors, in the read as well as the update case. Inspecting the costs for each transaction, we find that when reaching 32 process pairs, Solaris Doors is

$$\frac{CTR^{32PP}_{Doors}}{CTR^{32PP}_{TCP}} = \frac{1\ 183\ 038ns}{343\ 389ns} \approx 3.45$$

times as expensive as TCP/IP for the CTR case, and

$$\frac{STR^{32PP}_{Doors}}{STR^{32PP}_{TCP}} = \frac{477\ 533ns}{141\ 673ns} \approx 3.37$$

times as expensive as TCP/IP for the STR case. For the update case, we find indeed similar results. Solaris Doors is in the update case

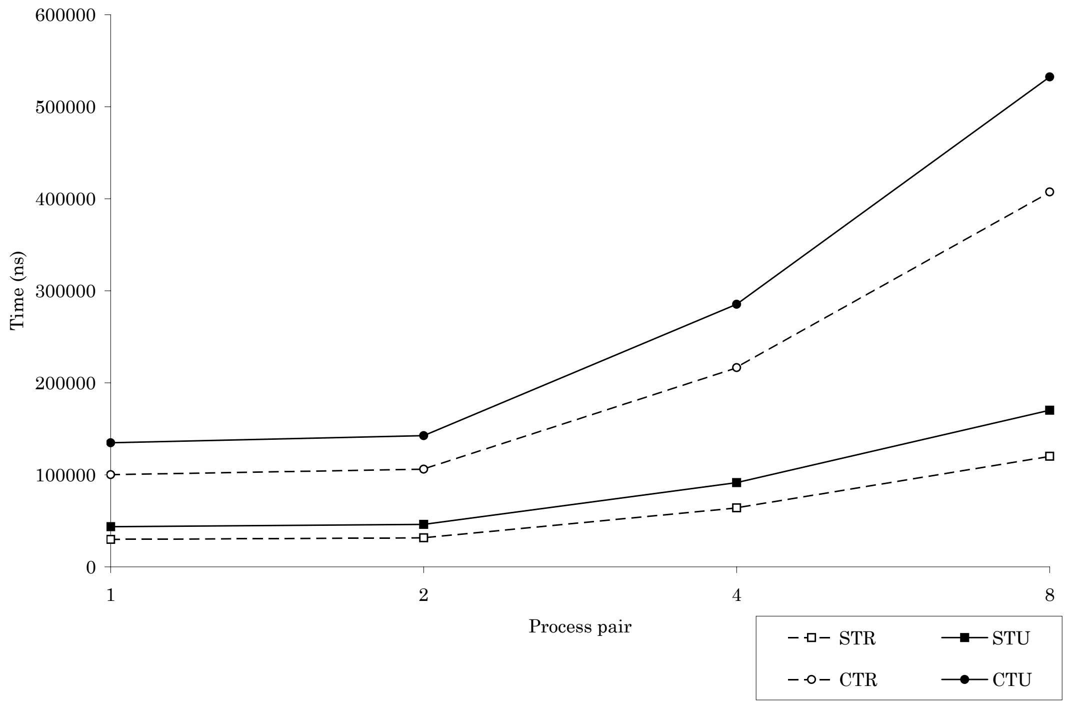


Figure 9.13: Varying the number of process pairs for transactions in SE. There are four curves in the graph. The dashed curves denote the read transactions, the solid curves denote the update transactions. The curves denoted with a square is the complex transactions, whereas a circle denotes the simple transactions. The graph suggests no surprising effects. For fewer process pairs, the total costs are increasingly smaller.

$$\frac{CTU^{32PP}_{Doors}}{CTU^{32PP}_{TCP}} = \frac{1\,782\,126ns}{522\,651ns} \approx 3.41$$

times as expensive as TCP/IP for the CTU case, and

$$\frac{STU^{32PP}_{Doors}}{STU^{32PP}_{TCP}} = \frac{724\,078ns}{220\,2885ns} \approx 3.29$$

times as expensive as TCP/IP for the STU case.

As mentioned in section 5.2 at page 54, we suspect that Solaris Doors does not scale well due to heavy thrashing.

Varying the number of process pairs in SE

This section expands on the most significant cost in SE, namely synchronization. Although a similar analysis to the one in the previous section could yield interesting results, the only inter-process synchronization technique implemented is System V Semaphores. To plot the inter-process System V Semaphores against the much lighter intra-process POSIX Mutexes makes no sense, because the latter benchmark was not implemented to handle inter-process synchronization. [3]

However, it is interesting to note how the costs change when the number of simultaneous process pairs involved under the SE regime is varied. We argued in section 6.1.6 at page 65

[3]However, to implement POSIX Mutexes for inter-process synchronization is one of our suggestions for future work in chapter 11.

that when assuming 32 nodes executing one CTU transaction each at the same time, we can expect 11 processes to perform operations on a semaphore at any given time. Because we did not sample for 11 processes, we settled for 8 process pairs over 16, to account for the simpler transactions. However, this implies that there is no need to examine more than 8 process pairs, because the system will not exceed 32 nodes.

Figure 9.13 shows how the cost vary when the number of process pairs vary from 1 to 8. There are four curves in the graph. The dashed curves denote the read transactions, the solid curves denote the update transactions. The curves denoted with a square is the complex transactions, whereas a circle denotes the simple transactions. The graph suggests no surprising effects. For fewer process pairs, the total costs are increasingly smaller.

9.5 Conclusion

The conclusion introduced in this section presents a brief summary of the tendencies discovered during our research. These tendencies should not be perceived as definite truths, but merely as directions of trends.

Several trends have been discovered during our research. The following sections give a brief summary of each trend.

9.5.1 The relative difference when using original costs

This section describes the relative difference between SN and SE when original cost figures are used. These cost figures are described in chapter 6.

The SE DBMS architecture is cheaper when running simple transactions

During the analysis it became evident that when using the original cost estimates, SE wins by a head compared to SN when running simple transactions. The administrative overhead imposed by the NOC component enables SN to parallel complex transaction. However, it works against its purposes when executing transactions only containing one operation. This was also the case in our previous work. A possible solution which is yet to be properly introduced in chapter 11, is to push the NOC responsibility to the node actually performing the operation.

The SN DBMS architecture is cheaper when running complex transactions

During the analysis we discovered that the overhead imposed by the NOC in SN pays off when executing complex transactions. Because different operations are assumed to go to different nodes, all operations are performed in full parallel, and only the additional work forced on the NOC adds to the total time used to execute the transactions.

9.5.2 The results are sensitive to the relative difference in costs

In sections 9.4.1 and 9.4.2 we varied the ratio between the cost for message passing and the cost for synchronization. It became evident that the results for the two architectural approaches indeed are dependent on the relationship between these two costs. An analysis was performed for each transaction type including a general read transaction and a general update transaction. The two latter transaction types are transactions were the number of operations are not predetermined.

During the analysis of the general read and update transactions, we increased the number of operations involved in a transaction to infinity. As the number of operations involved increases, the most dominant costs (message passing in SN and synchronization in SE) become increasingly more dominant compared to the other costs. Eventually, the other costs virtually vanish. At this point, a ratio between the costs emerges. For the general read transaction, this ratio is 4. For the general update transaction, the ratio is 3.33.

In the following we illustrate for the general read transaction, but the same holds for the general update transaction. For the read case, this ratio suggests that if the cost of message passing is approximately 4 times the cost of synchronization, SN and SE will perform equally well. If the cost of message passing is less than 4 times the cost of synchronizing, SN is the better architectural alternative. Similarly, if the cost of message passing is more than 4 times the cost of synchronizing, SE is preferable.

Several of the cost figures used in the calculations presented in this report are not benchmarked. For a general DBMS, they may therefore not be sufficiently accurate. However, for the assumed DBMS we have investigated, message passing and synchronization should indeed be the most significant costs. Therefore, given that the DBMS follows the same lines, we believe that the ratios presented above may be used as a mild rule of thumb for choosing between the SN or SE memory architecture. If the costs associated with message passing and synchronization are known, the ratios presented above may be used to get a hint of which architecture that will yield the better results.

Of course, for systems having design solutions differing vastly from the solutions presented in chapter 2, the cost ratios will also differ.

9.5.3 The gain of parallelism

As explained in the previous section, SN profits by its ability to parallel the execution of complex transactions. The paralleling possibilities of SN allows the approach to have a high cost for message passing and still be the better of the two candidates when executing complex transaction. There is of course, as explained earlier, a crossing between the performances of SN and SE if the cost of message passing is too high. The location of this crossing is influenced by the level of parallelism.

9.5.4 Niagara performs poorly for single threaded tasks

Compared to traditional processors, the Niagara processor has many slower cores opposed to a few faster cores. At the face of being power-efficient, this could be seen as one of the disadvantages of the Niagara architecture. Because the cores run at a low frequency, executing only a single instruction per cycle, Niagara performs poorly for single threaded tasks. This became evident when running benchmarks 3, 4 and 5. These benchmarks were executed at both the Sun Fire T1000 Server and a Sun Fire V890 Server. On average the V890 Server performed almost one order of magnitude better than the T1000. On the other hand, Sun Microsystems does not promote Niagara to excel at single threaded programs. The key to unlocking the potential of Niagara is not to enforce single threaded execution, but to adapt programs to the highly threaded environment.

Chapter 10

A hybrid DBMS architecture for Niagara

As we have seen from the analysis, both the SN and SE architectural approaches have strengths and weaknesses. It is perhaps possible to combine the attractive properties of both approaches. The object of this chapter is to introduce such a Niagara tailored hybrid memory architecture.

This chapter first presents a general hybrid architecture suggested by [NZT96] in section 10.1. This architecture was mentioned in our previous work, and is used as an inspiring basis for the hybrid architecture we present in section 10.2. Although a proper analysis of the hybrid following the lines of this report would be ideal, we do not have the possibility to perform a complete second analysis within the scope of this research. We therefore present a selection of scenarios to illustrate that further work in this area is of great interest. These scenarios are presented in section 10.3.

10.1 A general hybrid architecture

Because both SN and SE have their benefits and drawbacks, it is natural to believe that one might achieve interesting results by combining the two approaches. The article [NZT96] presents such an architecture. This two level hierarchic hybrid architecture is illustrated in figure 10.1.

The figure illustrates clusters of SE systems combined in an SN manner. Each SE system contains several processors which share memory and disk. The processors are connected through a bus. Several of these systems are connected via an interconnection network, forming an SN system. According to [NZT96], this hybrid architecture combine the advantages of SE and SN, and compensate for their drawbacks.

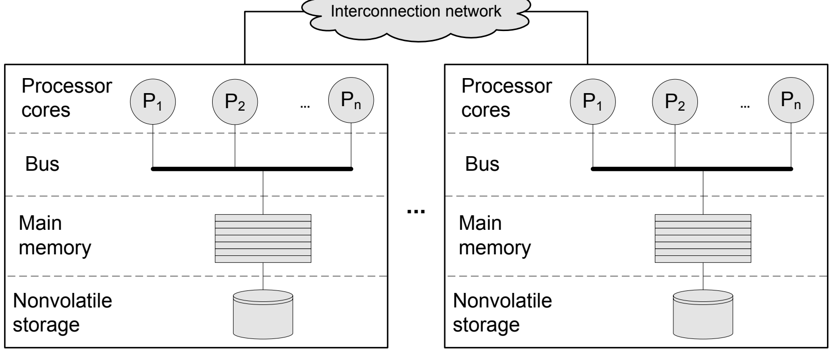


Figure 10.1: A general hybrid architecture [NZT96].

10.2 A Niagara tailored hybrid architecture

Our motivation for suggesting a Niagara tailored hybrid architecture is based on the observation that SN scales better than SE for increasingly complex transactions. Although SN benefits greatly from its possibility to execute operations in parallel, message passing is expensive compared to the isolated cost for synchronization. At the face of introducing both costs, the hybrid architecture suggested in the following seeks to minimize both the amount of message passing and synchronization.

It is in the nature of tailored architectures that it takes advantage of the underlying hardware architecture. And indeed, the general hybrid architecture suggested in figure 10.1 indeed resembles the Niagara hardware architecture[1].

Comparing the hardware architecture to the hybrid architecture, we make the following observations. Intra core, each of the eight cores in Niagara is similar to the SE sub systems that shares memory and disk. Inter core, the interconnection network is similar to the interconnect between the cores on the Niagara chip. It is therefore interesting to use the cores as SE-like sub systems, while an SN-like policy is enforced between the cores. The transaction operations are executed intra core, whereas all communication is inter core. In the following, we expand on these observations.

10.2.1 Inter core communication

Niagara utilizes eight cores which can execute four native threads each. This gives a total of 32 nodes in the DBMS model we have used in our calculations. However, if we turn our attention to the eight cores, it becomes apparent that the hardware configuration of Niagara could yield an alternative way of realizing a DBMS.

Instead of naming every single thread a node, an entire core can be run as a node. Instead of running 32 nodes, this segmentation results in 8 nodes, each spanning their own subset of the database in an SN manner. Each core will have four threads working in an SE-like fashion, which will be discussed in section 10.2.2.

For the cases where a transaction needs to perform operations in subsets owned by other nodes, a message passing scheme between the cores must be present. The hardware configuration of Niagara is indeed prepared for inter core communication. Inter core communication may be realized via a tailored message passing protocol over the L2 cache [2].

As stated above, the message passing cost between nodes is indeed the most significant cost in the SN architectural approach. Although a message passing protocol that takes advantage of the Niagara hardware architecture most likely would reduce the cost associated with message passing dramatically, it would probably continue to be the most significant cost. However, because the database is split into 8 subsets instead of 32, the message passing will occur less frequently in this architecture. In addition, it will most likely involve fewer nodes.

For a transaction touching subsets owned by other cores, the cores owning these subsets must be involved in the execution. When a transaction arrives at node, the NOC thread in the core forwards the operations that is regarding other subsets to the correct cores over the interconnection. Thus, communication between the cores only involves the NOCs of each core. Again, because there are eight cores opposed to 32 nodes, one might expect that the amount of messages drops dramatically compared to a strict SN scheme.

[1]For reference, this architecture is illustrated in figure 3.3 at page 25.

[2]This is one of our suggestions for further work, see section 11.2.

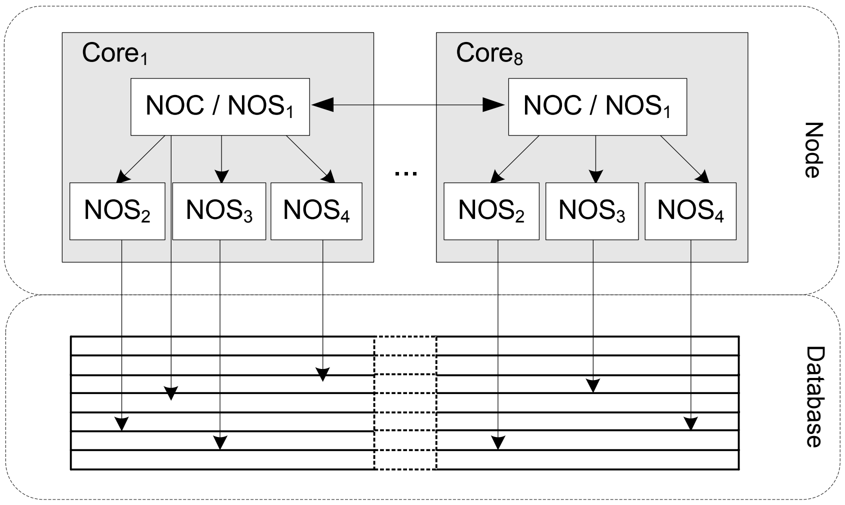


Figure 10.2: A Niagara tailored hybrid process architecture. The hybrid has 8 nodes, each node is mapped to a Niagara core. Message passing is performed inter core in a SN manner. Each node has four native processes, mapped to the four native threads within a Niagara core. Within each node, one of the processes has node controller responsibility, distributing operations between all four threads, including itself. All executions are performed intra core. All involved processes participates in a distributed commit. If a transaction operation touches a subset owned by another node, the NOC of that node exchanges operation and data messages with the NOC from the node where the transaction originated.

10.2.2 Intra core execution

Turning our attention to the core, each core has four concurrent processes. The threads within a core share the L1 cache, $\frac{1}{8}$ of the L2 cache and $\frac{1}{8}$ of the main memory. As with a strict SE scheme, these threads are bound to interfere with each other without synchronization. It is therefore apparent that some synchronization mechanism must be present. The hardware configuration states that the L1 cache is private to each core, but shared for all four threads within a core. The L1 cache has a low latency ($\approx 1ns$ access time), so synchronizing intra core at L1 is very attractive. This can be realized through a tailor-made synchronization scheme, perhaps with fundament in POSIX mutexes set for inter-process synchronization [3].

Intra-core, there are four synchronized threads with access to the same set of data. Although the core might resemble a strict SE scheme, it is desirable for a complex transaction touching only the subset owned by one node to be executed in parallel by the available threads. Therefore, as illustrated in figure 10.2, the core also utilize elements from SN.

Within each core, one of the threads has an SN-like node controller (NOC) responsibility, distributing operations between all four threads, including itself. In order to reduce congestion we assume that the NOC only distributes operations to itself if all other threads within the core are occupied. The gain of introducing parallelism comes with a cost. It is necessary to introduce some form of communication and distributed commit between the threads within the core. If it is desirable to avoid message passing within the core, parallel execution of complex transactions within the core is not possible. Lacking parallelism, the parts of a transaction executed at a core would be run single threaded. As this scheme would not use a default NOC within each core, all threads in all cores would have to pass messages to each other in order to

[3]This is one of our suggestions for further work, see section 11.1.

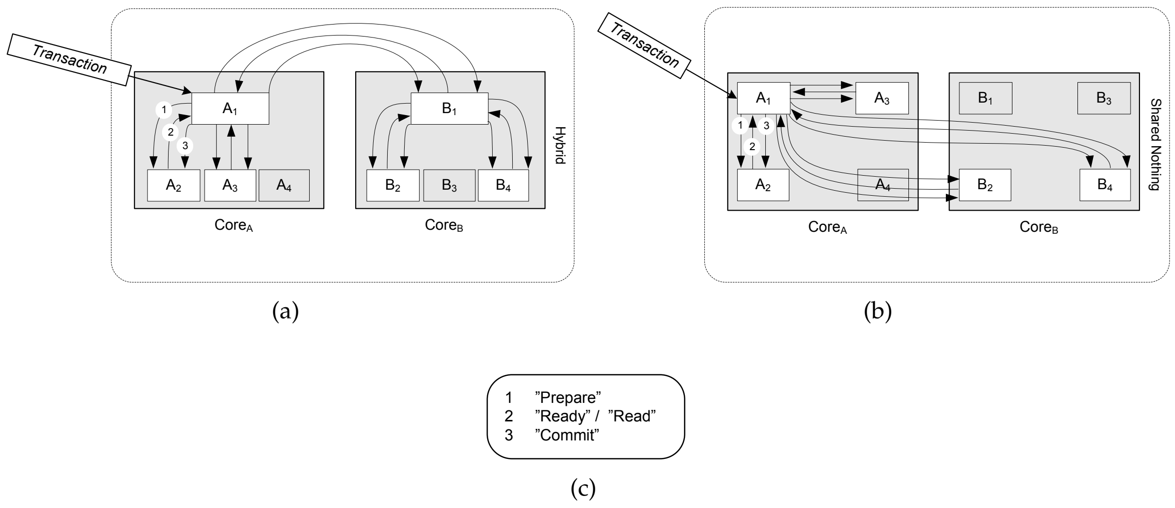


Figure 10.3: Scenario 1: The effect of message passing. Figure (a) illustrates the case when the CTU transaction is executed on the hybrid, whereas figure (b) illustrates the case when the CTU transaction is executed on SN. Figure (c) describes the content of each message. Clearly, more messages are sent in (a).

perform transactions spanning multiple subsets. This would not result in a reduced cost for message passing.

The introduction of this SN-like behavior has the advantage that the execution of an STU transaction and a CTU transaction will yield almost the same costs, given that a transaction operate within the same core. The only cost added will be the overhead imposed by the controller. This overhead will be very similar to that performed by the NOC described earlier in this report. However, because the communication as well as synchronization can be realized through L1 cache within a core, the associated cost will be significantly smaller.

In addition, we expect that this architecture will use less time on execution. We have previously calculated that $\approx \frac{1}{3}$ of the processes are operating on a given synchronization primitive at any given time. As described earlier in this report, a strict SE scheme yields 11 processes. As there are only four threads within each SE sub system, the percentage of the total transaction execution time spent on synchronization will obviously be less.

10.3 The tailored hybrid versus Shared Nothing

This section presents a brief comparison between the hybrid architecture and SN as described throughout the report. In the following we illustrate different scenarios of the distributions of operations when executing the CTU transaction on the two architectures. The comparison is based on the CTU transaction, because the execution of this transaction involves the highest number of nodes and messages (update transactions involve the additional *commit*-message). We give a thorough explanation of the figure semantics in the first scenario. The figures illustrating the subsequent scenarios share these semantics.

10.3.1 Scenario 1: The effect of message passing

Figure 10.3 illustrates the number of messages necessary to carry out the CTU transaction for both architectures according to the 2PC-*presumed commit* protocol[4] . The figure illustrates the case where a transaction contains operations distributed over two cores, A and B.

In the hybrid case (figure 10.3(a)), the transaction arrives at process A_1. This process has the NOC responsibility, and forwards the operations to the correct destinations. In this case, two of the operations touch tuples within the subset of core A. Therefore, two available processes, A_2 and A_3 receive the operations.

However, two operations need to perform updates on tuples residing in the subset owned by core B. A_1 forwards these operations to the process with NOC responsibility in core B, namely B_1. B_1, in turn, forwards the operations to two available processes in its core. The processes A_2, A_3, B_2 and B_4 now acts as NOSes in the transaction execution. Following the annotation from figure 10.3(c), a prepare message is piggybacked on all operation messages.

After the NOSes have performed the updates, they respond with a *ready*-message to the NOC in their core. After receiving the *ready*-messages from B_2 and B_4, B_1 responds to the initiating process A_1. After receiving a *ready*-message from all involved processes, A_1 issues a *commit*-message. This message propagates in the same manner as the previous messages.

When the CTU transaction is executed in SN (figure 10.3(b)), the same processes are involved as in the case of the hybrid, except for process B_1. A_1 has the NOC responsibility, and exchanges messages with all NOSes directly, opposed to the B_1 intermediary in the hybrid. The SN 2PC execution is described in detail in figure 2.4 at page 10.

It is apparent that the hybrid carries an additional overhead compared to SN. In this example, the number of messages is 15 in the hybrid case, whereas 12 messages are sent in the SN case.

As suggested in section 10.2.2, a tailored message passing protocol for intra core communication will significantly improve the performance of the hybrid. Indeed, most of the messages in the example are passed within the core. Because the processes within a core share L1 cache, message passing at this level is clearly tractable. However, it is apparent from figure 10.3(b) that also SN may greatly benefit from this protocol.

If we assume the cost of inter-core communication to be equal to the cost used for message passing in the calculations presented earlier, this type of communication is considerably more expensive than communication through L1 cache. As explained in chapter 9, the cost of message passing between nodes is large enough to wipe out the effect of all other costs. Therefore, we have to compare the number of messages passed between cores to see which approach that probably yields best results. It is important to keep in mind that the hybrid also carries the cost of synchronization. However, because all synchronization may be kept in L1 cache, we expect this cost to be in the same order of magnitude as the results obtained from the benchmarking of intra-process synchronization using POISX mutexes.

In the example presented in figure 10.3, 3 messages are passed between cores in the case of the hybrid. In the SN case, 6 messages are passed. Under the assumption that it is possible to realize intra-core communication through L1 cache, the hybrid appears to yield better results than SN when the operations of the CTU transaction are distributed in this fashion.

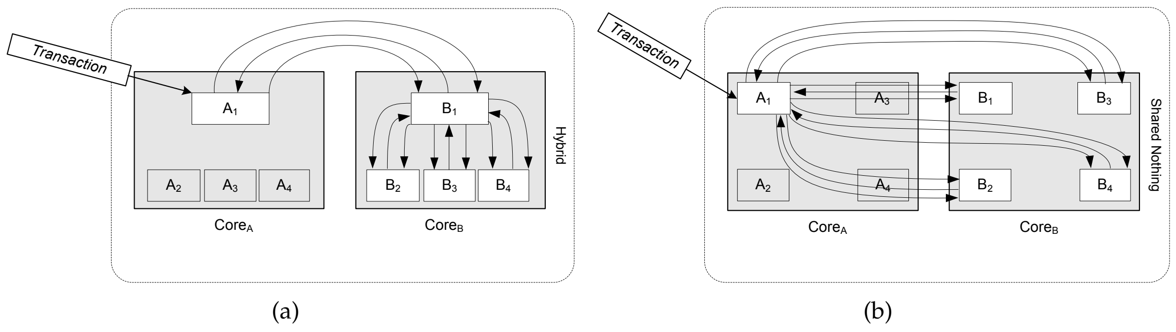


Figure 10.4: Scenario 2: The hierarchic advantage. Figure (a) illustrates the case when the CTU transaction is executed on the hybrid, whereas figure (b) illustrates the case when the CTU transaction is executed on SN.

10.3.2 Scenario 2: The hierarchic advantage

Figure 10.4 illustrates the scenario where the transaction received solely contains operations that touch tuples residing in subsets owned by another core.

For the hybrid case illustrated in figure 10.4(a), all message passing between the NOSes and the NOC is realized within the core. The reason is that the NOC in the core receiving the transaction (A_1) involves the NOC in the core where the tuples reside (B_1). There are only three messages passed between cores. These messages regard the distributed commit between the two NOCs.

In the SN case, as illustrated in figure 10.4(b), only one NOC, A_1, is involved in the execution. Because the receiving core does not own any of the tuples to be updated, all messages between the NOC and the NOSes is passed between the cores. For the CTU case, this results in 12 messages, all of which are between cores.

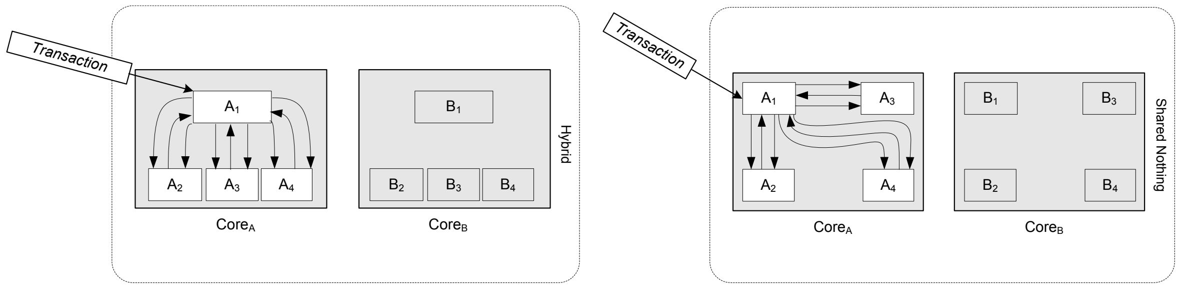
It is apparent that the NOC component in SN is more prone to congestion than the NOCs in the hybrid. However, this becomes even more evident for transactions containing more operations. For 32 operations evenly distributed between all cores, all processes have one operation to execute. In SN, this implies that 31 processes must interact with the NOC as NOSes. In other words, the NOC is involved with passing $31 \times 3 = 93$ messages. Although transactions with 32 operations are of rare breed, the NOC can easily be a bottleneck in such scenarios. In the hybrid, the NOC component in every core is involved in the execution. Each NOC is responsible for communicating with the NOSes within the core. Even though the hybrid way yields more messages, the number of messages passed by each NOC component is small compared to the single NOC in SN. In addition message passing within different cores is performed in parallel.

Although the total number of messages in the hybrid is higher than in SN, there are substantially fewer messages passed between cores. Given that message passing intra core is realized through L1 cache, the hybrid appears to outperform SN in this scenario.

10.3.3 Scenario 3: The disadvantage of clustered operations

Figure 10.5 illustrates the scenario where the transaction received solely contains operations that touch tuples residing in the subset owned by the receiving core. Hence, all messages

[4]2PC is described for SN in section 2.4.2.

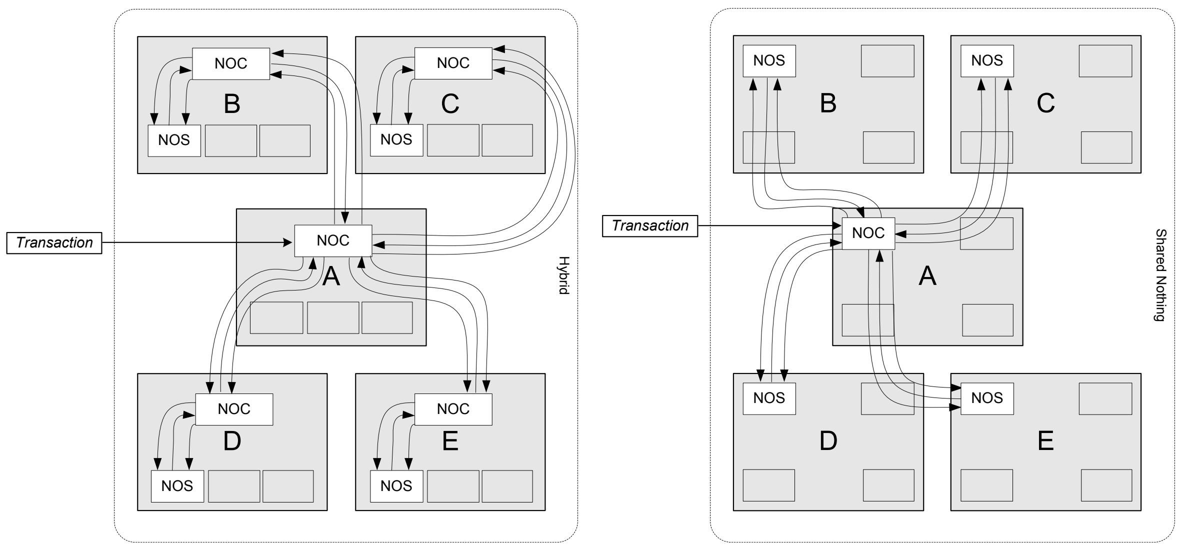


(a) (b)

Figure 10.5: Scenario 3: The disadvantage of clustered operations. Figure (a) illustrates the case when the CTU transaction is executed on the hybrid, whereas figure (b) illustrates the case when the CTU transaction is executed on SN.

between the NOC and the NOSes are passed within the core. This holds for both the hybrid and SN architectures, illustrated in figures 10.5(a) and 10.5(b), respectively. Because the cost for message passing therefore is equal, SN will perform better, due to the hybrid's need for synchronization.

10.3.4 Scenario 4: The disadvantage of evenly distributed operations



(a) (b)

Figure 10.6: Scenario 4: The disadvantage of evenly distributed operations. Figure (a) illustrates the case when the CTU transaction is executed on the hybrid, whereas figure (b) illustrates the case when the CTU transaction is executed on SN.

Figure 10.6 illustrates the scenario where the transaction received solely contains operations that touch tuples residing in subsets owned by distinct different cores. Thus, five cores are involved. For both the hybrid and the SN architectures, the number of messages passed between cores is 12. This is illustrated in figures 10.6(a) and 10.6(b), respectively.

The hierarchic nature of the hybrid architecture, implies however that the NOC component in each core owning a tuple involved in the transaction passes 3 additional messages with the NOS within the core. As stated in scenario 2, message passing within different cores is performed in parallel. In addition, the hybrid carries the cost of synchronization. This implies that SN yields better results following this scenario.

However, in a saturated system, the hybrid yields 8 process pairs that may communicate simultaneously between cores, opposed to 32 process pairs in SN. In this case, the hybrid yields a lower cost per message and may therefore perform better.

10.3.5 Summary

We expect the hybrid architecture to have several very attractive properties. First, it reduces the amount of message passing dramatically, because there are only eight nodes (opposed to 32) involved in message passing. Second, a Niagara-tailored message passing protocol will drop the cost of message passing compared to the micro benchmarked costs presented earlier in this report. This assumes however that such a protocol indeed is feasible. Due to the small size of the L1 caches, we suspect that realizing such a protocol is not straight forward.

It appears however, that if message passing must be realized through L2 cache, SN will outperform the hybrid in the scenarios just presented. In this case, it will not matter whether messages are passed within the core or not. On the other hand, the hierarchical nature of the hybrid yields more messages than in SN. Because the cost associated with message passing is independent of core boundaries, the total number of messages becomes of vital importance[5].

Third, introducing parallel intra core execution of operations increases the throughput of complex transactions. Because there is only four threads to be synchronized (opposed to 32), there is less synchronizing. If a tailored synchronization mechanism that takes advantage of the shared L1 cache is applied, the synchronization cost will also drop.

On the reverse side, this architecture must deal with the most significant costs for both SN and SE. Adopted from SN are the cost of message passing and distributed transaction processing. It also encapsulates the SE cost of synchronization. In order to find out whether this hybrid architecture indeed yields a performance gain or not, one have to build a calculation model for the DBMS, in a similar fashion as the calculations performed for the SN and SE cases in this report.

Further research in this area is of great interest.

[5]Recall that message passing within different cores is performed in parallel when utilizing the hybrid.

Chapter 11
Further work

Although the approach pursued throughout this report is far more in touch with real life than the approach taken in our previous work, there are still many aspects which need attention. In addition we have identified several areas which are suggestive for additional research. In this chapter we therefore present several fields of interest for further work:

- A Niagara-optimized synchronization mechanism
- A Niagara-optimized message passing protocol
- Examine why Solaris Doors does not scale on Niagara
- Examine the effects of queuing
- Examine the effects of background processes
- Optimize the use of the node controller in SN
- Validate the hybrid architecture
- Achieve fault tolerance

The following sections describe each topic in detail.

11.1 A Niagara-optimized synchronization mechanism

In SE, the most frequent and by far most expensive cost is associated with synchronizing. We used System V Semaphores to implement synchronization. The System V Semaphore technique is indeed heavyweight, and it could as a beginning be interesting to examine the total SE cost based on the much more lightweight POSIX mutexes. POSIX mutexes is used in this report for intra process synchronization, but it has also the ability to synchronize inter process.

Albeit the use of POSIX mutexes could yield better results, we expect that a synchronization mechanism tailored for Niagara could be far more efficient. Given the Niagara architecture and the L2 cache response time of $\approx 100ns$, it should be possible to synchronize far less expensive than $\approx 11\ 300ns$.

11.2 A Niagara-optimized message passing protocol

In SN, the cost associated with message passing is even more dominant than the synchronization cost is in SE. We implemented two alternative methods of message passing in this report, namely message passing over TCP/IP sockets and over Solaris Doors. Because the L2 cache is shared between all cores, and thus, all logical CPUs (or nodes), it should be no reason to go off chip for message passing between nodes.

Our benchmark results has lead us to believe that this is not the case for either technique. As stated above, the L2 cache has an access time of $\approx 100ns$. Our benchmark results suggest a message passing cost two orders of magnitude greater. Clearly, there is much overhead. It seems likely that the following is the case:

TCP/IP The transmission short-circuits two deep in the stack, probably off chip.

Doors The shared data structure used for message passing resides in main memory.

A tailored message-passing protocol could easily benefit of the shared L2 cache for message passing. Further, as illustrated at page 22 in figure 3.2, L1 is internal to each core, but shared across the four native threads within each core. Thus, a tailored message-passing protocol could utilize the L1 cache for message passing between nodes within the same core. Of course, given the small size of the L1 cache in Niagara, this is perhaps only possible for the small 2PC messages, not the larger data-contained messages.

Such a message-passing mechanism needs a portion of the cache to be dedicated to message passing. This dedicated portion calls in turn for synchronization. This implies that this mechanism also will depend on the choice of synchronization mechanism explained in section 11.1.

Another suggestion, perhaps in a way more radical, is to simply dedicate a bus solely for message passing. This makes the message passing mechanism independent of synchronization costs.

Further research in the area of tailoring a message passing mechanism for Niagara is of great interest.

11.3 Examine why Solaris Doors does not scale on Niagara

In section 5.2 we presented the costs associated with passing messages when using Solaris Doors. It became apparent that Solaris Doors does not scale as well as TCP/IP when increasing the number of concurrent process pairs. We suggested that a possible reason to this might be heavy trashing of shared data structures. However, there might very well be other reasons, such as whether or not the early build of Solaris 11 handles Doors in an optimal manner when running on the T1000 server.

An interesting field of further research is to look into the reasons to why Solaris Doors do not scale on Niagara.

11.4 Examine the effects of queuing

Throughout the report we have chosen to ignore the effects of queuing. In SN we have assumed that all operations of a complex transaction touch different subsets of the database. In both SN and SE we have assumed that the effects of queuing are negligible. In real life, however, this is not necessarily true. In the SN case some of the operations of a complex transaction will occasionally touch the same database subset, and there will not be full parallelism. Equally in SE, the database locks and semaphores are not always available when need. In both approaches, if there is a rush of transactions, there might not always be an available node.

An interesting field of further work is to investigate the impact queuing has on the total transaction execution time.

11.5 Examine the effects of background processes

In chapter 2 we identified several background processes. These are:

- DB buffer flushing (SN and SE)
- Checkpoint logging (SN and SE)
- Cache invalidation (SE)

Because these processes are not inside the transaction scope, they are not taken into consideration when the total transaction execution times are calculated in chapters 7 and 8. However, these processes will occupy some of the available resources, and thus decrease the overall performance of the system.

An interesting field of further work is to investigate the impact background processes have on the total transaction execution time.

11.6 Optimize the use of the node controller in SN

As explained throughout the report, most of the work performed by the node controller when executing the STU and the STR transactions is pure overhead. Because the simple transactions only involve a single tuple, a distributed commit is not necessary. An interesting approach to this problem would be to push the node controller to the transaction performing the operating.

In the way our system is defined, when a simple transaction (STU or STR) is received from the external interface, the receiving node gets the role of controller and decides which node to be the slave. Instead of doing this, it would be desirable for the receiving node to pass the responsibility of being a controller to the node performing the operation. In this way, the overhead with message passing and logging needed to perform 2PC is avoided, and thus could make the simpler transactions more cost efficient.

11.7 Validate the hybrid architecture

Although we have proposed a hybrid architecture in chapter 10, the architecture still remains to be validated. Utilizing this hybrid architecture as a basis, instead of SN or SE, it is necessary to perform a research following the lines of the research presented in this report. Uncovering whether the proposed architecture indeed is capable of combining the advantages of SN and SE without adopting all their drawbacks, is indeed an interesting field for further work.

11.8 Achieve fault tolerance

An important topic of database systems that is not discussed in our research is fault tolerance. Achieving hardware fault tolerance when running a DBMS on an MCSD chip is of great interest. It indeed appears difficult to achieve independent failure modes when all nodes are located within the same die.

In the SN architecture, a failed node will prevent access to parts of the database, leaving the database only partially available. It would be of great interest to develop mechanisms for either to make a failing node hand its data responsibility over to an available node, or to make available nodes take responsibility over data belonging to nodes that have recently failed. It would be interesting to investigate approaches in directions of the solutions presented in [HTBH95].

BIBLIOGRAPHY

[ABC$^+$76] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim Gray, Patricia P. Griffiths, W. Frank King III, Raymond A. Lorie, Paul R. McJones, James W. Mehl, Gianfranco R. Putzolu, Irving L. Traiger, Bradford W. Wade, and Vera Watson. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[aMSST06a] Dieter an Mey, Samuel Sarholz, Alexander Spiegel, and Christian Terboven. Der UltraSPARCT1 Prozessor("Niagara") – Erste Erfahrungen. *"http://www.rz.rwth-aachen.de/computing/hpc/hw/niagara/Niagara_ZKI_2006-03-31_anMey.pdf"*, 2006.

[aMSST06b] Dieter an Mey, Samuel Sarholz, Alexander Spiegel, and Christian Terboven. The UltraSPARC T1 ("Niagara") based Sun Fire T2000 Server. *"http://www.rz.rwth-aachen.de/computing/hpc/hw/niagara.php"*, 2006.

[BJ05] Lars-Erik Bjørk and Truls Jørgensen. A relative performance comparison between the Shared Nothing and the Shared Everything DBMS architecture on a Multi Core, Single Die chip. *IDI, NTNU: "http://www.idi.ntnu.no/˜ trulsjor/project.pdf"*, 2005.

[BJA05] William Bryg and Sun Microsystems Jerome Alabado. The UltraSPARC T1 Processor - High Bandwidth For Throughput Computing. *"http://www.sun.com/processors/whitepapers/UST1_bw_v1.0.pdf"*, 12 2005.

[DG92] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Processing. *Communications of the ACM*, 36(6), 1992.

[Gra91] Jim Gray. *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, 1991.

[Hor84] Charles Hornig. A Standard for the Transmission of IP Datagrams over Ethernet Networks. *Request for Comments: 894, http://www.ietf.org/rfc/rfc894.txt*, 04 1984.

[HT93] Svein Olaf Hvasshovd and Øystein Torbjørnsen. A Software Architecture for a Continuosly Available Shared-Nothing Parallel DBMS Based on ATM Technology. *SINTEF Delab*, 1(1), 1993.

[HTBH95] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall PTR, 1988.

[MLO86] C. Mohan, B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4):378–396, 1986.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 04 1965.

[Moo75] Gordon E. Moore. Progress in digital integrated electronics. In *Proceedings of IEEE Digital Integrated Electronic Device Meeting*, pages 11–13, 1975.

[NV01] Nagendra Nagarajayya and S.R. Venkataramanan. Fast Sockets, An Interprocess Communication Library. *"http://developers.sun.com/solaris/articles/fast_sockets.pdf"*, 02 2001.

[NZT96] Michael G. Norman, Thomas Zurek, and Peter Thanisch. Much Ado About Shared-Nothing. *SIGMOD Record*, 25(3):16–21, 1996.

[She05] Denis Sheahan. Developing and Tuning Applications on UltraSPARC T1 Chip Multithreading Systems. *"http://www.sun.com/blueprints/1205/819-5144.pdf"*, 12 2005.

[Ste78] Steve R. Bourne. The Bourne Shell . *"http://steve-parker.org/sh/"*, 1978.

[Ste99] W. Richard Stevens. *UNIX Network Programming Volume 2 Interprocess Communication*. Prentice Hall, 2 edition, 1999.

[Sun03] Sun Microsystems. Introduction to Throughput Computing. *Sun Microsystems*, 2003.

[Sun05a] Sun Microsystems. *Programming Interfaces Guide*. Sun Microsystems, Inc., 2005.

[Sun05b] Sun Microsystems. Solaris 10 Reference Manual Collection. *Sun Microsystems, "http://docs.sun.com/app/docs/coll/40.10"*, 2005.

[Sun05c] Sun Microsystems. Solaris Dynamic Tracing Guide. *Sun Microsystems, http://docs.sun.com/app/docs/doc/817-6223*, 2005.

[Sun05d] Sun Microsystems. Sun Fire T1000 and T2000 Server Architecture White Paper. *"http://www.sun.com/servers/coolthreads/coolthreads_architecture_wp.pdf"*, 12 2005.

[Sun05e] Sun Microsystems. Sun Fire T2000 Server. *Sun Microsystems, "http://www.sun.com/servers/coolthreads/t1000/"*, 2005.

[Sun06a] Sun Microsystems. Solaris 10. *Sun Microsystems, "http://www.sun.com/software/solaris/specs.jsp "*, 2006.

[Sun06b] Sun Microsystems. Sun Fire CoolThreads Servers. *"http://www.sun.com/emrkt/trycoolthreads/coolthreads.jsp"*, 01 2006.

[Sun06c] Sun Microsystems. Sun Studio 11 Collection. *Sun Microsystems, "http://docs.sun.com/app/docs/coll/771.7"*, 2006.

[Sun06d] Sun Microsystems. UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005, Draft D2.0, 17 Mar 2006. *http://opensparc.sunsource.net/specs/UST1-UASuppl-current-draft-HP-EXT.pdf*, 2006.

[Swe01] L. Sweeney. *Information Explosion: Confidentiality, Disclosure, and Data Access - Theory and Practical Applications for Statistical Agencies*. Urban Institute, Washington, DC, 2001.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 2 edition, 1996.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001.

[Tuo02] Ilkka Tuomie. The Lives and Death of Moore's Law. *First Monday*, 7(11), 11 2002.

[Van05] Ashlee Vance. Sun employs scout to do dirty work on rock chips. *The Register, "http://www.theregister.co.uk/2005/11/04/sun_rock_scout/print.html"*, 2005.

APPENDIX A
ABBREVIATIONS

In this section, abbreviations commonly used in the report are listed and explained:

2PC Two-phase commit - A common protocol for committing distributed transactions

2PL Two-phase locking - A common protocol for locking resources in a database

ASCII American Standard Code for Information Interchange - A character encoding based on the English alphabet. Represents text in computers, communications equipment, and other devices that work with text

CMT Chip Level Multi Threading - The execution of instructions from multiple threads within one processor chip at the same time. Each core in Niagara utilizes this property. Sun Microsystems terms the Niagara chip "Radical CMT". In order to make the difference between ordinary CMT processors and Niagara clearer, we have chosen to term the Niagara chip MCSD (see below).

CPU Central Processing Unit - The main computational section of a computer that interprets and executes instructions

CTR Complex Transaction Read - A transaction reading four tuples

CTU Complex Transaction Update - A transaction updating four tuples

DBMS Database Management System - A software system that facilitates the creation, maintenance and use of a database

IP Internet Protocol - A data-oriented protocol used for communicating data across a packet-switched internetwork

IPC Inter process communication - A set of techniques for the exchange of data between two or more threads in one or more processes.

LRU Least Recently Used - A policy for managing queues

MCSD Multi Core Single Die - A hardware processor architecture that consists of several cores on the same die. Each core utilize the CMT property (see above).

MTU Maximum Transmission Unit - Size of the largest datagram that a given layer of a communications protocol can pass onwards

NOC Node Controller - The node running the transaction controller process

NOS Node Slave - The node running the transaction slave process

OS Operating System - A program which acts as an interface between a user of a computer and the computer hardware

RPC Remote Procedure Call - The procedure being called and the calling procedure are in different processes

SE Shared Everything - The processors share main memory and disk(s)

SMP Symmetric Multiprocessing - A multiprocessor computer architecture where two or more identical processors are connected to a single shared main memory. Most common multiprocessor systems today use an SMP architecture.

SN Shared Nothing - The processors do not share memory nor disk(s)

STR Simple Transaction Read - A transaction reading a single tuple

STU Simple Transaction Update - A transaction Updating a single tuple

TCP Transmission Control Protocol - A network protocol that guarantees reliable and in-order delivery of sender to receiver data

UDP User Datagram Protocol - A unreliable, connectionless network protocol that traditionally is considered faster than TCP due to less overhead involved (such as the sequence and flow control offered by TCP)

APPENDIX B
SHELL SCRIPTS

This appendix corresponds to section 4.2 For convenience, table B.1 lists all parameters varied for the benchmarks. The following sections in this appendix lists the shell scripts executed for each micro benchmark.

Benchmark 1: Message passing using TCP/IP

`nof_msg`	1 000 000									
`nof_proc`	2	4	6	8	16	32	64			
`msg_size`(B)	32	64	128	256	512	1024	2048	4096	8192	16384

Benchmark 2: Message passing using Solaris Doors

`nof_msg`	1 000 000									
`nof_proc`	2	4	6	8	16	32	64			
`msg_size`(B)	32	64	128	256	512	1024	2048	4096	8192	16384

Benchmark 3: Building or interpreting messages and log posts

`nof_msg`	1 000 000									
`post_size`(B)	32	64	128	256	512	1024	2048	4096	8192	16384

Benchmark 4: Writing to log

`nof_msg`	1 000 000							
`buffer_size`(MB)	10	20	40	80	160	320	640	1280

Benchmark 5: Synchronizing using POSIX Mutexes

`nof_mutexes`	1	10	1000	10000						
`nof_set/release`	100 000 000			10 000 000			1 000 000		100 000	

Benchmark 6: Synchronizing using System V Semaphores

`nof_set`	1 000 000						
`nof_proc`	2	4	6	8	16	32	64

Table B.1: The parameters varied in all the micro benchmarks performed.

run_pong

```
#!/usr/bin/sh

NOF_MESSAGES=$1
MESSAGE_SIZE=$2
NOF_PROC=$3

PORT=7000
i=0
while [ $i -lt $NOF_PROC ]
do
p=`expr $i \% 32`
    ../exec/pong $PORT $MESSAGE_SIZE > log/log_pong_${NOF_PROC}p_${PORT}_${NOF_MESSAGES}m_${
         MESSAGE_SIZE}b.log &
    pbind -b $p $!
    i=`expr $i + 1`
    PORT=`expr $PORT + 1`
done
```

Figure B.1: `run_pong`: Script for executing the `pong` benchmark.

run_ping

```
#!/usr/bin/sh

HOST=127.0.0.1
NOF_MESSAGES=$1
MESSAGE_SIZE=$2
NOF_PROC=$3

PORT=7000
i=`expr $NOF_PROC - 1`

while [ $i -ge 0 ]
do
    p=`expr $i \% 32`
    ../exec/ping $HOST $PORT $NOF_MESSAGES $MESSAGE_SIZE > log/log_ping_${NOF_PROC}p_${PORT}
         _${NOF_MESSAGES}m_${MESSAGE_SIZE}b.log &
    pbind -b $p $!
    i=`expr $i - 1`
    PORT=`expr $PORT + 1`
done
```

Figure B.2: `run_ping`: Script for executing the `ping` benchmark.

B.1 Benchmark 1: Message passing using TCP/IP sockets

The scripts in this section were used to automate the benchmarking of the costs associated with sending and receiving messages using TCP/IP sockets. These scripts are described in section 4.2.1.

B.1.1 `run_pong` and `run_ping`

The script shown in figure B.1 is used to automate the execution of the program `pong.c`. The script takes three input parameters, the number of messages (*NOF_MESSAGES*), the message size (*MESSAGE_SIZE*) and the number of processes (*NOF_PROC*). The port number of the first process is by default set to 7000. The port numbers of the following processes are automatically increased as the processes are started. Each process is bound to a processor by a call to `pbind`. This is done to ensure that processes are evenly distributed among processors. The output of each process is logged to a separate log file.

The script shown in figure B.2 is used to automate the program `ping.c`. Because all communication is done within the same computer, the host is set to `127.0.0.1` by default. This script is very similar to that of `pong.c`. However, processes are bound to process in the *rpps*

```
#!/bin/sh

echo starting init
../exec/pingpong_init
echo starting pong
../script/run_pong $1 $2 $3
echo starting ping
../script/run_ping $1 $2 $3
echo start pp race
../exec/pingpong_start $3
echo finish
../exec/pingpong_finish $3
echo done
```

Figure B.3: `rpps`: Script for executing `run_ping` and `run_pong`

run_pp_benchmark

```
#!/bin/sh


echo =======================
echo Benchmark 4: TCP IP
echo =======================
echo nof_msg : 1000000
echo msg_size: 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 B
echo nof_proc: 2,4,8,16,32,64
echo =======================
echo
echo

echo =======
echo   32B
echo =======
./rpps 1000000 32 1
sleep 5
./rpps 1000000 32 2
sleep 5
./rpps 1000000 32 4
sleep 5
./rpps 1000000 32 8
sleep 5
./rpps 1000000 32 16
sleep 5
./rpps 1000000 32 32
sleep 5
./rpps 1000000 32 64

sleep 5
echo =======
echo   64B
echo =======
```

Figure B.4: `run_pp_benchmark`: Script for executing `rpps`.

opposite order. This is done in order to ensure that corresponding processes in the process pair is executed on different logical CPUs.

B.1.2 rpps

The script `rpps` (short for run_pingpong_scheduler) shown in figure B.3 is a small script used to execute `run_pong` and `run_ping`. Because `pong.c` is the server process, this process is executed first.

run_doors

```
#!/usr/bin/sh

MESSAGE_SIZE=$1
NOF_PROC=$2
NOF_MESSAGES=$3

echo door_init:
../exec/door_init

i=`expr $NOF_PROC - 1`
echo door_server:
while [ $i -ge 0 ]
do
    p=`expr $i \% 32`
    ../exec/door_server $MESSAGE_SIZE /tmp/.bogusdoor${i} $NOF_MESSAGES > log/
        log_doors_server_${NOF_PROC}p_${i}_${NOF_MESSAGES}m_${MESSAGE_SIZE}B.log &
    pbind -b $p $!
    i=`expr $i - 1`
done

i=`expr $NOF_PROC - 1`
echo door_client:
while [ $i -ge 0 ]
do
    p=`expr $i \% 32`
    ../exec/door_client $MESSAGE_SIZE /tmp/.bogusdoor${i} $NOF_MESSAGES > log/
        log_doors_client_${NOF_PROC}p_${i}_${NOF_MESSAGES}m_${MESSAGE_SIZE}B.log &
    pbind -b $p $!
    i=`expr $i - 1`
done

echo door_start:
../exec/door_start $NOF_PROC
echo door_finish:
../exec/door_finish $NOF_PROC
echo done
```

Figure B.5: `run_doors`: Script for executing the `doors` benchmark.

B.1.3 `run_pp_benchmark`

The script `run_pp_benchmark` shown in figure B.4 is a script used to automate the scheduling, feeding `rpps` with input values as listed in table B.1. We show the execution of the 32B chunk. All other sizes are executed in similar fashion. Between execution of `rpps`, we sleep 5 seconds.

B.2 Benchmark 2: Message passing using Solaris Doors

The scripts in this section were used to automate the benchmarking of the costs associated with sending and receiving messages using Solaris Doors. These scripts are described in section 4.2.2.

B.2.1 `run_doors`

In the same fashion as `run_ping` (and `run_pong`), the script shown in figure B.5 is used to automate the execution of the program `doors_server.c` and `doors_client.c`. The script takes three input parameters, the message size (*MESSAGE_SIZE*), the number of processes (*NOF_proc*) and the number of messages (*NOF_MESSAGES*). Each process is bound to a processor by a call to `pbind`. This is done to ensure that processes are evenly distributed among processors. The output of each process is logged to a separate log file. The `pbind` calls are reversed for `doors_client`, in order to ensure that corresponding processes in the process pair is executed on different logical CPUs.

run_doors_benchmark

```
#!/bin/sh

echo =================================
echo Benchmark 1: Solaris Doors
echo =================================
echo msg_size: 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 B
echo nof_proc: 1, 2, 4, 8, 16,32,64
echo nof_msg : 1000000
echo =================================
echo
echo
echo =======
echo   32B
echo =======
rm /tmp/.*door*
./run_doors 32 1 1000000
rm /tmp/.*door*
sleep 5
./run_doors 32 2 1000000
rm /tmp/.*door*
sleep 5
./run_doors 32 4 1000000
rm /tmp/.*door*
sleep 5
./run_doors 32 8 1000000
rm /tmp/.*door*
sleep 5
./run_doors 32 16 1000000
rm /tmp/.*door*
sleep 5
./run_doors 32 32 1000000
rm /tmp/.*door*
sleep 5
```

Figure B.6: `run_doors_benchmark`: Script for scheduling the `run_doors` benchmark.

B.2.2 `run_doors_benchmark`

The script `run_doors_benchmark` shown in figure B.6 is a script used to automate the scheduling, feeding `run_doors` with input values as listed in table B.1. We show the execution of the 32B chunk. All other sizes are executed in similar fashion. Between execution of `run_doors`, we sleep 5 seconds.

B.3 BENCHMARK 3: BUILDING OR INTERPRETING MESSAGES AND LOG POSTS

run_build

```
#!/bin/sh

echo =================================
echo Benchmark 3: Build
echo =================================
echo nof_posts: 1000000
echo post_size: 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384

NOF_POSTS=$1

LOGFILE="log/log_build_${NOF_POSTS}.log"

echo 32B
echo 32B > $LOGFILE
../exec/build 32 $NOF_POSTS >> $LOGFILE

echo 64B
```

Figure B.7: Script for scheduling the benchmark of building messages.

run_myLog

```
#!/bin/sh

BUFFER_SIZE=$1
NOF_POSTS=1000000

LOGFILE="log/log_mylog_${NOF_POSTS}_${BUFFER_SIZE}.log"

echo 32b
echo 32b > $LOGFILE
../exec/myLog $BUFFER_SIZE 32 $NOF_POSTS >> $LOGFILE

echo 64b
```

Figure B.8: Script for scheduling the benchmark of writing to log.

run_myLog_benchmark

```
#!/bin/sh

echo ================================
echo Benchmark 4: myLog
echo ================================
echo nof_posts: 1000000
echo buffer_size: 10, 20, 40, 80, 160, 320, 640, 1280 MB

echo 10485760B - 10 MB
./run_myLog 10485760
sleep 5

echo 20971520B - 20 MB
```

Figure B.9: `run_myLog_benchmark`: Script for scheduling the `run_myLog` benchmark.

The script in this section is used to automate the benchmarking of the costs associated with building message and log posts. This scripts is described in section 4.2.3.

B.3.1 `run_build`

The script takes one input parameter, the number of posts to be built (*NOF_POSTS*). The program `build.c` , however, takes two parameters. All output from the program is logged to a log file. The script also provides progression feedback to the user interface. We show the execution of the 32B chunk. All other sizes (as listed in table B.1) are executed in similar fashion.

B.4 BENCHMARK 4: WRITING TO LOG

The scripts in this section are used to automate the benchmarking of the costs associated with writing to log. These scripts are described in section 4.2.4.

B.4.1 `run_myLog`

The script shown in figure B.8 is used to automate the program used to benchmark the costs associated with writing to log. The script takes one input parameter, namely the buffer size (*BUFFER_SIZE*). All output from the program is logged to a log file. The script also provides *run_sr_mutex*

```
#!/bin/sh

echo ==========================
echo Benchmark 5 Mutex
echo ==========================
echo nof_mutexes: 1, 10, 100, 1000
echo nof_sets_releases: 100000000, 10000000, 1000000, 100000

LOGFILE="log/log_sr_mutex.log"

echo 1 100000000
echo 1 100000000 > $LOGFILE
../exec/sr_mutex 1 100000000 >> $LOGFILE

echo 10 10000000
echo 10 10000000 >> $LOGFILE
../exec/sr_mutex 10 10000000 >> $LOGFILE

echo 100 1000000
echo 100 1000000 >> $LOGFILE
../exec/sr_mutex 100 1000000 >> $LOGFILE

echo 1000 100000
echo 1000 100000 >> $LOGFILE
../exec/sr_mutex 1000 100000 >> $LOGFILE
```

Figure B.10: Script for scheduling the benchmark of setting and releasing mutexes.

progression feedback to the user interface. We show the execution of the 32B chunk. All other sizes are executed in similar fashion.

B.4.2 `run_myLog_benchmark`

The script `run_myLog_benchmark` shown in figure B.9 is used to automate the scheduling, feeding `run_myLog` with input values as listed in table B.1. We show the execution of the 10MB chunk. All other sizes are executed in similar fashion. Between execution of `run_myLog`, we sleep 5 seconds.

B.5 Benchmark 5: Synchronizing using POSIX mutexes

The script in this section is used to automate the benchmarking of the costs associated with setting and releasing mutexes. This scripts is described in section 4.2.5.

B.5.1 `run_sr_mutex`

The script in figure B.10 is indeed simple and is used mostly for logging purposes and to avoid erroneous input that would easily occur if the benchmark was to be executed manually. The mutex benchmark differs from the other benchmarks by the amount of tests executed. Whereas the other benchmarks vary the parameters so that all possible combinations is executed, there is a 1-1 relationship between the parameters in the mutex script, as illustrated in the figure. The script logs the output of `sr_mutex.c` to the log file `log_sr_mutex.log`. The script provides progression feedback to the user interface.

run_lsem

```
#!/usr/bin/sh


NOF_PROC=$1
NOF_SETS=$2

i=`expr $NOF_PROC - 1`

echo lseminit:
../exec/lseminit $NOF_PROC
echo lsemset:
while [ $i -ge 0 ]
do
    p=`expr $i \% 32`
    ../exec/lsemset $NOF_SETS > log/log_semaphore_${NOF_PROC}p_${NOF_SETS}s_${i}.log &
    pbind -b $p $!
    i=`expr $i - 1`
done
echo lsemstart:
../exec/lsemstart $NOF_PROC
echo lsemrm:
../exec/lsemrm $NOF_PROC
echo done
```

Figure B.11: Script for scheduling the benchmark of setting and releasing semaphores.

run_sem_benchmark

```
#!/bin/sh

echo ==========================
echo Benchmark 6 Semaphore
echo ==========================
echo nof_set:  1000000
echo nof_proc: 1,2,4,8,16,32,64
echo
echo
echo =======
echo   1P
echo =======
./run_lsem 1 1000000
sleep 5
echo =======
echo   2P
echo =======
```

Figure B.12: `run_lsem_benchmark`: Script for scheduling the `run_lsem` benchmark.

B.6 BENCHMARK 6: SYNCHRONIZING USING SYSTEM V SEMAPHORES

The scripts in this section are used to automate the benchmarking of the costs associated with setting and releasing semaphores. These scripts are described in section 4.2.6.

B.6.1 `run_lsem`

The script shown in figure B.11 is used to automate the programs used to benchmark the costs associated with setting and releasing semaphores. The script takes two input parameters, namely the number of processes *NOF_PROC* and the number of times each process is to set and release the semaphore (*NOF_SETS*). The script starts with running the program `lseminit.c` which creates a semaphore set containing a single semaphore. The script then starts up the number of wanted processes of the program `lsemset.c` and distributes these evenly over the processors using a call to `pbind`. All processes log to separate log files. The processes are then started with the execution of `lsemstart`. When these processes are executed, the semaphore set is removed by executing the program `semrm.c`.

B.6.2 `run_sem_benchmark`

The script `run_sem_benchmark` shown in figure B.12 is used to automate the scheduling, feeding `run_lsem` with input values as listed in table B.1. We show the execution of the 1 process part chunk. All other process pairs are executed in similar fashion. Between execution of `run_lsem`, we sleep 5 seconds.

Appendix C
Code

This appendix lists the code used for all the benchmarks. The only change made since compile time is the modification of some of the comments.

C.1 Benchmark 1: Message passing using TCP/IP sockets

This section lists the source code for all programs used in benchmark 1.

C.1.1 `ping.c`

Figures C.1, C.2, C.3 and C.4 present the code for `ping.c`.

ping.c *(continues on page 172)*

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <float.h>
#include <sys/ipc.h>
#include <sys/sem.h>


/* The size of the local in-buffer */
#define MAXDATASIZE 32000

int main(int argc, char *argv[])
{
    /* Initializing all variables to zero */
    int sockfd = 0,
    numbytes = 0,
    port = 0,
    loop = 0,
    nof_messages = 0,
    message_size = 0,
    sent_bytes = 0;

    long long int total_time = 0;
    char buf[MAXDATASIZE];
    struct hostent *he;

    /* Information about the address to connect to */
    struct sockaddr_in their_addr;
    hrtime_t start, end;
```

Figure C.1: C code for `ping.c` part 1

ping.c (continued from page 171, continues on page 173)

```c

    key_t start_key;
    key_t finished_key;
    int startsem_id;
    int finishedsem_id;
    struct sembuf start_op = {0, -1, 0};
    /* Set to allocate resource */
    struct sembuf finished_op = {0, 1, 0};



    /* Making sure all arguments are given */
    if (argc != 5)
    {
        fprintf(stderr,"usage: client hostname port nof_messages message_size \n");
        exit(1);
    }


    /* Creating a key for the start semaphore set */
    if ((start_key = ftok("start.txt", 'E')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* Creating a key for the finish semaphore set */
    if ((finished_key = ftok("finished.txt", 'E')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* grabbing the semaphore set allready created for the start semaphore */
    if ((startsem_id = semget(start_key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }

    /* grabbing the semaphore set allready created for the finished semaphore */
    if ((finishedsem_id = semget(finished_key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }

    /* Get the host info */
    he=gethostbyname(argv[1]);
    if (he == NULL)
    {
        perror("gethostbyname");
        exit(1);
    }

    /* Get the port */
    port = atoi(argv[2]);
    if (port == NULL)
    {
        perror("port");
        exit(1);
    }

    /* Get number of messages */
    nof_messages = atoi(argv[3]);
```

Figure C.2: C code for `ping.c` part 2

ping.c (continued from page 172, continues on page 174)

```
    if (nof_messages == NULL)
    {
        perror("nof_messages");
        exit(1);
    }

    /* Get message size */
    message_size = atoi(argv[4]);
    if (message_size == NULL)
    {
        perror("message_size");
        exit(1);
    }

    /* Creating the socket */
    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }

    /* Initializing the address struct to connect to */
    their_addr.sin_family = AF_INET;
    their_addr.sin_port = htons(port);
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(their_addr.sin_zero), '\0', 8);

    /* Tuning the socket, making sure there is no buffering */
    int nobuf=0;
    if (setsockopt(sockfd,SOL_SOCKET,SO_SNDBUF ,&nobuf,sizeof(int)) == -1)
    {
        perror("setsockopt");
        exit(1);
    }


    /* Connect to socket */
    int cnt = connect(sockfd, (struct sockaddr *)&their_addr,sizeof(struct sockaddr));
    if (cnt == -1)
    {
        perror("connect");
        exit(1);
    }

    /* Send handshake containing the number of messages that is to be sent */
    char str[100];
    sprintf(str, "%d", htonl(nof_messages));
    send(sockfd, str, 100,0);

    /* The buffer containing the message to send */
    char my_message[message_size];

    /*Init the loop variable */
    loop = nof_messages;

    /* In order for all processes to begin at the same time, execution is halting until the
         start semaphore is available */
    if (semop(startsem_id, &start_op, 1) == -1) {
        perror("semop");
        exit(1);
    }
```

Figure C.3: C code for `ping.c` part 3

ping.c (continued from page 173)

```

    /* Ready, set, GO! */

    while (loop)
    {
        /* Build the message to be sent */
        int i=0;
        char *tp;
        for (tp = my_message; tp < my_message + message_size-1; tp++)
        {
            char character = 'L';
            *tp = character;
        }

        *tp = '\0';

        /* Get start time*/
        start = gethrtime();

        /* Sending the message just built */
        sent_bytes = send(sockfd, my_message, message_size,0); //send
        if (sent_bytes != message_size)
        {
            printf("Expected to send message of size %d b, was %d b\n", message_size,
                 sent_bytes);
        }

        /* Receiving response*/
        numbytes=recv(sockfd, buf, message_size, MSG_WAITALL);
        if (numbytes != message_size)
        {
            printf("Expected message of size %d b, was %d b\n", message_size, numbytes);
        }
        if (numbytes== -1)
        {
            perror("recv");
            exit(1);
        }
        /* Response recieved, get the end time */
        end = gethrtime();

        /* Add the time used to the total time */
        total_time += end - start;

        buf[numbytes] = '\0';

        /* Resetting the message buffer */
        memset(my_message, 0, message_size);

        loop--;
    }

    /* Adds 1 to the finished semaphore to indicate that the process is finished */
    if (semop(finishedsem_id, &finished_op, 1) == -1)
    {
        perror("semop");
        exit(1);
    }

    printf("[PING]Average \t %lld\n",(total_time/nof_messages));
}
```

Figure C.4: C code for `ping.c` part 4

C.1.2 pong.c

Figures C.5, C.6 and C.7 present the code for `pong.c`.

pong.c (continues on page 176)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <float.h>

/* Maximum number of pending connects */
#define BACKLOG 10

/* The size of the local in-buffer */
#define MAXDATASIZE 32000

int main(int argc, char *argv[])
{
    int listen_socket, receive_socket;
    int read_value, nof_messages, message_size, port, sin_size;
    int yes = 1;

    /* The total time spent working */
    long long int total_time;

    int loop = 0;

    /* The local in-buffer */
    char message_buffer[MAXDATASIZE];

    /* The address structure of the server socket */
    struct sockaddr_in listen_address;

    /* The address structure of the send/receive socket */
    struct sockaddr_in receiveAddress;

    /* Structures used to measure time differences */
    hrtime_t start, end;

    /* Making sure all arguments are given */
    if (argc != 3)
    {
        fprintf(stderr,"usage: client port message_size\n");
        exit(1);
    }

    port = atoi(argv[1]);
    message_size = atoi(argv[2]);

    if (message_size == NULL)
    {
        perror("message_size");
        exit(1);
```

Figure C.5: C code for `pong.c` part 1

pong.c (continued from page 175, continues on page 177)

```c
    }

    /* The buffer containing the message to send */
    char my_message[message_size];

    /* Creating the server socket */
    listen_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_socket == -1)
    {
        perror("socket");
        exit(1);
    }

    /* Tuning the socket, making sure there is no buffering */
    int nobuf=0;
    if (setsockopt(listen_socket,SOL_SOCKET,SO_SNDBUF,&nobuf,sizeof(int)) == -1)
    {
        perror("setsockopt");
        exit(1);
    }

    /* Making the address reusable */
    if (setsockopt(listen_socket,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1)
    {
        perror("setsockopt");
        exit(1);
    }

    /* Initializing the address of the server socket*/
    listen_address.sin_family = AF_INET;
    listen_address.sin_port = htons(port);
    listen_address.sin_addr.s_addr = INADDR_ANY;
    memset(&(listen_address.sin_zero), '\0', 8);

    /* Binding the server socket to the server address */
    int bind_res = bind(listen_socket, (struct sockaddr *)&listen_address, sizeof(struct
        sockaddr));
    if (bind_res == -1)
    {
        perror("bind");
        exit(1);
    }

    /* Setting the server socket to listen for connections */
    int listen_res = listen(listen_socket, BACKLOG);
    if (listen_res == -1)
    {
        perror("listen");
        exit(1);
    }

    sin_size = sizeof(struct sockaddr_in);

    /* Waiting for connect, using receive socket as a communication end-point */
    receive_socket = accept(listen_socket, (struct sockaddr *)&receiveAddress, &sin_size);
    if (receive_socket == -1)
    {
        perror("accept");
        exit(1);
    }

    memset(message_buffer, 0, sizeof message_buffer);

    /* Reading 100 bytes from the in-buffer*/
    read_value = recv(receive_socket, message_buffer, 100, MSG_WAITALL);
    if (read_value == -1)
```

Figure C.6: C code for `pong.c` part 2

pong.c (continued from page 176)

```
    {
        perror("Reading message");
        exit(0);
    }

    /* Converting the message in the in-buffer to an integer representing
    the number of messages to be sent and received */
    nof_messages = atoi(message_buffer);
    nof_messages = ntohl(nof_messages);

    /* Used to find the average time pr message */
    loop = nof_messages;

    total_time = 0.0;
    int sent_bytes = 0;

    while (nof_messages)
    {
        memset(message_buffer, 0, MAXDATASIZE);

        /* Receiveing message from the client */
        read_value = recv(receive_socket, message_buffer, message_size, MSG_WAITALL);

        /* Keeping track of the time when starting to work */
        start = gethrtime();


        if (read_value != message_size)
        {
            printf("Expected message of size %d b, was %d b\n", message_size, read_value);
        }

        if (read_value == -1)
        {
            perror("Reading message");
            break;
        }
        nof_messages--;

        /* Building the message to be sent */
        char *tp;
        for (tp = my_message; tp < my_message + message_size-1; tp++)
        {

            char character = 'T';
            *tp = character;
        }
        *tp = '\0';

        /* Keeping track of time when the work is finished */
        end = gethrtime();

        /* Sending the entire message via the receive socket */
        sent_bytes = send(receive_socket, my_message,message_size, 0);
        if (sent_bytes != message_size)
        {
            printf("Expected to send message of size %d b, was %d b\n", message_size,
                sent_bytes);
        }

        /* Adding the time used to the total time */
        total_time += end - start;
    }

    printf("[PONG] Average: \t %lld\n", (total_time / loop));
    close(receive_socket);
}
```

Figure C.7: C code for `pong.c` part 3

C.1.3 `pingpong_init.c`

Figures C.8 presents the code for `pingpong_init.c`.

pingpong_init.c

```c
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>


/* This file creates two semaphore sets: start and finished.
   Start is used to synchronize the start of all processes. Finish is used
   to synchronize the removal of the semaphore sets.*/
int main(int argc, char *argv[])
{


    key_t start_key;
    key_t finished_key;
    int startsem_id;
    int finishedsem_id;

    union semun
    {
            int val ;
            struct semid_ds *buf;
            unsigned short int *array;
    };


    /* Arguments for the semaphore sets */
    union semun start_arg;
    union semun finished_arg;

    /* Creating a key for each semaphore. The key is associated with a file */
    start_key = ftok("start.txt", 'E');
    finished_key = ftok("finished.txt", 'E');

    /* Creating the semaphore sets */
    startsem_id = semget(start_key, 1, 0666 | IPC_CREAT);
    finishedsem_id = semget(finished_key, 1, 0666 | IPC_CREAT);

    /* Initiating the values of the semaphore sets */
    start_arg.val = 0;
    finished_arg.val = 0;

    semctl(startsem_id, 0, SETVAL, start_arg);
    semctl(finishedsem_id, 0, SETVAL, finished_arg);

}
```

Figure C.8: C code for `pingpong_init.c`

C.1.4 pingpong_start.c

Figures C.9 presents the code for `pingpong_start.c`.

pingpong_start.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(int argc, char *argv[])
{
    key_t key;
    int semid;

    if(argc != 2)
    {
        printf("Usage: %s <nof_processes> \n", argv[0]);
        exit(1);
    }

    int nof_processes = atoi(argv[1]);


    /* Used to increment the semaphore before all processes may begin */
    struct sembuf op = {0, nof_processes, 0};

    /* Creating a key for the semaphore set */
    if ((key = ftok("start.txt", 'E')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* Grabbing the semaphore set allready created for the start semaphore */
    if ((semid = semget(key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }


    /* Ready, set, GO! */
    if (semop(semid, &op, 1) == -1)
    {
        perror("semop");
        exit(1);
    }

}
```

Figure C.9: C code for `pingpong_start.c`

C.1.5 `pingpong_finish.c`

Figures C.10 and C.11 present the code for `pingpong_finish.c`.

pingpong_finish.c, (continues on page 181)

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(int argc, char *argv[])
{
    key_t start_key;
    key_t finished_key;

    int startsem_id;
    int finishedsem_id;

    if(argc != 2)
    {
        printf("Usage: %s <nof_processes> \n", argv[0]);
        exit(1);
    }

    int nof_processes = atoi(argv[1]);
    int nof_grabs = nof_processes * -1;

    struct sembuf finished_op = {0, nof_grabs, 0};  /* Set to allocate resource */

    /* Creating a key that identifies the finish semaphore */
    if ((finished_key = ftok("finished.txt", 'E')) == -1)
    {
        perror("ftok");
        exit(1);
    }

    /* Grabbing the finish semaphore set */
    if ((finishedsem_id = semget(finished_key, 1, 0)) == -1)
    {
        perror("semget");
        exit(1);
    }
```

Figure C.10: C code for `pingpong_finish.c` part 1

pingpong_finish.c, (continued from page 180)

```
    /* Waiting until all processes are done executing */
    if (semop(finishedsem_id, &finished_op, 1) == -1) {
        perror("semop");
        exit(1);
    }

    printf("Removing finish semaphore \n");

    /* Removing the finish semaphore set */
    if (semctl(finishedsem_id, 0, IPC_RMID, (0, 0, 0)) == -1)
    {
        perror("semctl");
        exit(1);
    }


    /* Creating a key that identifies the start semaphore set*/
    if ((start_key = ftok("start.txt", 'E')) == -1)
    {
        perror("ftok");
        exit(1);
    }

    /* Grabbing the start semaphore set*/
    if ((startsem_id = semget(start_key, 1, 0)) == -1)
    {
        perror("semget");
        exit(1);
    }

    /* Removing the start semaphore set */
    if (semctl(startsem_id, 0, IPC_RMID, (0, 0, 0)) == -1)
    {
        perror("semctl");
        exit(1);
    }

    return 0;
}
```

Figure C.11: C code for `pingpong_finish.c` part 2

C.2 Benchmark 2: Message passing using Solaris Doors

This section lists the source code for all programs used in benchmark 2.

C.2.1 `doorclient.c`

Figures C.12, C.13 and C.14 present the code for `doorclient.c`.

doorclient.c (continues on page 183)

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/time.h>
#include <float.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/door.h>

char *message;
char *r_message;

/* The total time spent working */
long long int total_time;

/* Structures used to measure time differences */
hrtime_t start, end;
int nof_messages;

extern int errno;

int main(int argc, char *argv[]) {

    key_t start_key;
    key_t finished_key;
    int startsem_id;
    int finishedsem_id;
    struct sembuf start_op = {0, -1, 0};
    /* Set to allocate resource */
    struct sembuf finished_op = {0, 1, 0};

    /* Creating a key for the start semaphore set */
    if ((start_key = ftok("start.txt", 'E')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* Creating a key for the finish semaphore set */
    if ((finished_key = ftok("finished.txt", 'E')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* grabbing the semaphore set allready created for the start semaphore */
    if ((startsem_id = semget(start_key, 1, 0)) == -1) {
        perror("semget");
```

Figure C.12: C code for `doorclient.c` part 1

doorclient.c (continued from page 182, continues on page 184)

```
        exit(1);
    }

    /* grabbing the semaphore set allready created for the finished semaphore */
    if ((finishedsem_id = semget(finished_key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }

    int message_size = atoi(argv[1]);
    if (message_size  == NULL || message_size == 0)
    {
        perror("message_size");
        exit(1);
    }

    char* door_pathname = argv[2];
    if (door_pathname  == NULL || door_pathname == "")
    {
        perror("door_pathname");
        exit(1);
    }

    /* Get number of messages */
    nof_messages = atoi(argv[3]);
    if (nof_messages == NULL)
    {
        perror("nof_messages");
        exit(1);
    }

    message = (char*)malloc(message_size);
    r_message = (char*)malloc(message_size);
    int i;
    int did, d;
    door_arg_t darg;
    door_info_t info;

    if ((d=open(door_pathname, O_RDONLY)) < 0)
        perror("Open door"), exit(1);

    info.di_target=0;
    if (door_info(d, &info) < 0 ){
    perror("Door_info");
    printf("errno=%d\n", errno);
    exit(1);
    }

    int j = 0;
    int loop = nof_messages;

    /* In order for all processes to begin at the same time, execution is halting until the
        start semaphore is available */
    if (semop(startsem_id, &start_op, 1) == -1) {
        perror("semop");
        exit(1);
    }

    /* Ready, set, GO! */
    while(loop)
    {
```

Figure C.13: C code for `doorclient.c` part 2

doorclient.c (continued from page 183)

```
        /* Building the message to be sent */
        char *tp;
        for (tp = message; tp < message + message_size-1; tp++)
        {
            char character = 'C';
            *tp = character;
        }
        *tp = '\0';

        darg.data_ptr = (char *)message;
        darg.data_size = message_size;
        darg.desc_ptr = NULL;
        darg.desc_num = 0;
        darg.rbuf = r_message;
        darg.rsize = message_size;

        /* Keeping track of the time when starting to work */
        start = gethrtime();
        j++;
        door_call(d, &darg);
        /* Keeping track of time when the work is finished */
        end = gethrtime();
        /* Adding the time used to the total time */
        total_time += end - start;

        char *ptr = (char *)darg.data_ptr;
        loop--;
    }

    close(d);
    free(message);
    free(r_message);

    /* Adds 1 to the finished semaphore to indicate that the process is finished */
    if (semop(finishedsem_id, &finished_op, 1) == -1)
    {
        perror("semop");
        exit(1);
    }

    printf("[Client, msg: %d]Average: \t %lld \n", j, (total_time /nof_messages));

    return 0;
}
```

Figure C.14: C code for `doorclient.c` part 3

C.2.2 `doorserver.c`

Figures C.15, prog:doorserver2 and C.17 present the code for `doorserver.c`.

doorserver.c (continues on page 186)

```

#include <sys/door.h>
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <sys/types.h>

#define BOGUS_DOOR_COOKIE ((void*)(0xdeadbeef))

char *message;
int nof_messages;
int run=1;
int count=0;
static void server_proc();

/* The total time spent working */
long long int total_time;

/* Structures used to measure time differences */
hrtime_t start, end, tmp;

int main(int argc, char *argv[])
{
    int message_size = atoi(argv[1]);
    if (message_size  == NULL || message_size == 0)
    {
        perror("message_size");
        exit(1);
    }

    char* door_pathname = argv[2];
    if (door_pathname  == NULL || door_pathname == "")
    {
        perror("door_pathname");
        exit(1);
    }

    /* Get number of messages */
    nof_messages = atoi(argv[3]);
    if (nof_messages == NULL || nof_messages == 0)
    {
        perror("nof_messages");
        exit(1);
    }

    message = (char*)malloc(message_size);
    fdetach(door_pathname);
    init(door_pathname);
```

Figure C.15: C code for `doorserver.c` part 1

doorserver.c (continued from page 185), continues on page 187)

```
    while (run)
    {
        usleep(100);
    }
    sleep(1);
    fdetach(door_pathname);
    free(message);
    printf("[Server, msg: %d]Average: \t %lld \n", count, (total_time / nof_messages));
    return 0;
}

int init(char *door_pathname)
{
    int did, door, fd;
    struct stat buf;
    door_info_t info;

    if ((door = open(door_pathname, O_RDONLY)) >= 0)
    {
        if (door_info(door, &info) >= 0)
        {
            printf("door_info:info.di_target = %ld\n",
            info.di_target);
            if (info.di_target > 0)
            {
                (void) printf("door_server pid %ld already "
                        "running. Cannot start another "
                        "door_server pid %ld",
                        info.di_target, getpid());
                exit(1);
            }
        }
    }
    else
    {
        if (stat(door_pathname, &buf) < 0)
        {
            if ((fd = creat(door_pathname, 0644)) < 0)
            {
                    printf("doorfile creat failed\n");
                    exit(1);
            }
            (void) close(fd);
        }
    }

    if ((did =  door_create(server_proc, BOGUS_DOOR_COOKIE, DOOR_UNREF)) < 0)
    {
        perror("door_create");
        exit(-1);
    }

    if (fattach(did, door_pathname) < 0)
    {
        if ((errno != EBUSY) || (fdetach(door_pathname) <  0) || (fattach(did, door_pathname)
             < 0))
        {
            perror("door_attach");
        }
    }
    return 0;
```

Figure C.16: C code for `doorserver.c` part 2

doorserver.c (continued from page 186)

```c
}

static void server_proc(void *cookie, char *argp, size_t arg_size, door_desc_t *dp, size_t
    n_desc)
{
    /* Keeping track of the time when starting to work */
    start = gethrtime();

    if(count > 0)
    {
        total_time += end - tmp;
    }

    tmp = start;

    int ret;
    char *ptr = (char *)argp;

    if(ptr ==NULL || argp ==NULL)
    {
        printf("ptr or argp is NULL");

    }

    if (argp == DOOR_UNREF_DATA) /* fdetach() called by server in main() results in the final
        server_proc call */
    {
        door_return(NULL, 0, NULL, 0);
    }

    if (ptr == NULL) /* empty door call */
    {
        (void) door_return(NULL, 0, 0, 0); /* return the favor */
    }

    /* Building the message to be sent */
    char *tp;
    for (tp = message; tp < message + arg_size-1; tp++)
    {
        char character = 'S';
        *tp = character;
    }
    *tp = '\0';

    count++;

    if(count == nof_messages)
    {
        run = 0;
    }

    /* Keeping track of time when the work is finished */
    end = gethrtime();

    ret = door_return((void *)message, arg_size, NULL, 0);

    if (ret < 0)
    {
        perror("door_return"), exit(1);
    }
}
```

Figure C.17: C code for `doorserver.c` part 3

C.2.3 `door_init.c`

Figures C.18 presents the code for `door_init.c`.

door_init.c

```c
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>


/* This file creates two semaphore sets: start and finished.
   Start is used to synchronize the start of all processes. Finish is used
   to synchronize the removal of the semaphore sets.*/
int main(int argc, char *argv[])
{


    key_t start_key;
    key_t finished_key;
    int startsem_id;
    int finishedsem_id;

    union semun
    {
            int val ;
            struct semid_ds *buf;
            unsigned short int *array;
    };


    /* Arguments for the semaphore sets */
    union semun start_arg;
    union semun finished_arg;

    /* Creating a key for each semaphore. The key is associated with a file */
    start_key = ftok("start.txt", 'E');
    finished_key = ftok("finished.txt", 'E');

    /* Creating the semaphore sets */
    startsem_id = semget(start_key, 1, 0666 | IPC_CREAT);
    finishedsem_id = semget(finished_key, 1, 0666 | IPC_CREAT);

    /* Initiating the values of the semaphore sets */
    start_arg.val = 0;
    finished_arg.val = 0;

    semctl(startsem_id, 0, SETVAL, start_arg);
    semctl(finishedsem_id, 0, SETVAL, finished_arg);

}
```

Figure C.18: C code for `door_init.c`

C.2.4 door_start.c

Figures C.19 presents the code for `door_start.c`.

door_start.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(int argc, char *argv[])
{
    key_t key;
    int semid;

    if(argc != 2)
    {
        printf("Usage: %s <nof_processes> \n", argv[0]);
        exit(1);
    }

    int nof_processes = atoi(argv[1]);


    /* Used to increment the semaphore before all processes may begin */
    struct sembuf op = {0, nof_processes, 0};

    /* Creating a key for the semaphore set */
    if ((key = ftok("start.txt", 'E')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* Grabbing the semaphore set allready created for the start semaphore */
    if ((semid = semget(key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }


    /* Ready, set, GO! */
    if (semop(semid, &op, 1) == -1)
    {
        perror("semop");
        exit(1);
    }

}
```

Figure C.19: C code for `door_start.c`

C.2.5 door_finish.c

Figures C.20 and C.21 present the code for `door_finish.c`.

door_finish.c, (continues on page 191)

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(int argc, char *argv[])
{
    key_t start_key;
    key_t finished_key;

    int startsem_id;
    int finishedsem_id;

    if(argc != 2)
    {
        printf("Usage: %s <nof_processes> \n", argv[0]);
        exit(1);
    }

    int nof_processes = atoi(argv[1]);
    int nof_grabs = nof_processes * -1;

    struct sembuf finished_op = {0, nof_grabs, 0};  /* Set to allocate resource */

    /* Creating a key that identifies the finish semaphore */
    if ((finished_key = ftok("finished.txt", 'E')) == -1)
    {
        perror("ftok");
        exit(1);
    }

    /* Grabbing the finish semaphore set */
    if ((finishedsem_id = semget(finished_key, 1, 0)) == -1)
    {
        perror("semget");
        exit(1);
    }
```

Figure C.20: C code for `door_finish.c` part 1

door_finish.c, (continued from page 190)

```
    /* Waiting until all processes are done executing */
    if (semop(finishedsem_id, &finished_op, 1) == −1) {
        perror("semop");
        exit(1);
    }

    printf("Removing finish semaphore \n");

    /* Removing the finish semaphore set */
    if (semctl(finishedsem_id, 0, IPC_RMID, (0, 0, 0)) == −1)
    {
        perror("semctl");
        exit(1);
    }

    /* Creating a key that identifies the start semaphore set*/
    if ((start_key = ftok("start.txt", 'E')) == −1)
    {
        perror("ftok");
        exit(1);
    }

    /* Grabbing the start semaphore set*/
    if ((startsem_id = semget(start_key, 1, 0)) == −1)
    {
        perror("semget");
        exit(1);
    }

    /* Removing the start semaphore set */
    if (semctl(startsem_id, 0, IPC_RMID, (0, 0, 0)) == −1)
    {
        perror("semctl");
        exit(1);
    }

    return 0;
}
```

Figure C.21: C code for `door_finish.c`part 2

C.3 Benchmark 3: Building or interpreting messages and log posts

This section lists the source code for all programs used in benchmark 3.

C.3.1 build.c

Figures C.22 and C.23 present the code for `build.c`.

build.c, (continues on page 193)

```c
#include <stdio.h>
#include <sys/time.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

int main(int argc, char *argv[])
{
    int postsize;
    int nof_posts;

    /* Making sure all arguments are given */
    if (argc != 3)
    {
        fprintf(stderr, "Usage: <postsize> <nof_posts> \n");
        exit(1);
    }

    postsize = atoi(argv[1]);
    nof_posts = atoi(argv[2]);


    /* The buffer used to hold the post/message */
    char buffer[postsize];

    int i = 0;
    int j;

    /* The total time spent working */
    long long int total_time = 0;
```

Figure C.22: C code for `build.c` part 1

build.c, (continued from page 192)

```c
    /* Structures used to measure time differences */
    hrtime_t start, end;

    memset(buffer, 0, postsize);

    /* Starting to build posts/messages */
    start = gethrtime();
    char *ap;
    while (i<nof_posts)
    {
        j = 0;
        ap = buffer;
        while (j<postsize)
        {
            *ap = 'L';
            ap++;
            j++;
        }

    i++;
    }

    *(buffer+postsize-1)='\0';
    end = gethrtime();

    /* Adding the time used to the total time */
    total_time = end - start;

    printf("Average: \t%lld ns\n", (total_time/nof_posts));

    return 0;
}
```

Figure C.23: C code for `build.c` part 2

C.4 Benchmark 4: Writing to log

This section lists the source code for all programs used in benchmark 4.

C.4.1 `myLog.c`

Figures C.24 and C.25 present the code for `myLog.c`.

myLog.c, *(continues on page 195)*

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/time.h>

char *buffer;

int main(int argc, char *argv[])
{
    /* Initializing all variables to zero */
        unsigned int buffer_filled = 0,
        buffersize = 0,
        postsize = 0,
        nof_posts = 0;

    /* Making sure all arguments are given */
    if (argc != 4)
    {
        fprintf(stderr, "Usage: <buffersize> <postsize> <nof_posts> \n");
        exit(1);
    }

    /* Get the buffer size */
    buffersize = atoi(argv[1]);
    if (buffersize == NULL)
    {
        perror("buffer_size");
        exit(1);
    }

    /* Get the post size */
    postsize = atoi(argv[2]);
    if (postsize == NULL)
    {
        perror("post_size");
        exit(1);
    }

    /* Get number of posts */
    nof_posts = atoi(argv[3]);
    if (nof_posts == NULL)
    {
        perror("nof_posts");
        exit(1);
    }

    /* Init the buffer */
    buffer = (char*)malloc(buffersize);
    char *ap = buffer;

    /* Zero out the buffer before using it*/
    memset(buffer, 0, buffersize);
```

Figure C.24: C code for `myLog.c` part 1

myLog.c, (continued from page 194)

```c

    hrtime_t start, end;

    int i;
    i = nof_posts;
    int j;

    /* Get start time*/
    start = gethrtime();
    while (i)
    {
        j = postsize;

        /* If there is enough storage for the post at the end of the buffer */
        if ((buffer + buffersize - ap) >= postsize)
        {
            while(j)
            {
                *ap = 'L';
                ap++;
                j--;
            }
        }
        /* Else, there is not enough storage at the end of the buffer, starting at the
           beginning*/
        else
        {
            buffer_filled++;
            ap = buffer;
            i++;
        }
        i--;
        if( (ap + (256*1024)) < (buffer + buffersize))
        {
            ap += (256*1024);
        }

    }

    /* Finished, get end time*/
    end = gethrtime();

    long long int average_time = (end - start) / nof_posts;
    printf("Average: \t%lld ns \n", average_time);

    free(buffer);
}
```

Figure C.25: C code for `myLog.c` part 2

C.5 Benchmark 5: Synchronizing using POSIX Mutexes

This section lists the source code for all programs used in benchmark 5.

C.5.1 `sr_mutex.c`

Figures C.26 and C.27 present the code for `sr_mutex.c`.

sr_mutex.c, (continues on page 197)

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/time.h>

int main(int argc, char *argv[])
{
    /* Making sure all arguments are given */
    if (argc != 3)
    {
        fprintf(stderr,"usage: sr_mutex <nof_mutexes> <nof_set_rel> \n");
        exit(1);
    }

    /* The number of mutexes */
    int nof_mutexes = atoi(argv[1]);
    if (nof_mutexes == NULL)
    {
        perror("nof_mutexes");
        exit(1);
    }

    /* The number of times each mutex is set */
    int nof_set_rel = atoi(argv[2]);
    if (nof_set_rel == NULL)
    {
        perror("nof_set_rel");
        exit(1);
    }


    /* Creating an array of mutexes */
    pthread_mutex_t mutexes[nof_mutexes];

    /* Variables used to hold the total time used to lock and unlock */
    long long int locking, unlocking;
    locking = unlocking = 0;
    int i, j;

    /* Initializing the mutexes */
    for (i = 0; i < nof_mutexes; i++)
    {
        pthread_mutex_t mut;
        mutexes[i] = mut;
    }

    /* Structures used to measure time differences */
    hrtime_t    before_setting,
                after_setting,
```

Figure C.26: C code for `sr_mutex.c` part 1

sr_mutex.c, (continued from page 196)

```
                before_releasing,
                after_releasing;

    /* Looping through the array of mutexes j times, setting and releasing every mutex each
        iteration */
    for (j = 0; j < nof_set_rel; j++)
    {
        before_setting = gethrtime();
        for(i = 0; i < nof_mutexes; i++)
        {
            /* Setting the mutexes */
            pthread_mutex_lock(&mutexes[i]);
        }

        after_setting = gethrtime();

        /* Adding the time used to the total time spent locking */
        locking += after_setting - before_setting;


        before_releasing = gethrtime();
        for(i = 0; i < nof_mutexes; i++)
        {
            /* Releasing the mutexes */
            pthread_mutex_unlock(&mutexes[i]);
        }

        after_releasing = gethrtime();

        /* Adding the time used to the total time spent unlocking */
        unlocking += after_releasing - before_releasing;
    }

    printf("Setting average: \t%lld ns\n", locking/(nof_set_rel*nof_mutexes));
    printf("Releasing average: \t%lld ns\n", unlocking/(nof_set_rel*nof_mutexes));
}
```

Figure C.27: C code for `sr_mutex.c` part 2

C.6 Benchmark 6: Synchronizing using System V Semaphores

This section lists the source code for all programs used in benchmark 6.

C.6.1 `lsemset.c`

Figures C.28 and C.29 present the code for `lsemset.c`.

lsemset.c, (continues on page 199)

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(int argc, char *argv[])
{
    key_t start_key;
    key_t main_key;
    key_t finished_key;
    int startsem_id;
    int mainsem_id;
    int finishedsem_id;
    long long int total_time;
    hrtime_t start, end;
    struct sembuf start_op = {0, -1, 0};
    struct sembuf main_op = {0, -1, 0};  /* Set to allocate resource */
    struct sembuf finished_op = {0, 1, 0};  /* Set to allocate resource */
    long long int set_time = 0;
    long long int release_time = 0;

    if(argc != 2)
    {
        printf("Usage: %s <nof_sets> \n", argv[0]);
        exit(1);
    }

    int nof_sets = atoi(argv[1]);
    int loop = nof_sets;

    /* Creating a key for the start semaphore set */
    if ((start_key = ftok("start.txt", 'E')) == -1) {
        perror("ftok");
        exit(1);
    }


    /* Creating a key for the main semaphore set */
    if ((main_key = ftok("grabme.txt", 'E')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* Creating a key for the finish semaphore set */
    if ((finished_key = ftok("finished.txt", 'E')) == -1) {
        perror("ftok");
        exit(1);
    }


    /* grabbing the semaphore set allready created for the start semaphore */
    if ((startsem_id = semget(start_key, 1, 0)) == -1) {
        perror("semget");
```

Figure C.28: C code for `lsemset.c` part 1

lsemset.c, (continued from page 198)

```
            exit(1);
        }


        /* grabbing the semaphore set allready created for the main semaphore */
        if ((mainsem_id = semget(main_key, 1, 0)) == -1) {
            perror("semget");
            exit(1);
        }

        /* grabbing the semaphore set allready created for the finished semaphore */
        if ((finishedsem_id = semget(finished_key, 1, 0)) == -1) {
            perror("semget");
            exit(1);
        }


        /* In order for all processes to begin at the same time, execution is halting until the
           start semaphore is available */
        if (semop(startsem_id, &start_op, 1) == -1) {
            perror("semop");
            exit(1);
        }

        /* Ready, set, GO! */
        while(loop)
        {


            /* Grab semaphore */
            start = gethrtime();
            main_op.sem_op = -1;
            if (semop(mainsem_id, &main_op, 1) == -1)
            {
                perror("semop");
                exit(1);
            }
            end = gethrtime();
            set_time += (end - start);

            /* Release semaphore */
            start = gethrtime();
            main_op.sem_op = 1;
            if (semop(mainsem_id, &main_op, 1) == -1)
            {
                perror("semop");
                exit(1);
            }

            end = gethrtime();
            release_time += end - start;
            loop--;
        }

        /* Adds 1 to the finished semaphore to indicate that the process is finished */
        if (semop(finishedsem_id, &finished_op, 1) == -1)
        {
            perror("semop");
            exit(1);
        }

        printf("[SEM] Average set time: %lld \n", (set_time / nof_sets));
        printf("[SEM] Average release time: %lld \n", (release_time / nof_sets));
        return 0;
}
```

Figure C.29: C code for `lsemset.c` part 2

C.6.2 `lseminit.c`

Figures C.30 and C.31 present the code for `lseminit.c`.

lseminit.c, (continues on page 201)

```c
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>


/* This file creates three semaphore sets: start, main and finished.
   Start is used to synchronize the start of all processes. Main is the semaphore
   used to measure the time used to grab and release a semaphore. Finish is used
   to synchronize the removal of the semaphore sets.*/
int main(int argc, char *argv[])
{
    key_t start_key;
    key_t main_key;
    key_t finished_key;
    int startsem_id;
    int mainsem_id;
    int finishedsem_id;

    if(argc != 2)
    {
        printf("Usage: %s <nof_processes> \n", argv[0]);
        exit(1);
    }

    int nof_processes = atoi(argv[1]);

    union semun
    {
            int val ;
```

Figure C.30: C code for `lseminit.c` part 1

lseminit.c, (continued from page 200)

```c
            struct semid_ds *buf;
            unsigned short int *array;
    };

    /* Arguments for the semaphore sets */
    union semun start_arg;
    union semun main_arg;
    union semun finished_arg;

    /* Creating a key for each semaphore. The key is associated with a file */
    start_key = ftok("start.txt", 'E');
    main_key = ftok("grabme.txt", 'E');
    finished_key = ftok("finished.txt", 'E');

    /* Creating the semaphore sets */
    startsem_id = semget(start_key, 1, 0666 | IPC_CREAT);
    mainsem_id = semget(main_key, 1, 0666 | IPC_CREAT);
    finishedsem_id = semget(finished_key, 1, 0666 | IPC_CREAT);

    /* Initiating the values of the semaphore sets */
    start_arg.val = 0;
    main_arg.val = nof_processes;
    finished_arg.val = 0;

    semctl(startsem_id, 0, SETVAL, start_arg);
    semctl(mainsem_id, 0, SETVAL, main_arg);
    semctl(finishedsem_id, 0, SETVAL, finished_arg);

}
```

Figure C.31: C code for `lseminit.c` part 2

C.6.3 `lsemstart.c`

Figure C.32 presents the code for `lsemstart.c`.

lsemstart.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(int argc, char *argv[])
{
    key_t key;
    int semid;

    if(argc != 2)
    {
        printf("Usage: %s <nof_processes> \n", argv[0]);
        exit(1);
    }

    int nof_processes = atoi(argv[1]);


    /* Used to increment the semaphore before all processes may begin */
    struct sembuf op = {0, nof_processes, 0};

    /* Creating a key for the semaphore set */
    if ((key = ftok("start.txt", 'E')) == -1) {
        perror("ftok");
        exit(1);
    }

    /* Grabbing the semaphore set allready created for the start semaphore */
    if ((semid = semget(key, 1, 0)) == -1) {
        perror("semget");
        exit(1);
    }

    /* Ready, set, GO! */
    if (semop(semid, &op, 1) == -1)
    {
        perror("semop");
        exit(1);
    }

}
```

Figure C.32: C code for `lsemstart.c`

C.6.4 `lsemrm.c`

Figures C.33 and C.34 present the code for `lsemrm.c`.

lsemrm.c, (continues on page 198)

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(int argc, char *argv[])
{
    key_t start_key;
    key_t main_key;
    key_t finished_key;

    int startsem_id;
    int mainsem_id;
    int finishedsem_id;

    if(argc != 2)
    {
        printf("Usage: %s <nof_processes> \n", argv[0]);
        exit(1);
    }

    int nof_processes = atoi(argv[1]);
    int nof_grabs = nof_processes * -1;

    struct sembuf finished_op = {0, nof_grabs, 0};  /* Set to allocate resource */

    /* Creating a key that identifies the finish semaphore */
    if ((finished_key = ftok("finished.txt", 'E')) == -1)
    {
        perror("ftok");
        exit(1);
    }

    /* Grabbing the finish semaphore set */
    if ((finishedsem_id = semget(finished_key, 1, 0)) == -1)
    {
        perror("semget");
        exit(1);
    }

    /* Waiting until all processes are done executing */
    if (semop(finishedsem_id, &finished_op, 1) == -1) {
        perror("semop");
        exit(1);
    }

    /* Removing the finish semaphore set */
    if (semctl(finishedsem_id, 0, IPC_RMID, (0, 0, 0)) == -1)
    {
        perror("semctl");
        exit(1);
    }
```

Figure C.33: C code for `lsemrm.c` part 1

lsemrm.c, (continued from page 198)

```
    /* Creating a key that identifies the start semaphore set*/
    if ((start_key = ftok("start.txt", 'E')) == -1)
    {
        perror("ftok");
        exit(1);
    }

    /* Grabbing the start semaphore set*/
    if ((startsem_id = semget(start_key, 1, 0)) == -1)
    {
        perror("semget");
        exit(1);
    }

    /* Removing the start semaphore set */
    if (semctl(startsem_id, 0, IPC_RMID, (0, 0, 0)) == -1)
    {
        perror("semctl");
        exit(1);
    }

    /* Creating a key that identifies the main semaphore set */
    if ((main_key = ftok("grabme.txt", 'E')) == -1)
    {
        perror("ftok");
        exit(1);
    }

    /* Grabbing the main semaphore set */
    if ((mainsem_id = semget(main_key, 1, 0)) == -1)
    {
        perror("semget");
        exit(1);
    }

    /* Removing the main semaphore set */
    if (semctl(mainsem_id, 0, IPC_RMID, (0, 0, 0)) == -1)
    {
        perror("semctl");
        exit(1);
    }

    return 0;
}
```

Figure C.34: C code for `lsemrm.c` part 2

Appendix D
Result data

This appendix lists all results data from the benchmarks. The following sections list all results returned from each benchmark in turn.

D.1 Benchmark 1: Message passing using TCP/IP sockets

Table D.1 lists the results obatined from running bencmark 1, sorted by number of process pairs. All numbers are in ns. Numbers in columms *Ping* and *Pong* are averaged over 1 000 000 runs.

Nof_proc	Msg_size	Ping (ns)	Pong (ns)	(PI-PO)/4 (ns)
1p	*32B*	81941	675	20317
1p	*64B*	81879	1221	20165
1p	*128B*	84787	2314	20618
1p	*256B*	89521	4499	21256
1p	*512B*	96604	8875	21932
1p	*1024B*	112358	17628	23683
1p	*2048B*	138592	35125	25867
1p	*4096B*	192967	70112	30714
1p	*8192B*	302850	140117	40683
1p	*16384B*	524488	280071	61104
2p	*32B*	115407	686	28680
2p	*64B*	103334	1238	25524
2p	*128B*	104059	2373	25421
2p	*256B*	106744	4576	25542
2p	*512B*	109054	9326	24932
2p	*1024B*	123282	19210	26018
2p	*2048B*	170825	61690	27284
2p	*4096B*	253393	94030	39841
2p	*8192B*	352241	181065	42794
2p	*16384B*	590777	334594	64046
4p	*32B*	116886	760	29031
4p	*64B*	117195	1370	28956
4p	*128B*	119266	2675	29148
4p	*256B*	118460	4970	28372
4p	*512B*	125588	9720	28967
4p	*1024B*	140975	19070	30476
4p	*2048B*	171462	40374	32772

Nof_proc	**Msg_size**	**Ping (ns)**	**Pong (ns)**	**(PI-PO)/4 (ns)**
4p	*4096B*	235703	84216	37872
4p	*8192B*	349273	153426	48962
4p	*16384B*	594511	307379	71783
8p	*32B*	116319	768	28888
8p	*64B*	126364	1465	31225
8p	*128B*	115900	2909	28248
8p	*256B*	117643	5601	28011
8p	*512B*	126957	10557	29100
8p	*1024B*	142380	20533	30462
8p	*2048B*	176714	45202	32878
8p	*4096B*	245238	90388	38713
8p	*8192B*	384642	184684	49990
8p	*16384B*	602490	301159	75333
16p	*32B*	122020	772	30312
16p	*64B*	117773	1459	29079
16p	*128B*	118825	3079	28937
16p	*256B*	119617	5920	28424
16p	*512B*	131909	11417	30123
16p	*1024B*	151315	21935	32345
16p	*2048B*	181256	46166	33773
16p	*4096B*	248598	85264	40834
16p	*8192B*	375989	168681	51827
16p	*16384B*	659970	340236	79933
32p	*32B*	129196	902	32073
32p	*64B*	131626	1796	32457
32p	*128B*	311553	2637	77229
32p	*256B*	139149	7987	32791
32p	*512B*	151714	14948	34192
32p	*1024B*	174606	28840	36442
32p	*2048B*	241073	57880	45798
32p	*4096B*	329535	140113	47355
32p	*8192B*	550760	286798	65990
32p	*16384B*	1000219	555336	111221
64p	*32B*	851671	732	212735
64p	*64B*	1020099	1430	254667
64p	*128B*	271153	8748	65601
64p	*256B*	1040162	5294	258717
64p	*512B*	301468	33565	66976
64p	*1024B*	1021690	19665	250506
64p	*2048B*	457504	156787	75179
64p	*4096B*	678148	289596	97138
64p	*8192B*	1087808	554627	133295
64p	*16384B*	1934448	1103270	207795

Table D.1: Ping-pong data. Data from the TCP/IP benchmark.

D.2 Benchmark 2: Message passing using Solaris Doors

Table D.2 lists the results obatined from running benchmark 2, sorted by number of process pairs. All numbers are in ns. Numbers in columms *Client* and *Server* are averaged over 1 000 000 runs.

Nof_proc	Msg_size	Client	Server	C-S/4
1p	*32B*	34391	1900	8122
1p	*64B*	35305	2450	8214
1p	*128B*	37065	3531	8384
1p	*256B*	40636	5760	8719
1p	*512B*	47529	10156	9343
1p	*1024B*	61200	18962	10560
1p	*2048B*	88926	36409	13129
1p	*4096B*	143718	71625	18023
1p	*8192B*	252213	141747	27617
1p	*16384B*	469596	282370	46807
2p	*32B*	46889	2893	10999
2p	*64B*	46890	3418	10868
2p	*128B*	47018	4230	10697
2p	*256B*	50388	6556	10958
2p	*512B*	56929	10860	11517
2p	*1024B*	71131	19626	12876
2p	*2048B*	96512	37050	14866
2p	*4096B*	150948	72201	19687
2p	*8192B*	259277	142148	29282
2p	*16384B*	477477	282722	48689
4p	*32B*	82243	3712	19633
4p	*64B*	80382	4269	19028
4p	*128B*	80958	5354	18901
4p	*256B*	83576	7567	19002
4p	*512B*	85571	12032	18385
4p	*1024B*	92621	20983	17910
4p	*2048B*	112890	38862	18507
4p	*4096B*	168880	74828	23513
4p	*8192B*	283638	147264	34094
4p	*16384B*	511414	291761	54913
8p	*32B*	137858	4013	33461
8p	*64B*	135812	4558	32813
8p	*128B*	134668	5655	32253
8p	*256B*	135193	7862	31833
8p	*512B*	132138	12270	29967
8p	*1024B*	132167	21230	27734
8p	*2048B*	133912	39398	23629
8p	*4096B*	178921	76174	25687
8p	*8192B*	288258	150329	34482
8p	*16384B*	518820	298989	54958

Nof_proc	Msg_size	Client	Server	C-S/4
16p	*32B*	250016	4198	61454
16p	*64B*	249104	4747	61089
16p	*128B*	249326	5847	60870
16p	*256B*	248193	8061	60033
16p	*512B*	244848	12490	58090
16p	*1024B*	239211	21464	54437
16p	*2048B*	219213	39975	44810
16p	*4096B*	218098	79149	34737
16p	*8192B*	310043	157947	38024
16p	*16384B*	542695	314020	57169
32p	*32B*	470534	4311	116556
32p	*64B*	470567	4878	116422
32p	*128B*	472149	5981	116542
32p	*256B*	473204	8202	116250
32p	*512B*	469000	12644	114089
32p	*1024B*	463705	21667	110509
32p	*2048B*	430973	40488	97621
32p	*4096B*	396755	83279	78369
32p	*8192B*	416749	216515	50059
32p	*16384B*	760958	485625	68833
64p	*32B*	947301	9742	234390
64p	*64B*	946247	10867	233845
64p	*128B*	949495	13725	233942
64p	*256B*	949489	19009	232620
64p	*512B*	946572	29588	229246
64p	*1024B*	941146	51895	222313
64p	*2048B*	893211	95735	199369
64p	*4096B*	828116	187869	160062
64p	*8192B*	863334	436361	106743
64p	*16384B*	1557602	980597	144251

Table D.2: Doors data. Data from the Doors benchmark.

D.3 Benchmark 3: Building or interpreting messages and log posts

Table D.3 lists the results obatined from running bencmark 3, sorted by number of process pairs. All numbers are in ns and averaged over 1 000 000 runs.

Msg size (B)	**Time** (ns)
32	546
64	1092
128	2185
256	4381
512	8742
1024	17483
2048	34967
4096	69654
8192	139283
16384	278552

Table D.3: Build data. Data from the Build benchmark.

D.4 Benchmark 4: Writing to log

Table D.4 lists the results obatined from running bencmark 4. Log size is in MB, message size is in B. All numbers are in ns, and are averaged over 1 000 000 runs.

Msg size	**Log size** (MB)							
(B)	*10*	*20*	*40*	*80*	*160*	*320*	*640*	*1280*
32	544	544	544	544	553	559	560	560
64	1088	1088	1089	1088	1105	1118	1120	1108
128	2176	2176	2179	2177	2211	2237	2210	2224
256	4352	4352	4352	4354	4423	4417	4398	4464
512	8747	8711	8706	8707	8801	8797	8780	9141
1024	17410	17410	17410	17414	17565	17539	17871	17882
2048	34819	34820	34823	34828	35088	35357	35366	35300
4096	69639	70025	69646	69668	70466	70803	70240	70157
8192	139995	139284	139298	139744	140532	140341	140022	139863
16384	278565	278579	278975	279169	280574	280051	279532	279267

Table D.4: Log data. Data from the log benchmark.

D.5 Benchmark 5: Synchronizing using POSIX Mutexes

Table D.5 lists the results obatined from running bencmark 5, sorted by number of mutexes. All numbers are in ns. Numbers in columms *Set* and *Release* are averaged over the corresponding times the mutex(es) is set and released in columm *Set/Relase*.

Mutex	Set/Release	Set	Release
1	*100000000*	182	170
10	*10000000*	83	71
100	*1000000*	73	61
1000	*100000*	98	77

Table D.5: Mutex data. Data from the mutex benchmark.

D.6 Benchmark 6: Synchronizing using System V Semaphores

Table D.6 lists the results obatined from running bencmark 6, sorted by number of process pairs. All numbers are in ns. Numbers in columms *Set* and *Release* are averaged over 1 000 000 runs.

Process pairs	Set	Release
1PP	2269	2275
2PP	2445	2453
4PP	5676	5713
8PP	11295	11323
16PP	21854	21738
32PP	42739	42100
64PP	84917	84255

Table D.6: Semaphores data. Data from the semaphore benchmark.