Blekinge Institute of Technology
Doctorial Dissertation Series No. 02/01
ISSN 1650-2159
ISBN 91-7295-004-8

Software Design Conflicts

Maintainability versus Performance and Availability

Daniel Häggander

Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Sweden

ISSN 1650-2159
ISBN 91-7295-004-8

Printed in Sweden
Kaserntryckeriet AB
Karlskrona, 2001

"Things are not always what they seem."

-- Aesop (620-560 B.C.)

This thesis has been submitted to the Faculty of Technology, Blekinge Institute of Technology, in partial fulfilment of the requirements for the Degree of Doctor of Philosophy in Engineering.

Contact Information:

Daniel Häggander
Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Soft Center
S-372 25 RONNEBY
SWEDEN

Tel.: + 46 457 385819
Fax.:+ 46 457 271 25
email: Daniel.Haggander@bth.se
URL: http://www.ipd.hk-r.se/dha

Abstract

A major goal in software engineering is to reduce the cost of maintaining software systems. Finding design methods which make software more easily maintainable has thus been one of the most prioritized challenges during the past decade. While mainstream software design has concentrated on maintainability, other software disciplines e.g. high-performance computing and high-availability systems, have developed other design methods which primarily support the quality attributes that are more important in their areas. More recently, demands have been made for high performance and high availability in typical mainstream software. At the same time, traditional high-performance and high-availability systems tend to incorporate more advanced business functionality, i.e. different software disciplines have started to converge. The situation is not unproblematic since the software design methods developed for achieving performance and availability may have been developed with a limited influence from maintainability, and vice versa. It is thus important to identify and analyze emerging design conflicts.

In this thesis I have studied conflicts between maintainability design methods on the one hand, and performance and availability methods and techniques on the other. I present the results of four case-studies involving four different applications. It is a characteristic of these applications that half of the system can be regarded as a telecommunications system and the other as a typical main-stream system, i.e. all systems make high demands on performance and availability but also very high demands on high maintainability. In studying these applications, I have identified two major conflicts: granularity in dynamic memory usage and source code size. My results show that these two conflicts can cause problems of such amplitude that some applications become unusable. I found that conflicts in certain situations are inherent; in other cases they can be avoided - or at least reduced - by adjusting the design methods used. I have also shown that conflicts may quite simply be a matter of misconceptions. Ten guidelines have been combined into a simple process with the aim of helping software designers to avoid and reduce conflicts. A method which automatically reduces the dynamic memory conflict in object-oriented applications written in C++ has been developed, implemented and evaluated. Finally, I have defined optimal recovery schemes for high availability clusters.

List of Papers

The present thesis is based on the following papers. References to the latter will be made using the roman numbers applied to the particular paper.

[I] Daniel Häggander and Lars Lundberg, "Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor", *in Proceedings of the 27th International Conference on Parallel Processing*, Minneapolis, USA, pp. 262-269, August 1998.

[II] Daniel Häggander and Lars Lundberg, "Memory Allocation Prevented Telecommunication Application to be Parallelized for Better Database Utilization", *in Proceedings of the 6th International Australasian Conference on Parallel and Real-Time Systems*, Melbourne, Australia, pp. 258-271, November 1999 (Springer Verlag).

[III] Daniel Häggander, PerOlof Bengtsson, Jan Bosch and Lars Lundberg, "Maintainability Myth Causes Performance Problems in Parallel Applications", *in Proceedings of the 3rd International IASTED Conference on Software Engineering and Applications*, Scottsdale, USA, pp. 288-294, October 1999.

[IV] Daniel Häggander and Lars Lundberg, "A Simple Process for Migrating Server Applications to SMPs", *Journal of System and Software 57,* pp. 31-43, 2001.

[V] Daniel Häggander, Lars Lundberg and Jonas Matton, "Quality Attribute Conflicts - Experiences from a Large Telecommunication Application", *in Proceedings of the 7th IEEE International Conference on Engineering of Complex Computer Systems*, Skövde, Sweden, pp. 96-105, June 2001.

[VI] Daniel Häggander and Lars Lundberg, "Attacking the Dynamic Memory Problem for SMPs", *in Proceedings of the 13th International ISCA Conference on Parallel and Distributed Computing Systems*, Las Vegas, USA, pp. 340-347, August 2000.

[VII] Daniel Häggander, Per Lidén and Lars Lundberg, "A Method for Automatic Optimization of Dynamic Memory Management in C++", *in Proceedings of the 30th International Conference on Parallel Processing*, Valencia, Spain, September 2001.

[VIII] Lars Lundberg and Daniel Häggander, "Recovery Schemes for High Availability and High Performance Cluster Computing", *Research Report 2001:06,* ISBN:1103-1581, submitted for journal publication.

Related papers and work not included in this thesis are listed below, in chronological order.

[IX] Lars Lundberg and Daniel Häggander, "Multiprocessor Performance Evaluation of Billing Gateway Systems for Telecommunication Applications", *in Proceedings of the 9th International ISCA Conference on Computer Applications in Industry and Engineering*, Orlando, USA, pp. 13-18, December 1996.

[X] Lars Lundberg and Daniel Häggander, "Bounding on the Gain of Optimizing Data Layout in Vector Processors", *in Proceedings of ICS 98, the ACM International Conference on Supercomputing*, Melbourne, Australia, pp. 235-242, July 1998.

[XI] Daniel Häggander and Lars Lundberg, "Multiprocessor Performance Evaluation of a Telecommunication Fraud Detection Application", *in Proceedings of the ARTES Graduate Student Conference*, Västerås, Sweden, pp. 33-39, May 1999.

[XII] Daniel Häggander and Lars Lundberg, "Ten Performance Guidelines for Balancing Software Quality Attributes when Developing Large Real-time Applications for Multiprocessors", *Technical Report 99:16,* ISSN:1103-1581, September 1999.

[XIII] Lars Lundberg, Jan Bosch, Daniel Häggander and PerOlof Bengtsson, "Quality Attributes in Software Architecture Design", *in Proceedings of the 3rd International IASTED Conference on Software Engineering and Application*, Scottsdale, USA, pp. 353-362, October 1999.

[XIV] Daniel Häggander, PerOlof Bengtsson, Jan Bosch and Lars Lundberg, "Maintainability Myth Causes Performance Problems in SMP Applications", *in Proceedings of the 6th IEEE Asian-Pacific Conference on Software Engineering*, Takamatsu, Japan, pp. 516-519, December 1999.

[XV] Daniel Häggander, "Software Design when Migrating to Multiprocessors", *Department of Information Technology, Uppsala University, in partial fulfilment of the requirements for the Degree of Licentiate of Engineering*, *ISSN:0283-0574, DoCS 112,* Uppsala, November 1999.

[XVI] Lars Lundberg, Daniel Häggander and Wolfgang Diestelkamp, "Conflicts and Trade-offs between Software Performance and Maintainability", a chapter in the book *"Performance Engineering. State of the Art and Current Trends",* pp. 56-67, 2001 (Springer Verlag).

[XVII] Magnus Broberg, Daniel Häggander, Per Lidén and Lars Lundberg, "Improving the Performance of Multiprocessor Memory Management in Java", *to appear in Java Report*.

[XVIII] Daniel Häggander, Per Lidén and Lars Lundberg, "A Method and System for Dynamic Memory Management in an Object Oriented-Program", *Patent Application No. 0002679-9*, Sweden, July 2000.

Acknowledgments

This work has been carried out as part of the ARTES project *"Design Guidelines and Visualization Support for Developing Parallel"* at Blekinge Institute of Technology, Sweden. ARTES is a national research programme supported by the Swedish Foundation for Strategic Research (SSF).

First of all I would like to thank my supervisor *Professor Lars Lundberg*, whose skills in computer systems and unlimited patience have been a great benefit. Secondly, I wish to thank my previous supervisor, *Professor Hans Hansson*, whose tremendous ability to solve practical problems has been invaluable.

I wish to express my thanks to Ericsson Software Technology and the members of the FCC, BGw, DMO, SDP and READ projects, particularly *Jonas Matton*, for giving me the opportunity to study their work. Further, I thank SUN Microsystems and *Pär Känsälä* for all their help with multiprocessor hardware.

I would like to thank the my colleagues at the Department of Software Engineering and Computer Science at Blekinge Institute of Technology, and in particular *PerOlof Bengtsson*, *Magnus Broberg* and *Per Lidén*. A big thanks also goes to *Jane Mattison* for helping me with proof reading.

Contents

Software Design Conflicts

Maintainability versus Performance and Availability

1. Introduction

A prioritized goal in software engineering is to reduce the cost of maintaining software systems. A strong focus on software maintenance is considered important since maintenance is responsible for the major part of the costs in a product's life-cycle [Pigoski 97].

Finding design methods which make software more easily maintainable has thus been one of the greatest challenges during the past decade, at least for mainstream software design. A strong focus on maintenance-related quality attributes such as maintainability, variability and modifiability, to mention just a few, has resulted in a number of new design methodologies, the best known of which include object-orientation, frameworks, multi-layer design and design patterns. A software designer today is expected to have knowledge of these methodologies and to know how to use them.

While mainstream software design has concentrated on maintainability, other software disciplines have continued to develop new design methods that are effective in their own particular area. High-performance computing, often concentrating on scientific application, has, with the aid of parallel processing technology, constantly improved performance limits. In hardware, the results are Symmetric MultiProcessors (SMP) and the dawn of onChip Multiprocessors (CMP). Multiprocessor technology is dependent on parallel execution, i.e. methods for developing parallel software have

been developed in order to benefit from the high capacity of multiprocessors; a widely used example is multithreading.

Another system design area is high-availability systems. The market for this kind of software has been limited primarily to critical systems, e.g. airplanes and telecommunication equipment. Historically, these systems have normally provided a limited degree of business functionality and have often been built on propriety hardware and software. It is in this context that software design methods supporting high-availability have been developed.

More recently, demands have been made for high performance and high availability also for mainstream software applications. At the same time, traditional high-performance and high-availability software applications tend to incorporate more advanced business functionality, i.e. different software disciplines have started to converge. As a result, it has become necessary to consider maintainability with respect to other quality attributes, e.g. performance and availability, when designing application software.

The situation is not unproblematic since the methods developed for achieving performance and availability may have been developed with a limited influence from maintainability, and vice versa. Methods developed in different disciplines may at worst be incompatible, i.e. the convergence uncovers software design conflicts. These may come as a surprise to developers.

It is thus important to identify and quantify emerging design conflicts. Further, conflicts must be investigated in detail in order to find possible strategies for avoiding or reducing them. In some cases, it may just be a matter of small method adaptations. In other cases, however, alternative design methods may prove necessary.

Identifying and defining conflicts on a quality-attribute level creates a number of difficulties. One major problem is that the definition, scope and meaning of quality attributes, especially maintainability, are not unanimously and objectively understood. Another problem is that conflicts between quality attributes are often indirect, i.e. created by the design methods used to improve the quality attributes rather than the quality attributes themselves.

In this thesis, I have studied conflicts between maintainability design methods on the one hand, and performance and availability methods and techniques on the other. I present the results of four case-studies involving four different applications: Billing Gateway (BGw), Fraud

Control Center (FCC), Data MOnitor Server (DMO) and Service Data Point (SDP), respectively. Three of the applications (BGw, FCC and SDP) have been developed by Ericsson Software Technology AB. They have, however, all been developed by different departments and different designers. The fourth application (DMO) was developed by students at Blekinge Institute of Technology, and was based on requirements defined by Ericsson Software Technology AB. All applications are what we refer to as "telecommunication support systems". It is a characteristic of this kind of systems that one part of the system can be regarded as a telecommunications system and the other as a typical main-stream system, i.e. a system which has telecommunication characteristics but also advanced business functions. The "in-between" position makes these systems interesting objects of study in software design conflicts.

In studying these applications, I have identified two major conflicts: granularity of dynamic memory usage, and source code size. My results show that these two conflicts can cause problems of such amplitude that some applications become unusable. I found that conflicts in certain situations are inherent; in other cases they can be avoided - or at least reduced - by adjusting the design methods used. I have also shown that conflicts may quite simply be a matter of misconceptions.

Ten guidelines have been combined into a simple process with the aim of helping software designers to avoid and reduce conflicts. A method which automatically reduces dynamic memory conflict in C++ has also been developed and tested on synthetic programs as well as in one of the case-study applications. Finally, I have suggested a method which simplifies the definition of favorable recovery schemes for high-availability and high-performance cluster systems.

The remaining pages are organized as follows. Section 2 presents definitions of the three quality attributes considered in this thesis. An overview of the research methods is given in Section 3. The results, in the form of an overview as well as individual summaries of eight papers, are presented in Section 4. In Section 5, prior and related work is provided and references are listed in Section 6. A paper section, including the eight papers on which this thesis is based, ends the thesis.

2. Quality Attribute Definitions

This section elaborates on definitions of three quality attributes: performance, availability and maintainability. Official definitions from [IEEE 90], [ISO 00] and [McCall 94] (The Encyclopedia of Software Engineering, 1994) are presented. The definitions, scope and meaning of quality attributes are, however, not unanimously or objectively understood, especially not maintainability [Kajko-Mattson 01]. This is, unfortunately, also a problem in the present thesis. In cases where I have used a definition of a quality attribute which differs from the official one, I have presented a definition of my own. In providing such definitions I do not mean to suggest that they are necessarily better or more correct than ready-existing ones; my hope is quite simply that they will make my thesis more readily understandable.

2.1 Quality Attribute

> "A quality attribute is a feature or characteristic that affect an item's quality. Syn: Quality factor", [IEEE 90]

This thesis focuses on the quality of an application, i.e. a combination of software and hardware. The definition used in the present thesis is thus:

> "A quality attribute is a non-functional feature or characteristic that affect an application's quality" - definition used in this thesis

2.2 Performance

> **Performance** - "The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage", [IEEE 90]

In many reports which discuss software quality attributes, attribute performance is excluded. However, a related attribute which is often used is efficiency.

> **Efficiency (time-behavior)** - "The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions", [ISO 00]

The Encyclopedia of Software Engineering (1994) has a slightly different definition of efficiency, in which the latter is seen as a function of hardware.

> **Software efficiency** - "The amount of computing resources required by a program to perform a function", [McCall 94]

The performance definition used in this thesis is a combination of the first two definitions, [IEEE 90] and [ISO 00], respectively.

> **Performance** - "The degree to which an application accomplishes its designated functions within given constraints of appropriate response and processing times and throughput rates, under stated conditions" - definition used in this thesis

2.3 Availability

> **Availability** - "The degree to which a system or components is operational and accessible when required for use. Often expressed as a probability", [IEEE 90]

In ISO/IEC FDIS 9126-1:2000(E) availability is not included as a separate characteristic (attribute). Instead, ISO defines availability as a combination of maturity, fault tolerance and recoverability.

> **Maturity** - "The capacity of the software product to avoid failure as a result of faults in the software", [ISO 00]

> **Fault tolerance** - "The capability of the software product to maintain a specific level of performance in case of a software fault or of infringement of its specified interface", [ISO 00]

> **Recoverability** - "The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure", [ISO 00]

There are no conflicts between the various definitions. The official definitions also comply with those used in this thesis. I have not made any attempt to measure the availability of the applications studied. When the level of availability is discussed, the discussion is based on previous experience and theoretical calculations based on figures given by equipment vendors.

2.4 Maintainability

> **Maintainability** - "The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment", [IEEE 90]

> **Maintainability** - "The capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in environment, and in requirements and functional specifications", [ISO 00]

The Encyclopedia of Software Engineering (1994) has a narrower definition of maintainability. Changes in environment or in requirements are not included in its definition of maintainability.

> **Maintainability** - "Effort required to locate and fix an error in an operational program", [McCall 94]

The first two definitions eliminate the possibility of maintainability conflicting with some other attribute, thereby rendering a discussion about conflicts related to maintainability pointless.

A limitation of the scope to corrections, improvements or adaptations of the software to changes in the functional specifications makes the discussion of emerging conflicts more comprehensible.

With such a definition, maintainability can cause software design conflicts when it is required along with another attribute e.g., a design method used to improve maintainability can be in conflict with design

methods normally used to fulfil other quality attributes such as performance and/or availability.

> **Maintainability** - "The capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in the functional specifications" - definition used in this thesis

In the studies considered here I have occasionally used source code size (lines of code) to measure maintainability. Measuring maintainability by means of the source code size is not a generally accepted method (there is indeed no generally accepted method). However, previous research indicates that maintenance costs are strongly related to the amount of source code [Granja & Barrancp-Garcia 97] [Li & Henry 93].

3. Research Method

The overall research goal of this thesis is to:

> *Identify, quantify, analyze, and propose solutions for handling software design conflicts emerging when building maintainable applications with explicit and high demands on performance and availability.*

In order to achieve my goal I have performed empirical studies on industrial cases, conducted interviews with designers and carried out experiments on synthetics programs as well as developed several prototypes in collaboration with industrial partners.

My thesis is divided into two parts: in the first I identify, quantify and analyze software design conflicts; and in the second I evaluate and propose solutions which avoid or reduce software design conflicts. The rest of this section gives an overview of the research methods used. More detailed information about the research methods used is given in the individual papers.

3.1 Identify, Quantify and Analyze Conflicts

In order to identify, quantify and analyze design conflicts I have studied four different major telecommunication support applications written in C++ for use with UNIX. Three of the applications have been developed by Ericsson Software Technology AB. These are referred to as BGw, FCC, DMO and SDP. The last but one, DMO, was developed by students at Blekinge Institute of Technology. In three applications (BGw, FCC and DMO) maintainability is strongly prioritized. These applications are also very demanding with respect to performance due to real-time requirements on throughput and response time.

Symmetric multiprocessors and multithreaded programming have been used in order to give these applications a high, scalable performance. In the first three applications I studied conflicts between design methods for obtaining maintainability and methods for achieving performance. In the fourth study, the SDP case-study, I also had the opportunity to study availability aspects since this application

makes high demands on availability. The papers which identify, quantify and analyze software design conflicts are I, II, III, IV and V.

3.2 Avoiding and Reducing Conflicts

Alternatives for avoiding or reducing software design conflicts can be divided into three groups: adjustment, exchange and compensation. In many situations it is quite simple to adjust a design method in such a way that the conflict is eliminated or reduced to an acceptable level. However, in some situations, the conflict is inherent, necessitating an exchange of at least one of the design methods. If the conflict is found at an early stage, it is often possible to find an alternative design method which is more tolerant. However, if the conflict is discovered in a late development phase or after delivery, compensating methods may prove very useful. By compensating methods I mean methods which reduce the effect rather than the actual cause of the conflict. Such methods have also shown to be cost effective when conflicts are identified at earlier stages. The papers which suggest solutions as to how to avoid and reduce conflicts are IV, V, VI, VII and VIII.

4. Research Results

The first case-study was made on the BGw application. This is a mediation server written in multithreaded C++ and executed in UNIX on a symmetric multiprocessor. We found that the application has poor scale-up due to a bottleneck within the dynamic memory management. Two approaches to remove the bottleneck were evaluated: replacing the standard heap with a parallel heap (ptmalloc [Gloger]); and optimization of the application design by removing a number of heap allocations/deallocations respectively. Both methods were found to be efficient. However, the latter approach also improves performance when running the application on a single-processor computer.

The same bottleneck was identified also in the next case-study (FCC), even though FCC is based on a third party relational database management system (RDBMS). The study showed that the bottleneck could be removed either by dividing FCC into a number of individual processes or by using ptmalloc. Using interviews with the developers of the FCC as a base, we established that the aim of developing a maintainable application was the major reason for the increased number of dynamic memory allocations, i.e. which caused the bottleneck. Our evaluation shows that an alternative design, based on rigid but exchangeable components, has much better performance characteristics.

Using the results from the BGw and FCC studies as well as a study of a third application (DMO), I defined a simple design process based on ten guidelines. The process was designed with the aim of helping designers of applications to become more efficient in creating a balance between SMP performance and maintainability.

In the next case-study I checked if the quality attribute “conflict” which was identified in previous case-studies is bidirectional, i.e. if applications developed with a strong focus on run-time qualities such as performance and availability uncover unknown software design conflicts, resulting in maintainability problems. This was made possible by studying yet another telecommunication application (SDP). The application is part of a system which rates pre-paid subscribers in real-time. The result is on source code categorization, which showed that 85% of the source code was written in response to performance or availability demands. Work carried out by other researchers indicates that maintenance costs are strongly related to the amount of application code, i.e. the large amount of source code is a strong indication that

performance and availability have had a negative impact on maintainability.

Earlier studies carried out by the author indicate that methods used for obtaining maintainability often result in fine-grained dynamic memory usage, whereas a fine-grained dynamic memory usage can substantially damage performance, especially in multithreaded C++ applications running on SMPs.

To sum up, two major conflicts have been identified: granularity in dynamic memory usage, and source code size. My results show that these two conflicts can cause problems of such amplitude that some applications become unusable. I found that conflicts in certain situations are inherent; in other cases they can be avoided - or at least reduced - by adjusting the design methods. I have also shown that conflicts may quite simply be a matter of misconceptions.

The simplest way, we have discovered, to attack the dynamic memory problem is to optimize the C library memory allocation routines. This is because such an optimization has low impact on the implementation of the application. However, reducing the usage of short-lived memory allocations gives the best result in terms of performance.

I have thus developed a pre-processor-based method named Amplify which can reduce the use of small and short-lived memory allocations in (object-oriented) C++ applications. Amplify does this by speculating on the temporal locality of dynamic memory allocations. Amplify makes it possible to achieve two goals at the same time. First, using Amplify is not more complicated than replacing the C library memory allocation routines. Second, the optimization is almost as effective as a reduction of the usage of short-lived memory allocation would be.

One software design conflict identified was source code size. Handling high-availability issues within the source code is one of the reasons for excessive code size. An example is fail-over cluster schemes defined in the application code. I have defined a very simple method for obtaining general recovery schemes which are optimal for a number of important cases, making it possible to move this feature from the application source code to cluster software.

The rest of the section presents individual summaries of the eight papers on which this thesis is based. I also acknowledge where results have been achieved with the aid of anyone other than my supervisor, professor Lars Lundberg.

Paper I: Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor

This paper reports the results from performance evaluations made on a large real-time application (Billing Gateway) executing on symmetric multiprocessors. The Billing Gateway (BGw) collects billing information about calls from mobile phones. The application has been developed by Ericsson. BGw is written in C++ (approximately 100,000 lines of code) using object-oriented design, and the parallel execution has been implemented using Solaris threads [Lewis 96].

The parallel software design led us to believe that the application would scale-up well on a Symmetric MultiProcessor (SMP) [Hwang & Xu 98]. Performance evaluations showed, however, that the benefit of executing on multiprocessors was relatively small [IX]. One of the BGw versions failed to scale up at all (see Figure 1).

The major reason for this lack of benefit was that the standard sequential dynamic memory management combined with a large number of dynamic memory allocations and deallocations. The large number of allocations and deallocations is a result of the way in which the object-oriented design was used. This paper compares two approaches for improving the performance of the BGw: replacing the standard heap with a parallel heap (ptmalloc [Gloger]); and optimization of the application design by removing a number of heap allocations/deallocations respectively. The number of heap allocations/ deallocations was reduced by introducing object-pools for commonly used object types, and by replacing heap variables with stack variables. The time spent on re-designing the BGw was rather limited (one week). The parallel heap approach resulted in dramatic scale-up improvement. The optimization of the application design also resulted in a dramatic scale-up improvement. This approach also improves performance when the application is executed on a single-processor computer.

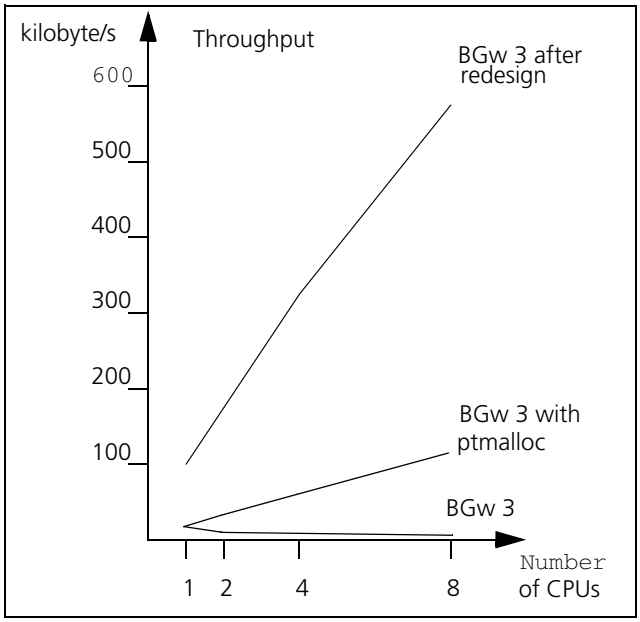


Figure 1. *The processing capacity of BGw (version 3) on an 8-way SMP.*

Paper II: Memory Allocation Prevented Telecommunication Application to be Parallelized for Better Database Utilization

This paper replicates the results of paper I on yet another application, the Fraud Control Center (FCC). The latter application has been developed by Ericsson and is part of an anti-fraud system that combats mobile telephone fraud with the aid of real-time analysis of network traffic [Lundin et al. 96]. It is based on a commercial relational database management system [Connolly at al. 96]. The parallelization of the FCC application using multithreaded programming was not only designed to scale-up the multithreaded part of the application but also to improve the performance of the database system [XI].

A database server cannot usually be fully utilized using a single client, particularly if the database server is executed on a symmetric multiprocessor. Multithreading makes it possible for an application implemented as a single process to have multiple database clients. This study shows, however, that object-oriented development - more specifically, its impact on dynamic memory management - prevented

the application from benefiting from multiple database clients. Although the database module performed most of the work, the limiting factor in the FCC was the poor performance of the multithreaded client part. Our use of a parallel heap implementation (ptmalloc [Gloger]) for optimizing the dynamic memory management in the FCC proved successful. We also showed that the performance characteristics of the multithreaded client version using ptmalloc were very similar to a version in which the clients executed in separate processes.

Paper III: Maintainability Myth Causes Performance Problems in Parallel Applications

Papers I and II show that multithreaded applications developed with object-oriented methodologies have a potential bottleneck within their dynamic memory management. The main reason is that object-oriented methodologies tend to increase the number of dynamic memory allocations. Our evaluations also show that restricting the use of object-oriented design significantly reduces the risk of congestion within the dynamic memory management.

Using interviews with the developers of the FCC as a base, paper III identifies maintainability together with flexibility as an indirect reason for the large number of dynamic memory allocations occurring. The assumption was that adaptable object designs (e.g. design patterns [Gamma et al. 97]) would give a more maintainable application.

An alternative design, based on rigid but exchangeable components, was, however, implemented and evaluated. Maintainability was predicted using a state-of-the-art method [Bengtsson et al. 99]. The evaluation results show that the alternative design has superior performance characteristics as well as higher maintainability. These findings show that the design decision chosen was based on an assumption which subsequently proved invalid, i.e. the performance problems in FCC were based on a myth.

This is a joint paper written in collaboration with PerOlof Bengtsson. The latter performed the maintenance evaluations; the interviews were carried out together.

The paper is also available in an earlier version [XIV].

Paper IV: A Simple Process for Migrating Server Applications to SMPs

The first three studies (papers I, II and III) all show a bottleneck within the dynamic memory management. The studies also showed that the bottleneck was not anticipated by the developers and resulted in a substantial decrease in performance. In all cases, however, the bottleneck could easily be removed by choosing an alternative design strategy; it could also be substantially reduced by applying a dynamic memory allocator optimized for SMPs. Further, all applications had been developed under approximately the same pre-conditions. The developers were thoroughly trained in modern software engineering, including object-oriented design, frameworks and design patterns. The applications include a substantial amount of third-party tools and software. The performance requirements were mainly met by using the latest in high-performance, off-the-shelf uni-processor hardware. However, higher requirements with regard to performance, or complementary requirements with respect to scalable performance necessitated a migration from a uni-processor to a multiprocessor architecture, i.e., SMP.

Designing efficient applications for SMPs is a vital task since communication and synchronization can easily limit the scale-up to a considerable degree. Efficient strategies for designing and implementing multiprocessor applications are widely recognized [Foster 95]. In this type of application, however, the parallel performance must be balanced against other quality attributes such as maintainability and flexibility.

For developers used to developing maintainable and flexible software but not familiar with parallel hardware architecture and parallel software, making the right trade-offs for higher multiprocessor performance can be very difficult, if not impossible. However, our results show that large improvements can be made by just making the developers aware of the problem with dynamic memory allocations on SMPs.

Using the results from papers I, II and III and a study of a third application (DMO), I have defined a simple process based on ten guidelines [XII]. The process is designed with the aim of helping designers of applications to become more efficient in creating a balance between SMP performance and maintainability.

Paper V: Quality Attribute Conflicts - Experiences from a Large Telecommunication Application

Previous research (papers I, II, III and IV) has shown that a strong focus on quality attributes such as maintainability and flexibility can result in less efficient applications, at least from a performance perspective. One of the main reasons for this is that applications developed using these new methodologies are more general and dynamic in their design. This is an advantage in terms of maintainability but gives poorer application performance. As a result, the ambition of building maintainable systems often results in poor performance.

Performance is often gained by customizations, i.e., by using less generic solutions. Common examples are application-specific buffering of persistent data and bypassing layers in multi-layer architectures. Customization of functions tends to increase the amount of application code since code that is normally located in the operating system or in third party software must be dealt with on an application level. Work carried out by other researchers indicates that maintenance costs are strongly related to the amount of application code [Granja & Barrancp-Garcia 97] [Li & Henry 93].

One hypothesis is thus that the quality attribute "conflict" seen in previous case-studies is bidirectional i.e., if applications developed with a strong focus on run-time qualities such as performance and availability uncover unknown software design conflict, with maintainability problems as a result. Note that I now also introduce the notion of availability into my reasoning. A second hypothesis is that the best balance between performance and availability on the one hand and maintainability on the other is obtained by focusing on maintainability in the software design, and by using high-end hardware and software execution platforms for solving the majority of the performance and availability issues. It has at least been shown that it is relatively easy to achieve high performance in applications developed with a strong focus on maintainability with the aid of SMPs [IV].

The aim behind paper V was to check if the above two hypotheses were correct. This was made possible by studying yet another telecommunication application. The latter is part of a system which rates pre-paid subscribers in real-time. The system has been developed for the cellular market by the Ericsson telecommunication company.

Rating of pre-paid subscribers is a mission-critical task which requires extremely high performance and availability.

The first of three results related to source code categorization, which showed that 85% of the source code was dictated by performance or availability. A result which supports the first hypothesis. The second result was a prototype developed according to a three-step process prioritizing maintainability. The prototype contained less than 7,000 LOC (lines of code) and was capable of handling three times more calls than the original application. The additional cost for the platform was 20-30% for the prototype as compared to the old application. A result which supports the second hypothesis. The most useful design strategies were extracted as three guidelines, which comprised the third result.

This study was carried out with the aid of Jonas Matton. The latter performed the source-code measurements as part of his master's thesis.

Paper VI: Attacking the Dynamic Memory Problem for SMPs

Dynamic memory management has constituted the most serious serialization bottleneck in many of the applications studied. The dynamic memory problem has been addressed in a number of ways. In this paper, we summarize and structure our experiences of some of these. There are two basic ways of attacking the problem: either you reduce the cost of using dynamic memory, or you reduce the use of short-lived dynamic memory allocations (see Figure 2 on the following page).

The problem can also be addressed at different levels, i.e. the operating system, the implementation, and the software architecture and design levels. Each of the investigated ways of attacking the dynamic memory problem has its own particular advantages and disadvantages.

I argue that one should focus on the operating system level when dealing with an existing code, and on the software architecture level when developing new applications.

Attacking the dynamic memory problem for SMPs

Level \ Type	Reduce cost	Reduce usage
Architecture and Design	Processes instead of threads	Rigid but exchange-able components
Implementation	Object pools (Customized allocator)	Use stack instead of heap
Operating System	Parallel allocator (General propose allocator)	No method found

Figure 2. *Categorization of methods which can be used to attack the dynamic memory problem on an SMP.*

Paper VII: A Method for Automatic Optimization of Dynamic Memory Management in C++

Paper VI points out that the simplest way to attack the dynamic memory problem is to optimize the C library memory allocation routines. However, the results discussed in paper I show that it can be more efficient to address the problem on the source code level, i.e., modify the application's source code. Such an approach makes it possible to achieve more efficient and customized memory management. To implement and maintain such design and source code optimizations is, however, both a laborious and costly since the procedure must be done manually.

Reducing the use of short lived memory allocation gives the best results in terms of performance. Attacking the problem at an operating system level is the most effective method in terms of man hours. Consequently, the most favorable method is, from our perspective, found down to the right in the matris (Figure 2), the vacant spot. A challenge is thus to fill this vacancy.

Paper III shows that the large amount of dynamic memory is to a considerable extent the result of designers' attempts to increase maintainability, i.e., make the application easier to adjust to new or modified requirements. Such adjustments are not usually made in run-time; temporal locality (defined in the paper) will thus characterize the run-time behavior of the dynamic memory allocations in the applications.

This paper presents an implementation of a pre-processor based method named Amplify which reduces the usage of small and short-lived memory allocations in (object-oriented) C++ applications. Amplify does this by speculating on the temporal locality of dynamic memory allocations. Test results show that Amplify can obtain significant speed-up in synthetic applications, and that it can also be effective on real applications (BGw). The methods is patent pending [XVIII].

This paper was complied with the help of Per Lidén. The latter aided me with the design and implementation of the pre-processor.

The increasing popularity of Java makes it interesting to determine if Java applications have similar performance problems with dynamic memory as C++ has, and if the same method, effective for C++ applications, is also useful for applications written in Java. To do this is, however, a not unproblemetic task since there are a number of fundamental differences between C++ and Java, e.g. language constraints, garbage collection and the virtual machine concept. These differences make it not only necessary to re-implement Amplify in Java, but also to reconsider some of the basics of the method (see [XVII]).

Paper VIII: Recovery Schemes for High Availability and High Performance Cluster Computing

One way of obtaining high-availability is to distribute an application in a cluster or a distributed system [Pfister 98]. If a computer breaks down, the functionality performed by that computer will be handled by some other computer in the cluster. Many cluster vendors support this kind of fault recovery, e.g., Sun Cluster [Sun Cluster], MC/ServiceGuard (HP) [MC/ServiceGuard], TrueCluster (DEC) [TrueCluster], HACMP (IBM) [Ahrens et al. 95], and MSCS (Microsoft) [Vogels et al.]. One way to define how functionality is redistributed in the clusters is to define recovery schemes. Lundberg et al. have proposed a recovery scheme that is optimal for a number of important cases and which defines a bound on the performance of the recovery schemes for any number of computers. In this paper we have attacked the same problem; we have, however, reformulated the problem definition, and suggested a simple algorithm which generates recovery schemes that are better than

those defined in [Lundberg & Svahnberg 99]. The bounds of the recovery schemes have been decided using the same methods as in [X].

This is a joint paper written in collaboration with Lars Lundberg. My contribution was the reformulation of the problem and the definition of the algorithm.

5. Related Work

This section presents prior and related work. It has been divided into three parts: quality attributes, dynamic memory allocators and recovery schemes for high-availability clusters. Additional prior and related work is presented in the individual papers (see paper section)

5.1 Quality Attributes (Factors)

The concept of quality factor originated in the late 1970s in conjunction with the research and development of software measurement technology. The factors provide the definition of quality required for a software product. The early work consisted of identifying sets of factors or characteristics, related attributes and criteria, and metrics. Further developments during the 80s have refined the factors or added additional factors for consideration. A survey of identified factors, attributes, criteria and metrics can be found in [McCall 94], where they are assembled into a framework with the aim of capturing relations between quality factors, related attributes and criteria, and metrics. It is interesting to note that the effectivity quality factor is in conflict, or, as McCall so logically argues, “can be more costly to achieve together” with all other factors except correctness and reliability (see Figure 5, p. 967 in [McCall 94]). The framework also defines relations between quality factors and system characteristics, see Table 6, p. 968. Of eleven listed system characteristics, only two, according to McCall, are related to efficiency; these are real-time applications and on-line usage. Introduction of software quality factors such as maintainability, flexibility and portability into real-time applications can thus be expected to be very costly. Further, the introduction of on-line usage into applications which have previously been executed off-line, is, as the framework shows, likely to lead to quality factor conflicts.

The conclusions outlined in Mcall's tables and classification comply well with the emerging conflicts I have found in the applications studied in this thesis.

Finding the right balance of quality attributes is thus of utmost importance. One possible method is to reduce the conflicts already in the requirement setting. Research has been conducted in this area, and methods and knowledge-base tools have been developed with the aim of

helping users, developers, and customers to analyze and identify conflicts among the requirements [Boehm & In 99]. However, the conflicts found using these methods and tools are far from complete, and even if the conflicts are identified, they may be unavoidable and must thus be dealt with when the system is designed, dealing with design conflicts is therefore still highly relevant.

A number of methods developed to identify conflicts on an architecture level have been proposed. Kazman et al. have developed the Architecture Trade-off Analysis Method (ATAM) which addresses the need for trade-offs between different quality attributes [Kazman et al. 99]. The main goal of ATAM is to illuminate risks early on in the design process. However, ATAM does not contain any concrete guidelines regarding software architecture and design. Lundberg et al. have developed a set of guidelines and a simple design process that can be used for making design and architectural trade-offs between quality attributes [XIII] [XVI]. Kruchten has suggested the "4+1 View Model" which makes it possible to communicate and engineer different aspects of a software system, e.g., development as well as run-time aspects [Kruchten 95]. In this way, the probability of identifying conflicts can be increased. The goal of my work has not been to investigate or make a survey of quality attribute "conflicts"; instead, I have studied software design conflicts; the quality attribute "conflicts" are what give my work context and relevance.

5.2 Dynamic Memory Allocators for C++ and SMPs

Dynamic storage allocation (DSA) and memory management (DMA) is a large area. Ben Zorn maintains a repository with related information [Zorn]. This repository includes information about explicit allocation mechanisms such as malloc/free and automatic storage reclamation algorithms such as garbage collection. Wilson et al. have made an excellent survey of literature and allocators [Wilson et al. 95]. Berger et al. have recently presented a flexible and efficient infrastructure for building memory allocators based on C++ templates and inherence. The main intention is to help programmers to build customized allocators [Berger et al 01]. Reports are also available on hardware support for dynamic memory management [Chang et al. 00]. Henceforth, this section will only focus on malloc/free allocators design for SMPs.

The allocator designed by Doug Lea is both fast and efficient [Lea]. However, the original version is neither thread-safe nor scalable on SMPs. Using Doug Lea's implementation, Wolfram Gloger created a thread-safe and scalable version called ptmalloc [Gloger]. The allocator is based on a multiple number of sub-heaps. When a thread is about to make an allocation it "spins" over a number of heaps until it finds an unlocked heap. The thread will use this heap for the allocation and for all future allocations. If an allocation fails, the thread “spins” for a new heap. Since the operating system normally keeps the number of thread migrations low this implementation works well for a number of applications. However, when the number of threads is larger than the number of processors, ptmalloc may prove unpredictable in terms of efficiency. We have successfully used this heap implementation in our own studies.

An SMP version of SmartHeap is commercially available [Microquill]. We have not had the opportunity to test this implementation on our synthetic programs since the software is not readily accessible. During our test on BGw, however, we were able to make some benchmark tests on this allocator.

Hoard [Berger et al. 00] is another allocator optimized for SMPs which we successfully used in one of our studies. This allocator focuses on avoiding false memory sharing and blowup in memory consumption. The allocator is scalable. However, we have found that Hoard has problems when threads frequently migrate between processors, e.g. when the number of threads is larger than the number of processors.

LKmalloc [Larson & Krishan 98], developed by Larsson and Krishnan, is yet another parallel memory allocator (this has not been investigated by us). It is intended to be fast and scalable in traditional applications as well as long-running server applications executed on multiprocessor hardware.

Prior to ptmalloc, Hoard, LKmalloc etc., - all of which are relatively recent contributions - there is little work available on parallel memory allocators.

The major difference between my work on dynamic memory allocation and the work described above is that I do not address the area of dynamic memory management in general. My concern is the specific problems which occur when aiming simultaneously at maintainability and performance, i.e., how design strategies for maintainability stress

the use of dynamic memory and how to avoid this becoming a bottleneck in applications running on SMPs.

5.3 Recovery Schemes for High-Availability Clusters

The problem of finding optimal recovery schemes for clusters of computers seems to be NP-hard [Garey & Johnson 79]. Lundberg et al. have proposed a recovery scheme which is optimal for a number of important cases and have defined a bound on the performance of the recovery schemes for any number of computers [Lundberg & Svahnberg 99]. Similar types of bounds have proven useful for other NP-hard resource allocation problems [Lundberg & Lennerstad 98].

6. References

[Ahrens et al. 95] G. Ahrens, A. Chandra, M. Kanthanathan and D. Cox, "Evaluating HACMP/6000: A Clustering Solution for High Availability Distributed Systems", *in Proceedings of the 1995 Faoult-Tolerant Parallel and Distributed System Symposium*, pp 2-9, 1995.

[Bengtsson et al. 99] P. Bengtsson, J. Bosch, "Architecture Level Prediction of Software Maintenance", *in Proceedings of 3rd European Conference on Maintenance and Reengineering*, Amsterdam 1999.

[Berger et al. 00] E. D. Berger, K. McKinley, R. Blumofe and P. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications", *in Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.

[Berger et al 01] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Composing High-Performance Memory Allocators", *in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[Boehm & In 99] B. Boehm and H. In, "Identifying Quality-Requirement Conflicts", *IEEE Computer*, August 1999, pp. 27-33.

[Chang et al. 00] J. M. Chang, W. Srisa-an, C. D. Lo, and E. F. Gehringer, "Hardware support for dynamic memory management", *in Proceedings of the Workshop for Solving the Memory-Wall Problem, at the 27th International Symposium on Computer Architecture,* Vancouver, BC, June 2000.

[Connolly at al. 96] T.Connolly, C. Begg and A. Strachan, *"Database Systems"*, Addison-Wesely, 1996.

[Foster 95] I. Foster, *"Designing and Building Parallel Programs"*, Addison Wesley, 1995.

[Gamma et al. 97] E. Gamma, R. Helm, R. Johnson, J. Vilssides, *"Design Patterns"*, Addison-Wesley, 1997.

[Garey & Johnson 79] M. R. Garey and D. S. Johnson, *"Computers and Interactability"*, W. H. Freeman and Company, 1979.

[Gloger] W. Gloger, "Dynamic memory allocator implementations in Linux system libraries", *http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html (site visited August 14th, 2001).*

[Granja & Barrancp-Garcia 97] J. C. Granja-Alvarez and J. Barrancp-Garcia, "A Method for Estimating Maintenance Cost in a Software Project: A Case Study", *Journal of Software Maintenance: Research and Practice*, John Wiley & Sons, Volume 9, pp. 166-175, 1997.

[Hwang & Xu 98] K. Hwang, Z. Xu, *"Scalable and Parallel Computing"*, WCB/McGraw-Hill, 1998.

[IEEE 90] IEEE, "IEEE Standard Glossary of Software Engineering Terminology", *IEEE Std 610.12-1990*, 1990.

[ISO 00] ISO 9126, "Software Qualities", *SO/IEC FDIS 9126-1:2000(E)*, 2000.

[Kajko-Mattson 01] M. Kajko-Mattson, "Can we Learn Anything from Hardware Preventive Maintenance", *in Proceedings of the 7:th IEEE International Conference on Engineering of Complex Computer Systems*, Skövde, Sweden, pp. 106-111, June 2001.

[Kazman et al. 99] R. Kazman, M. Barbacci, M. Klein, S.J. Carrière, "Experience with Performing Architecture Tradeoff Analysis", in Proceedings of the International Conference on Software Engineering, Los Angeles, USA, May 1999, pp. 54-63.

[Kruchten 95] P. Kruchten, "The 4+1 View Model of Architecture", *IEEE Software*, November 1995, pp. 42-50.

[Larson & Krishan 98] P. Larson and M. Krishan, "Memory Allocation for Long-Running Server Applications", *in Proceedings of the International Symposium on Memory Management*, Vancouver, British Columbia, Canada, October, 1998.

[Lea] D. Lea, "A Memory Allocator", *http://g.oswego.edu/dl/html/malloc.html (site visited August 14th, 2001).*

[Lewis 96] B. Lewis, *"Threads Primer"*, Prentice Hall, 1996.

[Li & Henry 93] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability", *Journal of Systems Software*, 1993;23, pp. 111-122.

[Lundberg & Lennerstad 98] L. Lundberg and H. Lennerstad, "Using Recorded Values for Bounding the Minimum Completion Time in Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, pp. 346-358, April, 1998.

[Lundberg & Svahnberg 99] L. Lundberg and Charlie Svahnberg, "Optimal Recovery Schemes for High-Availability Cluster and Distributed Computing", *in Proceedings of the 6th International Australasian Conference on Parallel and Real-Time Systems*, Melbourne, Australia, pp. 153-167, November 1999 (Springer Verlag).

[Lundin et al. 96] C. Lundin, B. Nguyen and B. Ewart, "Fraud management and prevention in Ericsson's AMPS/D-AMPS system", *Ericsson Review* No. 4, 1996.

[McCall 94] J. A. McCall, "Quality Factors", *Encyclopedia of Software Engineering*, Volume 2, pp. 958-969, 1994.

[Microquill] Microquill, "SmartHeap for SMP", *http://www.microquill.com/smp, (site visited August 22nd, 1999).*

[MC/ServiceGuard] MC/ServiceGuard, "Managing MC/ServiceGuard for HP-UX 11.0", *http://docs.hp.com/hpux/ha.*

[Pigoski 97] T. M. Pigoski, *"Practical Software Maintenance"*, Wiley Computer Publishing, 1997, table 3.1, p. 31.

[Pfister 98] G. F. Pfister, *"In Search of Clusters"*, Prentice Hall, 1998.

[Roberts & Johnson 96] D. Roberts, RE. Johnson, "Evolving frameworks: a pattern language for developing object oriented frameworks", *in Pattern Languages of Programming Design 3*, Addison-Wesley Publishing Co.:Reading MA; pp. 471-486, 1996.

[Sun Cluster] Sun, "Sun Cluster 2.1 System Administration Guide", *Sun Microsystems*, 1998.

[TrueCluster] DEC, TrueCluster 2.1 System Administration Guide, Digital Equipment Corporation 1998, *http://www.unix.digital.com/faqs/publications/cluster_doc/cluster_16.*

[Vogels et al.] W. Vogels, D. Dumitriu, A. Argawal, T. Chia and K. Guo, "Scalability of the Microsoft Cluster Service", *http://www.cs.cornell.edu/rdc/mscs/nt98.*

[Wilson et al. 95] R. Wilson, M. Johnstone, M. Neely and D. Boles, "Dynamic storage allocation: A survey and critical review", *in Prodeedings of the 1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995 (Springer Verlag)
A slightly updated version of this paper can be found at: ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps

[Zorn] B. Zorn, "Malloc and GC implementations", *http://www.cs.colorado.edu/~zorn/Malloc.html, (site visited August 14th, 2001).*

Paper Section

Paper I

Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor

Daniel Häggander and Lars Lundberg
27th International Conference on Parallel Processing
Minneapolis, USA
August 1998

I

Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor

Daniel Häggander
Ericsson Software Technology AB
S-371 23 Karlskrona, Sweden
Daniel.Haggander@epk.ericsson.se

Lars Lundberg
University of Karlskrona/Ronneby
S-372 25 Ronneby, Sweden
Lars.Lundberg@ide.hk-r.se

Abstract

The Billing Gateway (BGw) is a large multithreaded object oriented C++ application running on Sun Solaris. Due to frequent allocation and deallocation of dynamic memory, the initial implementation of this system suffered from poor performance when executed on a multiprocessor.
In this paper we compare two approaches for improving the performance of BGw. First we replace the standard Solaris heap with a parallel heap. In the second approach we optimize the application code by removing a number of heap allocations/deallocations. In order to do this, we introduce memory pools for commonly used object types and replace some heap variables with stack variables.
The parallel heap approach resulted in a dramatic speedup improvement. The optimization of the application code did also result in a dramatic speedup improvement. For this approach the performance using a single-processor computer was also increased by a factor of eight. The optimizations took approximately one week to implement.

1. Introduction

Today, multiprocessors are used in a number of applications. Multithreaded programing makes it possible to write parallel applications which benefit from the processing capacity of multiprocessors. However, some parallel applications suffer from large run-time overhead and serialization problems when using multiprocessors. The speedup for multithreaded applications is usually good when the first processors are added. When more processors are added, the speedup curve increases more slowly. In some cases the speedup can actually start to decrease when we add more processors.

One serialization problem is sequential dynamic memory management. Moreover, most dynamic memory handlers are implemented without any consideration of multiprocessing specific problems, e.g. false memory sharing [2].

Object-orientation helps the programmer to develop large and maintainable software. Unfortunately, this use of object-orientation often results in an more intensive use of dynamic memory, making the dynamic memory performance problem worse.

The standard implementation of dynamic memory is often rather inefficient. A number of more efficient implementations have been developed, e.g. SmartHeap [4] and Heap++ [5]. The most common strategy is to use different allocation algorithms depending on the size of the requested memory. QuickHeap is a typical example [7].

An optimized heap often results in better performance for none-parallel applications. However, a performance evaluation of a weather forecast model [3] showed that a heap implementation which was efficient on a single-processor caused performance problems on a multiprocessor.

For multiprocessor systems ptmalloc [6] can be efficient. ptmalloc is a heap implementation which can perform several allocations and deallocations in parallel. The ptmalloc implementation is a version of Doug Lea's malloc implementation that was adapted for multiple threads by Volfram Gloger, while trying to avoid lock contention as much as possible.

The Billing Gateway (BGw) is a system for collecting billing information about calls from mobile phones. The system is a commercial product and it has been developed by the Ericsson telecommunication company. BGw is written in C++ [11] (approximately 100,000 lines of code) using object-oriented design, and the parallel execution has been implemented using Solaris threads. The system architecture is parallel and we therefore expected good speedup when using a multiprocessor. However, the actual speedup of BGw was very disappointing.

By using the BGw as an example, we explain why dynamic memory management can cause poor speedup when using a multiprocessor, particularly for object oriented programs. We also describe and evaluate two methods which dramatically improved the speedup of the BGw using a Sun multiprocessor with eight processors. These two methods

are: using a parallel heap implementation and redesigning memory usage within the application code, respectively.

The rest of the report is structured in following way: Section 2 describes the BGw and its speedup problem. Section 3 describes the dynamic memory performance problem in detail. In section 4 a parallel heap implementation is investigated. Section 5 describes a redesign which reduces the number of heap allocations. Section 6 concludes the paper.

2. Billing Gateway (BGw)

2.1. Overview

BGw transfers, filters and translates raw billing information from Network Elements (NE), such as switching centers and voice mail centers, in the telecommunication network to billing systems and other Post Processing Systems (PPS). Customer bills are then issued from the billing systems (see figure 1). The raw billing information consists of Call Data Records (CDRs). A CDR contains information about a call (in certain rare cases information about a call may be split up into several CDRs, e.g. when the call is very long). Each CDR is 175-225 bytes long. The CDRs are continuously stored in files in the network elements. With certain time intervals or when the files have reached a certain size, these files are sent to the billing gateway.

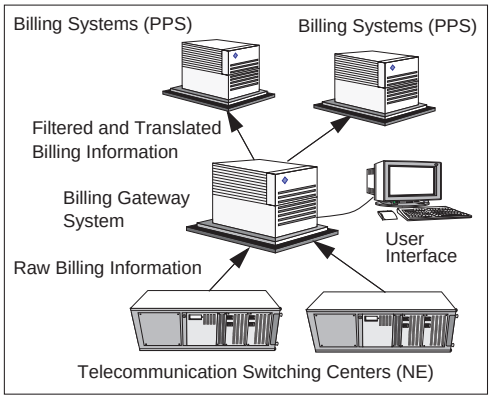


Figure 1. The Billing Gateway system.

There is a graphical user interface connected to the gateway system. In this interface the different streams of information going through the gateway are visualized as a directed graph, i.e. each billing application is represented as a graph. There are four major types of nodes in the application graphs.

Network element (NE) and post processing system (PPS) nodes represent external systems which communicate with the gateway, e.g. each switching center is represented as a NE node and each billing system is represented as a PPS node. There may be any number of NE and PPS nodes in an application.

The information streams in an application start in a NE node. An information stream always ends at a PPS node. Using a Filter node, it is possible to filter out some records in the information streams. In some cases the record format in the information streams has to be changed, e.g. when the post processing systems do not use the same record format as the network elements. Formatter nodes make it possible to perform such reformatting.

Figure 2 shows an application where there are two network elements producing billing information (the two leftmost nodes). These are called “MSC - New York” and “MSC - Boston” (MSC = Mobile Switching Center). The CDRs from these two MSCs are sent to a filter called “isBillable”. There is a function associated with each filter, and in this case the filter function evaluates to true for CDRs which contain proper information about billable services. CDRs which do not contain information about billable services are simply filtered out. The other CDRs are sent to another filter called “isRoaming”. In this case, there are two stream going out from the filter.

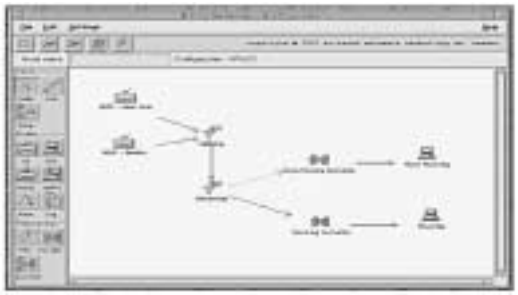


Figure 2. BGw configuration window.

The function associated with “isRoaming” evaluates to true if the CDR contains information about a roaming call (a roaming call occurs when a customer is using a network operator other than his own, e.g when travelling in another country). In this case, the record is forwarded to a formatter, and then to a billing system for roaming calls. If the filter function evaluates to false, the record is sent to a formatter and billing system for non-roaming calls. The billing systems are represented as PPS nodes.

The record format used by the billing systems differs from the record format produced by the MSCs. This is why the CDRs coming out of the last filter have to be translated into the record format used by the billing system before they can be sent from the gateway system to the billing systems.

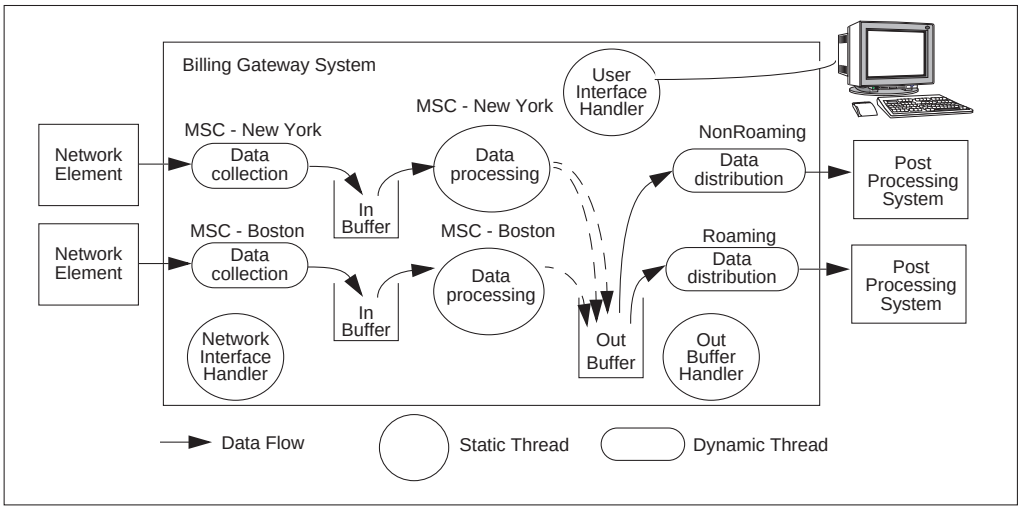


Figure 3. The thread structure of Billing Gateway.

The graph shown in figure 2 is only one example of how billing applications can be configured. The system can handle a large number of combinations of network elements, filters, formatters and post processing systems.

2.2. Implementation

Figure 3 shows the major threads for the application shown in figure 2. In order not to complicate the figure, some threads which monitor disk and memory usage etc. have been left out.

When there is no flow of data through the gateway, the system contains a number of static threads. When there is a flow of information going through the system, some additional threads are created dynamically.

When a network element wants to send a billing file to the gateway it starts by sending a connect message. This message is handled by the Network Interface Handler thread. This thread checks if the message is valid and determines the identity of the requesting network element. If the message is valid, a data collection thread is created. This thread reads the file from the network element and stores it on disk. When the complete file has been stored the data collection thread notifies the thread that shall process the data, and then the data collection thread terminates.

The data processing, i.e. the part of BGw that does the actual filtering and formatting, is implemented in a different way. When a configuration is activated, BGw creates one data processing thread for each NE node within the configuration (see figure 3). Every thread is bound to a certain NE, i.e a data processing thread can only process files which have been collected by the corresponding NE.

For the application in figure 2 each file of billing information from the network elements may generate a transmission of either zero, one or two files of billing information to the post processing systems. If all CDRs are filtered out as unbillable, no file is generated by the data processing thread. If all billable CDRs are either roaming or non-roaming, one file is generated for the billing system for roaming or non-roaming calls. If the MSC file contains billable CDRs for both roaming and non-roaming calls, one file is generated for each of the two billing systems. The files generated by the data processing threads are put into an outbuffer. When a file has been generated, the data processing thread notifies the Out Buffer Handler thread. The data processing thread then starts to process the next file in its In Buffer. If the In Buffer is empty, the thread waits until the next files has been collected.

The Out Buffer Handler is notified that a new file has been put into its buffer and a data distribution thread is created. This thread sends the file to the corresponding post processing system. The data distribution thread terminates when the file has been transmitted.

2.3. Performance

Billing Gateway versions 1 and 2 have been evaluated, using a Sun Sparc Center 2000 and a Sun Enterprise 4000 with eight processors [10]. These evaluations showed that a good speedup was achieved when the first processors were

added. When more processors were added, the speedup curve started to fall off. The speedup curves for BGw 1 and BGw 2 can be seen in figure 4. The configuration is the same as in figure 2 with the exception that there are eight network elements (MSCs) instead of two.

BGw 3 is the latest version of the Billing Gateway. This version has some new features. The most significant one is a new language which makes is possible to define more complex filters and formatters. The new language is of “C” style with sub-functions and local variables. The new language makes it easier to adapt BGw to new environments and configurations. However, the speedup of BGw 3 was extremely poor (see figure 4).

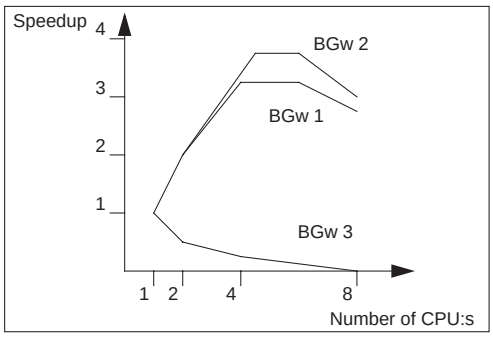


Figure 4. The speedup of the 3 BGw releases.

Previous investigations [3] have shown that dynamic memory can cause performance problems for multithreaded applications executing on multiprocessors. To find out if the dynamic memory caused the performance problem in BGw 3, an additional test was made.

One of the new features in BGw 3 which uses dynamic memory very frequently is the run-time interpreter of the filter- and formatter-language. The implementation of the language interpreter uses memory from the heap to store the local variables. The design of the application is made in such a way that at least one sub-function call was done when a CDR was processed by a filter or a formatter. This design led to a very intensive use of the heap, even for small configurations.

Another feature which uses dynamic memory frequently is the CDR decoder. More exactly, the decoding of CDRs containing dynamic structures of information. The implementation of the decoding algorithm is (more or less) the same in BGw 1, 2 and 3.

In order to test if dynamic memory management caused the speedup problems, the problematic CDRs were removed from the test files. Also, the filters and formatters used in the test was redefined in such a way that the number of sub-function calls and the use of local variables was minimized. The speedup of the adapted configuration and the new work-load was almost linear, i.e. almost optimal. Consequently, dynamic memory management caused most of the speedup problems in BGw 3.

In the next section we will discuss why dynamic memory management causes such large performance problems.

3. Dynamic memory management

Some features within the C++ language, such as dynamic binding [11] make it possible to implement a more maintainable software design. This type of design often use dynamic memory to become as general as possible. Reuse issues, such as class libraries and object oriented frameworks [9], advocate a general design and implementation, often resulting in very frequent allocation and deallocation of dynamic memory.

In C++, dynamic memory is allocated via the operator new. The operator returns a pointer to an allocated memory area The memory is then deallocated by the operator delete. Most compilers simply map these operators directly to the malloc() and free() functions of the standard C library. Consequently, C++ applications which have been developed to be general and reusable, use the malloc() and the free() functions very frequently. Figure 5 shows examples of memory management in C and C++ respectively.

In a multithreaded application, all threads use the same memory [8][13]. This means that all threads use the same heap for dynamic memory allocations. Therefore, the allocation- and deallocation-functions should be reentrant. Otherwise, they have to be protected against simultaneous usage. The C library functions malloc() and free() are usually not reentrant and must therefore be protected. In the Solaris implementation the heap is protected by a global mutex. The mutex is locked on entrance and unlock before returning, making sure that only one allocation or deallocation can be performed at the same time.

A major problem for a multithreaded application running on a multiprocessor, is that only one thread can allocate or deallocation memory at the same time. According to Amdahl’s Law, no application can run faster than the total execution time of its sequential parts. For applications where dynamic memory is allocated and deallocated frequently, the dynamic memory management significantly decreases the speedup.

However, even if an application spends all its time allocating and deallocating memory, i.e in malloc() and free(), the speedup should never be less than one, which was the case for BGw 3 (see figure 4). Consequently, there must be some additional explanation.

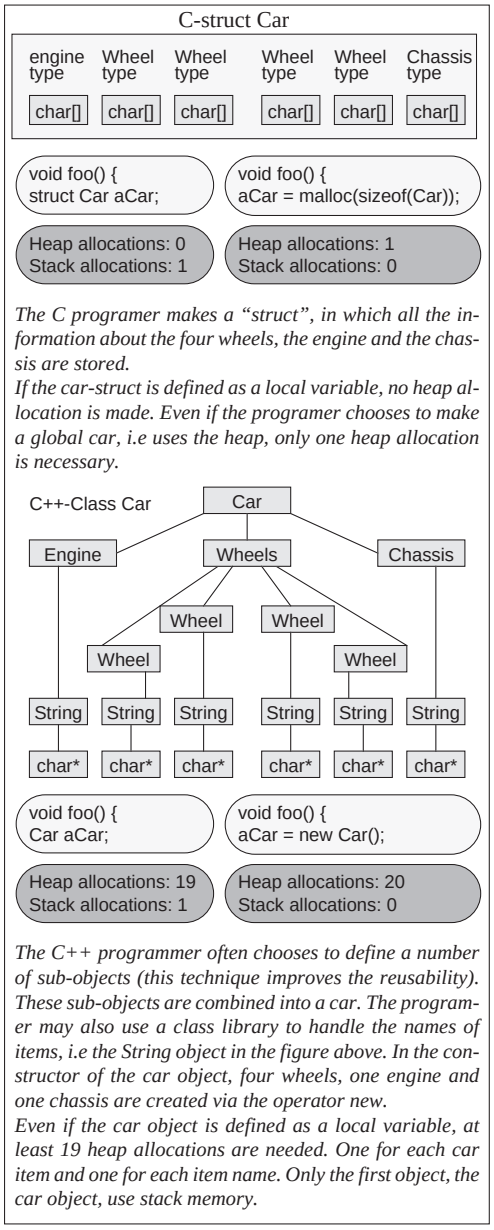


Figure 5. Data distribution in C and C++ respectively.

To lock and unlock a free mutex is fast in Solaris. On a single-processor, only one thread can execute at the same time. It is, therefore, rare that a thread tries to lock the heap mutex while another thread is holding it.

On a multiprocessor, several threads can execute at the same time. It is, therefore, more likely that two threads want to allocate or deallocate dynamic memory at the same time. In that case, one of the threads will fail to lock the mutex which protects the heap. In these cases, a queue of blocked threads is created. When the global mutex is unlocked, all the blocked threads in the queue try to complete their locks. It is common that an allocation is followed by a deallocation or a new allocation, i.e the thread which unlocks the mutex is often the one which succeeds in locking it again.

The many lock attempts generated produce large system overhead. The time in system mode increases and the time in user mode decreases when the number of failed locks increases. This can seen by using the Unix command time.

The system overhead produced when locking and unlocking the global mutex protecting heap, is one reason for speedup figures less than one (see figure 4). Another reason is false memory sharing.

Cache memories consist of a number of blocks (cache lines). The size of the cache line is system dependent. Sun computers have a cache line size on 16-64 bytes [1]. Sun multiprocessors have one cache for each CPU. It is possible that a number of threads have allocated variables which are stored in the same cache line. If these threads execute on different CPUs, false memory sharing will appear. When one thread changes its variable, the other copies of the cache line are invalidated. When the copies are invalidated, the other threads which are accessing data in the same cache line have to update the caches on their CPUs. The overhead costs of constantly updating the caches can be large.

The Solaris dynamic memory implementation has an alignment of 16 bytes and a minimum allocation size of 16 bytes. If the multiprocessor uses caches with a cache line size of 64 bytes (Enterprise 4000), up to 4 CPUs can share the same cache line (16 bytes x 4 is 64 bytes).

By overloading the malloc*()* function in libc (se figure 6), we were able to monitor the physical placement of the allocated memory. The malloc*()* function was overloaded with a new function which called the original malloc*()*, stored the address value returned from it and finally evaluated the possibility of false memory sharing. An allocation which shared a cache line with a memory area of another thread was identified as a false memory sharing candidate. Tests on Billing Gateway show that 50% of all dynamic memory allocations are such candidates.

4. A parallel heap implementation - ptmalloc

Two main problems with the standard malloc() and free() functions are that they can not be executed in parallel and the large overhead caused by the global mutex protecting the heap. These two problems would be solved if malloc() and free() were re-written in a way which made it possible for threads to allocate and deallocate memory in parallel.

It is rather simple to replace the standard heap implementation in existing applications. The heap can either be inserted when the application is linked or the heap library can be preloaded before the application is started, using dynamic libraries [12] (see figure 6).

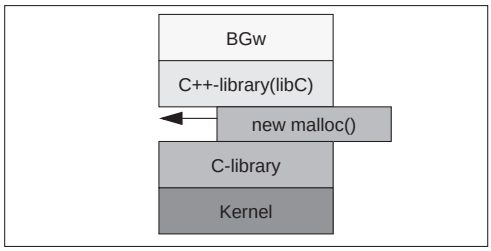


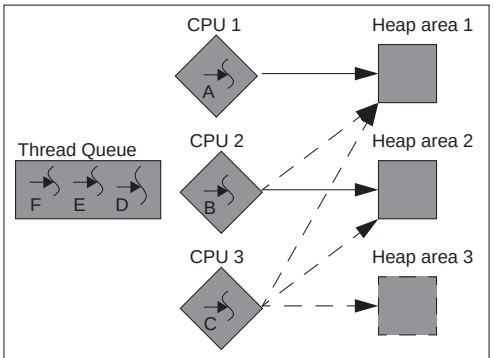
Figure 6. Overload the standard heap implementation.

A straight forward solution to implement a parallel heap is to have one heap area for each thread within an application. However, each heap area must be protected with a mutex even in this case. This is to make sure that no other thread currently uses it for deallocation. However, the mutex protection would probably not cause any major performance problems. Allocating memory from one thread and then deallocating it from another is not that common.

Many applications use a very large number of threads, and having a very large number of heap areas results in bad memory utilization. Most applications also create and delete threads dynamically. To create a new heap area each time a thread is created would generate large overhead for such applications.

A better idea is to have one heap area for each simultaneous allocation, i.e one heap area for each processor. The number of processors is low and static. However, to identify the processor currently used, is very costly. In Solaris, this operation requires a system call.

Most operating systems try to schedule a thread on the same processor as much as possible. Consequently, when a thread allocates dynamic memory, it is often executing on the same processor as it was when it made the previous allocation To use the same heap area as the last time can therefore be a good approximation.



An application has 6 threads and executes on a multiprocessor with 3 processors. On CPU 1, thread A is currently allocating memory from heap area 1.
When thread B, which is running on CPU 2, wants to allocate memory it first tests heap area 1. This heap is locked. Thread B then tests heap area 2. This heap is free and thread B can therefore lock and use this area.
Thread C is scheduled on CPU 3 and tries to allocate memory. Thread C tests heap areas 1 and 2, which are both locked. Thread C therefore creates a new heap (heap area 3), which is locked and used.

Figure 7. One heap for each simultaneous allocation.

If one heap area is occupied, e.g if a thread is scheduled up on a new processor, the thread will try to find a new heap area using trylock (se figure 7). With the trylock function a thread can test the status of a mutex without blocking the thread. A trylock locks the mutex if it is free, otherwise it returns an error status. The heap areas can in this way be tested one by one until a free area is found. The new area found will be used for further allocations, i.e. the thread will start the next allocation by looking at this heap area.

If no free heap area is found, a new heap area is created (see figure 7). In this way the number of heap areas will be dynamically adjusted to the number of processors.

The implementation discussed so far is basically the implementation of ptmalloc. In order to test how often a thread has to choose a new heap area, the source code of ptmalloc was modified. An alarm was raised every time a thread changed heap area. Tests made on BGw showed that the number of times a thread had to change heap area was very limited.

ptmalloc has the same interface as the C library functions and provides the same functionality. ptmalloc can therefore

replace the standard implementation without changing the source code of the application.

ptmalloc also reduces the risk of having threads which share the same cache block. The reason for this is that a certain heap area is often shared by a small number of threads and a thread can often use the same heap area for long periods of time. Consequently, ptmalloc reduces the risk of false sharing. However, the problem with false memory sharing is still not completely solved, e.g ptmalloc has a static memory alignment of 8 byte.

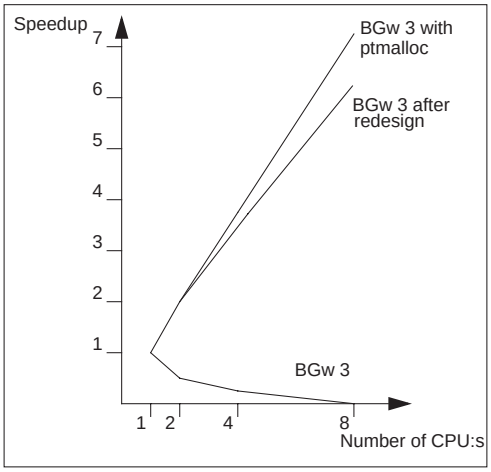


Figure 8. The speed-up of BGw using ptmalloc.

5. Redesign of BGw

There are, at least, three different aspects of dynamic memory management which affects performance negatively. First, the sequential nature of most implementations causes a serialization bottleneck. This problem was successfully removed by using a parallel heap implementation like ptmalloc. The second problem is false memory sharing due to the fact that variables from different threads, executing on different processors, often share the same cache line. This problem was reduced by using ptmalloc. The third problem is that it is relatively costly to allocate and deallocate memory from the heap. This problem affects not only the speedup, but also the performance on a single-processor computer. Replacing the standard heap implementation with ptmalloc did not affect the performance on a single-processor computer.

In order to improve also the single-processor performance we introduced memory pools for commonly used object types. Instead of deallocating an object, the object is returned to the pool and when a new object of the same type is needed it can be reclaimed from the pool. These pools are managed by the application code, thus reducing the number of heap allocations and deallocations. We also replaced a number of heap variables with variables on the stack. After doing these optimizations, the speedup was more or less the same when using the standard heap implementation compared to ptmalloc, i.e. the heap was no longer a serialization bottleneck.

The simple stack functionality offers faster allocations and deallocation of memory compared to the more complex heap. The stack memory is also thread specific, i.e each thread has its own stack. The allocations and deallocations can therefore be performed in parallel. Another benefit of having different memory areas for each thread is that the space locality is better, thus reducing false memory sharing.

Applications often need temporary memory to perform an operation. A bad habit is to use the operator new, even if the size is constant.

```
readAndPrint(int fd) {
unsigned char* buff =  new unsigned char[16];
read(fd, buff,  16);
printf("%s",buff);
delete buff;}
```

An alternative implementation is to use memory from the stack.

```
readAndPrint(int fd) {
char buff [16];
read(fd, buff,  16);
printf("%s",buff);}
```

The typical situation were dynamic memory is used are when the size of the allocated memory has to be decided in run-time. In this cases the C library function, alloca() can be used instead of the operator new. The function allocates a dynamic number of bytes from the stack.

The designers used approximately one week to replace a number of heap allocations with stack allocations and implement memory pools for commonly used objects (BGw 3 consists of more than 100,000 lines of C++ code). The redesign improved the speedup significantly (see figure 9). Figure 9 also shows that the design changes did not only improve the speedup. They also improve the performance of the application when it executes on a single-processor with a factor of eight. Better single-processor performance is one of the benefits of redesigning the application code compared to using a parallel heap implementation.

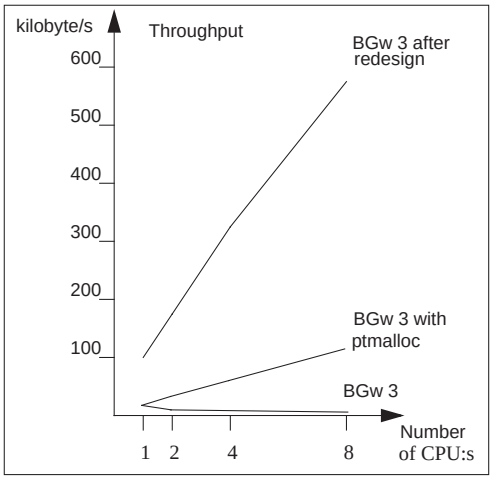


Figure 9. The new throughput of BGw.

6. Conclusions

Object oriented design, reusability and maintainability aspects all encourage the use of dynamic memory. The Billing Gateway (BGw 3) was developed using object oriented design, and reusability and maintainability were important aspects.

This study shows that for BGw 3, and similar multithreaded object oriented applications, dynamic memory can cause severe performance problems when using a multiprocessor. In the first version of BGw 3, the sequential implementation of dynamic memory allocation/deallocation and false memory sharing resulted in a performance degradation when using more than one processor.

Using a parallel dynamic memory handler, such as ptmalloc, is the easiest solution to the performance problem. The speedup improvement when using ptmalloc was dramatic. In fact, the speedup was almost linear, i.e. almost optimal. Consequently, the performance problems in BGw 3 were caused entirely by dynamic memory management, i.e. there were no other overhead problems or serialization bottlenecks. We expect that most multiprocessor manufactures will offer implementations similar to ptmalloc in the future. However, ptmalloc is not a commercial product, and the use of non-commercial products is often restricted at professional software companies.

An alternative solution to the dynamic memory performance problem, is to modify the application code in such a way that the number of heap allocations/deallocations is reduced. We were able to reduce the number of heap allocations/deallocations by introducing memory pools for commonly used object types and by replacing some heap variables with stack variables. It turned out that this approach did not only result in almost linear speedup; the performance using a single-processor computer was also improved by a factor of eight (see figure 9). The redesign reduced the number of heap allocations/deallocations significantly, thus making it possible to use the standard heap implementation without suffering from any serialization problems. The time for redesigning BGw in this way was one week. Consequently, the performance improvement was extremely good considering the limited effort.

Object oriented techniques make it possible to build large and complex systems, which are reusable and maintainable. However, if not carefully designed, object oriented programs may run into serious performance problems when using a multiprocessor. The poor performance due to dynamic memory management is not isolated to BGw. A number of related products have similar problems.

References

[1] Adrian Cockcroft, "SUN Performance and Tuning", Prentice Hall, 1995.

[2] M. Dubois, J. Skeppstedt, and P. Stenström, "Essential Misses and Data Traffic in Coherence Protocols", J. Parallel and Distributed Computing 29(2):108-125, October 1995.

[3] R. Ford, D. Snelling and A. Dickinson, "Dynamic Memory Control in a Parallel Implementation of an Operational Weather Forecast Model, in Proceedings of the 7:th SIAM Conference on parallel processing for scientific computing, 1995.

[4] http://www.microquill.com

[5] http://www.rougewave.com

[6] http://www.cs.colorado.edu/~zorn/Malloc.html

[7] Nick Lethaby and Ken Black, “Memory Management Strategies for C++”, Embedded Systems Programming, San Francisco, June, 1993.

[8] B. Lewis, “Threads Primer”, Prentice Hall, 1996.

[9] T. Lewis, L. Rosenstein, W. Pree, A. Weinand, E. Gamma, P. Calder, G. Anderst, J. Vlissides, and K. Schmucker, "Object Oriented Application Frameworks", Manning Publication Co, 1995.

[10] L. Lundberg and D. Häggander, "Multiprocessor Performance Evaluation of Billing Gateway Systems for Telecommunication Applications”, in Proceedings of the ISCA 9th International Conference in Industry and Engineering, December, Orlando 1996.

[11] B. Stroustrup, “The C++ Programming Language”, Addison-Wesley, 1986.

[12] Microsystems, Inc., “Linker And Libraries”, 1994.

[13] Soft, “Solaris Multithreaded Programming Guide”, Prentice Hall, 1995.

Paper II

Memory Allocation Prevented Telecommunication Application to be Parallelized for Better Database Utilization

Daniel Häggander and Lars Lundberg
6th International Australasian Conference on
Parallel and Real-Time Systems
Melbourne, Australia
November 1999

II

Memory Allocation Prevented Server Application to be Parallelized for Better Database Utilization

Daniel Häggander and Lars Lundberg

Department of Software Engineering and Computer Science
University of Karlskrona/Ronneby,
S-372 25 Ronneby, Sweden
Daniel.Haggander@ipd.hk-r.se, Lars.Lundberg@ipd.hk-r.se

Abstract. The rapid growth in the telecommunication market increases the performance requirements on applications supporting telecommunication networks. Many of these applications are based on commercial Relation DataBase Management Systems (RDBMS), a component which tends to become a bottleneck. Therefore, it is important to fully use the capacity of the database. A database server can usually not be fully utilized using a single client, especially not if the database server is executed on a Symmetric MultiProcessor (SMP). Multithreading makes it possible for single process applications to have multiple database clients. However, it is not trivial to increase the performance using the multithreading technique, bottlenecks can easily limit the performance significantly. In this study we have evaluated a telecommunication application which has been multithreaded for a better utilization of its parallel RDBMS. Our results show that multithreading can be an effective way to increase the performance of single process applications. However, this requires dynamic memory management optimized for multiprocessors.

1 Introduction

Fraud is one of the cellular operator's biggest problems. In some cases, operators have lost nearly 40% of their revenues. According to the Cellular Telecommunication Industry Association (CTIA), the global loss of revenue due to fraud in 1996 exceeded one billion USD, excluding costs of anti-fraud investments [7].

The Fraud Control Center (FCC) is an application that identifies and stops fraud activities in cellular networks. The application is a commercial product and has been developed by the Ericsson telecommunication company. FCC is written in C++ [10] (approximately 10,000 lines of code), using object-oriented design and it is based on a third party parallel RDBMS [1]. Since, the RDBMS stands for an essential part of the application's functionality, it is reasonable to believe that its performance has a great impact on the total performance of the FCC.

In order to fully utilize the capacity of a database server, especially a parallel one, the workload has to consist of a sufficient number of parallel database requests. For most applications this is no problem since traditional database systems often have several independent database client processes generating requests (see left part of Figure 1). However, some applications only have a single database client. A reason

for this may be that the database client itself acts as a server for other systems. Shared resources make it preferable to build server applications within a single process. The result is a system with only one database client (see right part of Figure 1).

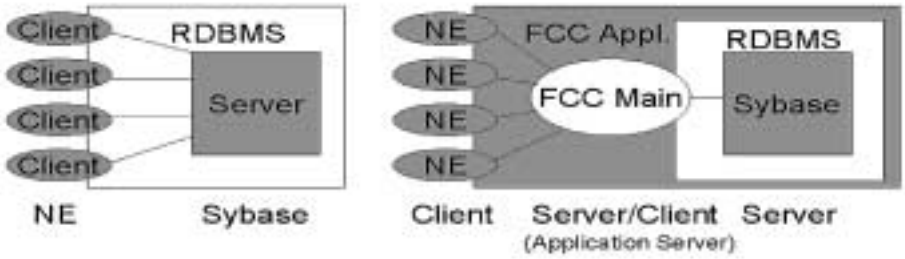


Fig. 1. Two RDBMS architectures.

In cases like this (see right part of Figure 1), multithreading can be an alternative to obtain the number of parallel requests needed. Multithreading permits multiple database clients within the same process. However, multithreaded programming on SMP:s is not trivial. Earlier studies have shown that congestion within dynamic memory can easily limit the performance significantly, especially if the application is written in C++ using object oriented design [2, 3, 6]. In this study we are going to quantify the performance gain of using multiple database clients and also evaluate the performance of FCC on an 8-way SMP.

The rest of the paper is structured in the following way. Section 2 describes the functionality and design of the FCC. In Section 3 the method used in this study is described, and the results are presented in Section 4. Section 5 and Section 6 discusses and concludes the paper, respectively.

2 Fraud Control Center (FCC)

Fig. 2. The FCC system.

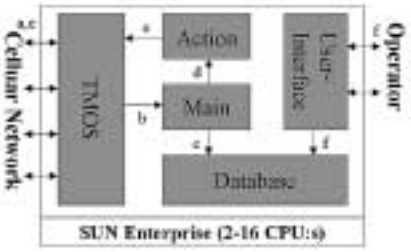


Fig. 3. The five modules of FCC.

2.1 System Overview

When operators introduce cellular telephony into an area, their primary concern is to establish capacity, coverage and signing up customers. However, as their network matures financial issues become more important, e.g. lost revenues due to fraud. The type of fraud varies from subscriber fraud to cloning fraud. Subscriber background and credit history check is the two main solutions to prevent subscriber fraud.

In cloning fraud the caller uses a false or stolen subscriber identity in order to make free calls or to be anonymous. FCC is a part of an anti-fraud system that combats cloning fraud with real-time analysis of network traffic [6].

Figure 2 shows an overview of the FCC system. Software in the switching network centers provides real-time surveillance of suspicious activities, e.g. irregular events associated with a call. The idea is to identify potential fraud calls and have them terminated. However, one single indication is not enough for call termination. FCC allows the cellular operator to decide certain criteria that have to be fulfilled before a call is terminated, e.g. the number of indications that has to be detected within a certain time period. An indication of fraud in the switching network is called an *event* in the rest of this paper.

The events are continuously stored in files in the cellular network elements (NE:s). With certain time intervals or when the files contain a certain number of events these files are sent to the FCC. The size of an event is about 250 bytes and maximum number of events in a file is 20-30. In FCC, the events are stored and matched against pre-defined rules. If an event triggers a rule, a message is sent to the switching network and the call is terminated.

FCC consists of five software modules, all executing on the same SMP (see Figure 3). The TMOS module is an Ericsson propriety platform that handles the interaction with the switching network (3a), i.e. collecting event files and sending messages about call terminations. The collected events are passed on to the Main module (3b) in FCC. In the Main module the event files are parsed and divided into separate events. The events are then stored and checked against the pre-defined rules using module three, the database (3c). If an event triggers a rule, the action module is notified (3d). This module is responsible for executing the action associated with a rule, e.g. a call termination via the TMOS module (3e). The last module is the graphical user interface of the FCC (3f). The rest of this paper will focus on the Main and Database modules.

2.2 Implementation

A commercial RDBMS (Sybase [8]) that allows SMP execution was used in the FCC. It also reduces the development and maintenance cost compared to using ordinary files. In most database applications, a database server is serving a large number of clients via a network. Therefore, the performance of most RDBMS has been heavily optimized for processing a large number of requests in parallel. Database servers running on SMP:s are usually divided into a number of processes. In Sybase these processes are called engines. In order to obtain maximum performance, each engine should handle a number of parallel requests. Having parallel database requests is therefore very important for database servers running on SMP:s.

If the database server is handling a large number of clients, there will automatically be a large number of parallel requests. However, in the telecommunication domain, we often have a situation where the database server only receives requests from one client. FCC is one such application. Although there are some additional clients performing maintenance and supervision tasks, the main part of the database requests are made by a large single module (the Main module), executing in a single UNIX process.

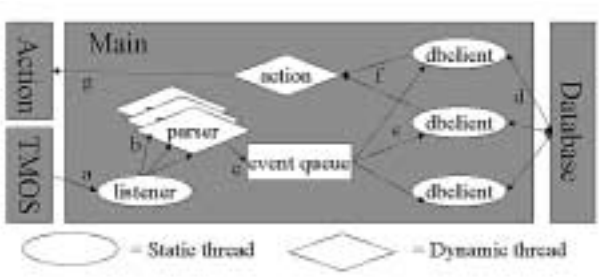


Fig. 4. The thread architecture of FCC

In order to improve performance, FCC has implemented parallel execution, using Solaris threads. The parallel execution makes it possible to obtain a behavior similar to a system with multiple clients, i.e. clients are modeled by threads. The parallel execution is also intended to scale-up other parts of the application which are not directly related to the RDBMS.

The processing within the Main module is based on threads. Figure 4 shows how the threads are interacting. The events are distributed to the Main module packed in files. A listener thread receives the event file (4a) and creates a parser thread (4b). After it has created the parser thread, the listener thread is ready to receive the next file. The parser threads extract the events from the file and insert the events into an event queue (4c), where they are waiting for further processing. When all events in a file have been extracted, each parser thread terminates. The number of simultaneous parser threads is dynamic. However, FCC makes it possible to define an upper limit.

The parser in FCC is designed using object-oriented techniques in a way that makes it highly adaptable. It is very important for FCC to quickly support new types of events since a new network release often introduces new event types or changes the format of old event types. However, the adaptable and object-oriented design results in frequent use of dynamic memory.

The Main module has a configurable number of connections towards the database module (4d), i.e. the database server. A dbclient thread handles each connection. A dbclient thread handles one event at a time by taking the first event from the event queue (4e) and then processing it. The interaction with the database is made with SQL commands [8] via a C-API provided by the database vendor. Each SQL command is constructed before it is sent to the database module. Since the final size of a SQL command is unknown at compile time, dynamic memory has to be used for its construction. The dbclient thread is also responsible for initiating resulting actions (4f, 4g) before it processes the next event in the event queue.

In order to achieve high performance a database expert from Sybase was consulted. The actions taken in order to improve the performance at that point were:

- A separate disk for the transaction log.
- Index tables on a separate disk for the six most accessed tables.
- Partition of the two most accessed tables.
- Small and static tables were bound to cache.

These optimizations are more or less standard (for more information about Sybase optimizations, see [9]). Database optimization techniques are not a subject for this study.

Two of FCC's five modules are more CPU consuming than the other. These are the Database and the Main modules. Neither of them is directly bound to any particular processor. However, the engines in Sybase are designed in a way that prevents an engine from being scheduled off a processor and an engine allocates a processor for some period of time even though there are no immediate request waiting to be processed. As a result, each engine more or less allocates a processor. The Main module mainly uses the remaining processors.

3 Method

The intention with this study was to quantify the gain of using multiple database clients and to evaluate the performance of FCC on an 8-way SMP. In order to do this, FCC was moved to an experiment environment together with a special designed workload simulator. Within this environment we evaluated the optimal number of clients for one database engine (processor). Using this result, we evaluated and compared two FCC versions on an 8-way SMP, a multi process and multithreaded version, respectively. The evaluation showed that the multithreaded version had a significantly lower throughput compared the multi process version.

The evaluation also showed that FCC had its maximum throughput using six database engines. We therefore explicitly quantified the optimal number of clients for six database engines, for the multi process version as well as for the multithreaded version. The measurements showed that the multithreaded version had a lower throughput for any number of threads. Detailed measurements on the multithreaded clients, i.e. measurements on a multithreaded client in which we had removed all database calls, showed that the low throughput was a result of a negative scale-up within the client. Consequently, the multithreaded client was a serious sequential bottleneck.

We had reasons to suspect that dynamic memory management caused the bottleneck, since earlier evaluations have identified dynamic memory as a potential bottleneck for applications written in object oriented C++ [10]. An additional FCC version, using a dynamic memory management optimized with ptmalloc [11], was developed, evaluated and compared with the results from the previous evaluation.

3.1 The experiment environment

The hardware used for the evaluations was a Sun Enterprise 4000 server with 8 processors. The server had two disk packages attached via separate disk controllers. One disk package was serving the operating system (Solaris), and the other package was used directly by the Sybase database. The package used by the database contained 12 disks. All disks were partitioned and configured in the same way as the real FCC disks. The database (Sybase 11.5) was install according to the FCC installation description.

The TMOS, Action and the User interface modules were manually removed before FCC was installed in the experimental environment.

FCC has an overload mechanism. When FCC no longer is capable of processing all incoming events, it starts dropping events and notifies the loss. The maximum capacity of FCC was measured by increasing the number of events sent to FCC until the overload protection was activated.

One of the intentions with the experiment was to leave the database installation and configuration intact. However, disk tracing during the initial phase of the performance evaluation indicated a heavily write activity on the disk handling the database transactions. This disk became a direct bottleneck in the system (see Figure 8). One simple solution to this kind of problem is to use a disk package with “fast write”, i.e. a write cache powered by battery for availability reason. However, disk packages of this type are very expansive. In our experiment we instead used the buffering within Solaris file system to simulate the characteristic of a “fast write” disk package. Apart from this, no changes within the installation or the configuration of the database were made.

It would be extremely difficult to use a real telecommunication network in the experiment. The solution was to develop an event generator that simulated a switching network. The simulator also made it possible to control the workload in detail. The simulator replaced the switching network and the TMOS module, i.e. the generator was connected directly to the Main module via the TCP/IP interface between the TMOS and the Main module (see Figure 5). The same simulator was used in the industrial Ericsson project.

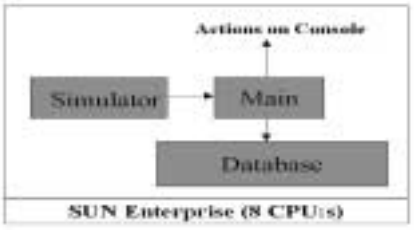


Fig. 5. The experiment environment.

$$F(t) = 1 - e^{-\lambda t}$$
$$t = \ln(x) / \lambda$$
$$x = \text{random } [0..1]$$

Fig 6. Function for workload generation.

It is common practice in the telecommunication community to use simulated workloads based on an exponential distribution function. The function used for event

generation can be seen in Figure 6. The construction of the simulator made it possible to control the number of events generated. Lambda (λ) equals the number of events generated per second in average and *t* equals the random time between two events.

Other parameters to the simulator were subscriber id range, maximum number of events in a file, number of switches simulated, the time between two file transfers and the number of different events types generated.

The following setting was used during the experiment:

- Subscriber identification: "0000000001-0000099999" (randomized)
- The maximum number of events in one file: 20
- The number of switches simulated: 5
- The file transfer interval in seconds: 5
- Number of different events types generated: 10

Ericsson considers these settings to be as a good representation of a medium size network.

In the test case when the Main module was divided into processes, each process was served by its own simulator instance. The total processing capacity was computed by summarizing the workload of the simulators. Finally, all events were tested against the three rules described below:

- Two or more indications from the same calling subscriber id in one minute.
- Two or more indications from the same calling subscriber id in one minute + a match of called subscriber.
- Thousand or more indications from the same subscriber id in one hour.

3.2 Evaluation on an 8-way SMP

The hypothesis is that an introduction of multiple clients will increase the database utilization, i.e. the performance of the application, compared to using a single client. For a single engine database, i.e. a database server that only benefits from the power of one processor, the potential increase in performance is limited. However, multiple clients still make it possible to effectively hide disk access time and latency within protocols used for the clients to server communication. The FCC throughput was measured when using up to 8 database clients. For results see Figure 7.

We had now quantified the optimal number of database clients per database engine (processor). The next step was to find the optimal number of database engines for the FCC when it is executed on an 8-way SMP. The maximum throughput was measured for both a multi process and a multithreaded FCC version. Measurements were made for one to 8 engines (see Figure 8).

The evaluation showed that the multithreaded version had significantly lower throughput than the multi process version. We thought that a possible reason for this could be that the multithreaded version has its peak performance for a different number of clients, i.e. the optimal number of clients may not be the same for the two versions. We therefore explicitly quantified the optimal number of clients when using

6 database engines. The same measurements were made for both versions (see Figure 8). The results showed that the multithreaded version had a lower throughput for any number of clients. In order to examine the reason, we measured the throughput of the multithreaded client only. We did this by removing all database calls made by the client. This showed that the multithreaded client had a negative scale-up when increasing the number of threads (database clients). Consequently, the multithreaded client was a serious sequential bottleneck.

3.3 Optimizing the dynamic memory management

We had now located the bottleneck to the multithreaded client. We also know that earlier experiments have indicated that object-oriented techniques dramatically increase the number of dynamic memory allocations for an application [3]. Object-oriented design primary concerns the maintainability and the reusability aspects of a product. In an object-oriented design, the application is modeled as a large number of objects. System functionality and data are then distributed over these objects. The general opinion in the software community is that small and well-specified objects result in a maintainable and reusable design [4]. More complex objects are obtained by combining a number of small objects. For example, a car can be represented as a number of wheel objects, a car-engine object and a chassis object. A car-engine-object uses a string object for its name representation and so on. Requirements on maintainability has forced the designers, not only to use many and small object, but also to build applications where objects are created at run-time, e.g. applications where the number of wheels on a car can be decided after the code has been complied. Since, each run-time creation of an object requires at least one dynamic memory allocation, the number of memory allocations, generated in an object-oriented application, can be enormous (for more information about object-orientation and dynamic memory management, see [3]).

Dynamic memory, in C++, is allocated using the operator new which is an encapsulation of the c-library function malloc(). Most implementations of malloc() do not support parallel entrance, which makes memory allocations very costly on a SMP. However, even more costly is the system overhead generated by the contention for entrance.

Our hypothesis was that contention in the dynamic memory management was a bottleneck in the multithreaded FCC database client (i.e. the Main module). This hypothesis was tested doing measurements on an optimized FCC versions. We used ptmalloc [11] that replaced the standard memory allocation routines. Ptmalloc is a malloc() implementation which can perform several allocations in parallel. The implementation is a version of Doug Lea's malloc implementation that was adapted for multiple threads by Wolfram Gloger, while trying to avoid lock contention as much as possible. Since ptmalloc has the same interface as the regular malloc(), this optimization does not require re-compilation of the source code. This FCC version was developed, evaluated and compared with the earlier results.

4 Results

The first part of this section presents the results of our measurements (see Figures 7, 8 and 9), while the second part shortly describes some performance related issues which Ericsson experienced during the real FCC project.

4.1 Throughput measurement

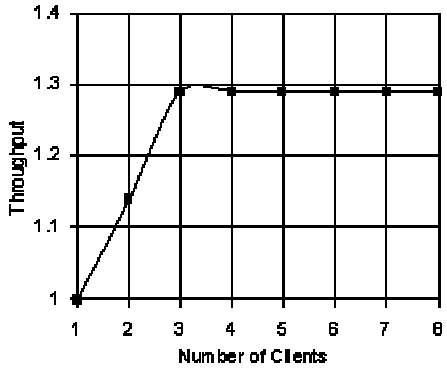


Fig. 7. Multiple clients using a single engine.

Figure 7 shows the increase in performance when using up to 8 database clients on a single processor database server. The maximum scale-up using a single engine database server was 1.3 and measured for 3 threads.

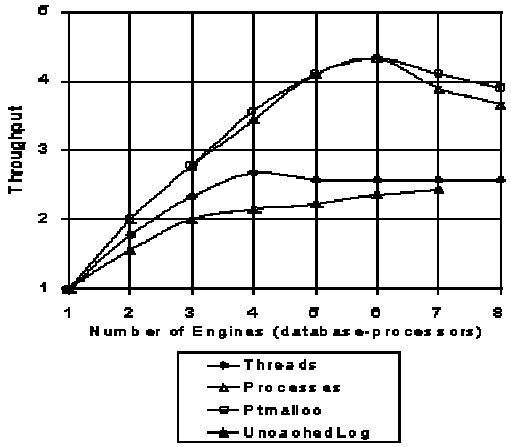


Fig. 8. FCC throughput on an 8-way SMP.

In Figure 8, the number of threads is three per engine and the figure shows the increase in performance when using up to 8 database server engines on an 8 processor SMP. The reason for having three threads per engine was that the thread scale-up diagram indicated three clients to be the optimal number (see Figure 7).
The very first version of FCC did not scale-up sufficiently. The reason for this was contention due to an un-cached transaction log. The rest of the FCC versions, the multi process, the multithreaded and the ptmalloc versions, all used a buffered transaction log (see Section 3.1).

The multi process version has its maximum scale-up (4.3) using six engines. The lower performance for 7 and 8 engines can be explained in the following way. If the FCC Main module should be able to generate requests enough to fully load the database server, two processors have to be used. The maximum performance was therefore achieved when six processors were used for the database server and two by the FCC Main module (see Section 2.3).

The multithreaded FCC version has a maximum scale-up of only 2.7 and this already for four engines. However, the same version with an optimized dynamic memory management, i.e. the ptmalloc version, shows approximately the same performance characteristic as the multi process version.

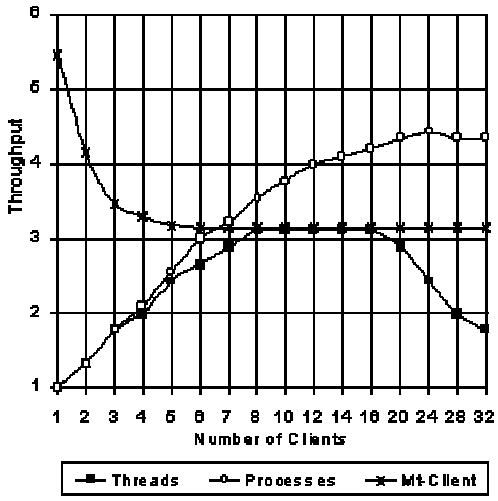


Fig. 9. Multiple clients using six engines.

Figure 9 shows how the multi process versions scales-up in a better manner than the multithreaded when we are increasing the number of clients. The multi process version reaches a maximum scale-up of 4.4 using 24 clients (processes), while the multithreaded version has the maximum scale-up of 3.1, already for 8 clients (threads). A quit modest scale-up considering a theoretical maximum of six or more. The third curve in Figure 9 shows the performance of the FCC Main module when the database server calls have been omitted. The curve shows how the bottleneck within the dynamic memory management, when using more than five threads, prevents the clients to generate a sufficient number of database requests. Consequently, the

maximum throughput of FCC is limited by the multithreaded client and the reason for this is congestion within the dynamic memory management.

When comparing the multithreaded result when using multiple clients on six engines (Figure 9) with the results for the total FCC throughput test (Figure 8), the first test comes up with a somewhat better peak performance. Our intention in the first test was to quantify the scale-up for multiple clients. Therefore, the parser functionality in FCC was made sequential, in order not to interfere. This change obviously had a positive effect on the total FCC performance. We believe the reason for this is that a sequential parser gives less dynamic memory contention. The fact that the difference in peak performance between the two tests is much smaller for the FCC versions with dynamic memory optimizations supports this theory.

4.2 Experiences from the FCC project

The project management had already from the beginning identified performance, in terms of throughput and response time, to be of great importance. The project group was assigned the task of building a prototype of the FCC system. The intention was to collect workload information, in order to specify the database capacity needed. The prototype indicated that the database had to utilize three engines in order to handle the expected workload. An additional result of the prototype was that at least five database clients had to be used for the database access. Otherwise, the database would not be able to use the SMP efficiently. Although there was a lot of attention to performance, the first version of the system had serious performance problems (see UncachedLog, in Figure 8).

The impact of introducing multithreaded programming is large. One of the main problems is that multithreaded and regular code can not be linked into the same executable since they use different library definitions. In the case of FCC, it resulted in a separation of the Main and the TMOS modules. The interaction with TMOS was normally made via an API. Access via inter-process communication was now necessary. The designers decided to use a TCP/IP based protocol. The interface towards the Action module also had to be modified, since the “fork()” function in multithreaded programs makes a complete copy of the parent process, including all existing threads [5]. Further on, a new set of “thread safe” libraries had to be ordered from Sybase, since the ones previously used were of an unsafe type.

Since most software developers were unfamiliar with concurrent programming, they also lacked experience of documenting concurrent designs. An additional problem, noticed late, was that it is really hard to test concurrent software. A problem that consumes “man hours” and decreases the quality of the product.

During the time FCC has been in live operation, two performance problems have been identified. The first concerns to the deletion of events and the other the generation of statistical reports.

FFC has a clean up functionality which each night deletes events of a certain age. The number of events to delete can be large. Sybase usually locks tables on a page basis. However, the database has a build in congestion protection. When a transaction holds more than a certain number of pagelocks, in FCC’s case 200, the whole table is locked. The nightly deletion of events triggered the congestion protection which resulted in that no more events could be insert into the database until the deletion was

competed, i.e. the FCC system stalled during the deletion phase. This operation could take a couple of hours to perform.

In the design phase, none or very little attention had been paid to the performance of generating statistical reports, e.g. the database design was completely focused an efficient event flow. This led to poor performance for the report generation. Both of these problems were relatively easy to solve. However, the process of identifying, implementing and verifying the solution for an operating product always becomes expensive.

5 Discussion

One important conclusion from the industrial Ericsson FCC project is that the present and future performance requirements of this kind of telecommunication systems can only be meet with multiprocessor systems. At least this is the conclusion made by Ericsson.

The experience from the industrial project showed that the functional requirements of FCC and similar applications require some kind of database. In order to reduce the cost for development and maintenance, a commercial database management system (often a RDBMS) will, in most cases, be used when developing the database. For the same reasons, the designers use object-orient design techniques. Consequently, a large and growing number of telecommunication applications use RDBMS, object-orientation and SMP:s.

The overall performance of FCC relies to a large extent on its capability to utilize the processing capacity of the RDBMS. In order to obtain high performance, it is very important for the server to process a number of database requests in parallel. Performance evaluations showed that there should be 3-4 clients for each database server engine (processor). A database server with six engines can obtain 4.4 times the performance of a one-engine system, if a sufficient number of simultaneous requests are generated (see Figure 9). Consequently, the performance of the commercial RDBMS scales up in a reasonable way on an SMP.

One way of obtaining parallel requests is to implement each database client as one Unix process, and then create a sufficient number of such processes. This is, however, not possible in all telecommunication applications. The reason for this is that the process issues the requests to the database sever, i.e. the database client process, is acting as a server to the systems (switching centers etc.) in the telecommunication network. In order to provide the server functions to the network elements, the database client often has to be implemented as one process. Consequently, many telecommunication applications are single client systems.

In order to obtain high performance in such database applications, the designers may use multithreading. However, experience from other multithreaded object-oriented applications shows that dynamic memory management tends to be a performance bottleneck [3]. This was also the case for FCC. There are two reasons why FCC has an intensive use of the dynamic memory. The object oriented design and the dynamic construction of database requests, respectively. By optimizing dynamic memory management the speedup using 8 processors was increased from 2.7 to 4.4, now bound by the database server. However, before the optimizations of dynamic memory the bottleneck was in the database client, i.e. the Main module in

FCC. Consequently, optimizations of the database sever has no effect unless we have removed the performance bottleneck caused by dynamic memory, e.g. by using a parallel memory handler such as ptmalloc.

We expect that the problem with dynamic memory management will escalate when the number of processors, and thus the number of threads increase. Moreover, the urge to increase the reusability and maintainability of large telecommunication systems will promote object-oriented design increasing the use of dynamic memory. Consequently, future systems will need much more efficient implementations of dynamic memory than the ones provided by the computer and operating system vendors today.

6 Conclusion

A large and growing number of telecommunication applications use RDBMS, object-orientation and SMP:s, the FCC is one example. The performance of a commercial RDBMS scales up in a reasonable way on a SMP (4.3 times on an 8-way SMP), if the workload consists of a sufficient number of parallel requests. Parallel requests can be generated using multiple clients. Performance evaluations using a SMP with 8 processors showed that there should be at least 3-4 clients for each database server engine (processor). However, in some applications there is only one database client process.

Multithreading can in many cases be used for obtaining the performance benefits of a multi client database application in applications with only one client process. However, this study shows that dynamic memory management tends to be a performance bottleneck, particularly when using object-oriented development techniques. Although the database module performed most of the work, the limiting factor in FCC was the multithreaded client (the Main module). Increasing the number of threads in the Main module reduces the performance of the module unless we use a dynamic memory handler optimized for SMP:s.

We successfully used ptmalloc for optimizing the dynamic memory management in FCC. The performance characteristic of the multithreaded client version that used ptmallloc was very similar to the version with multiple client processes. Consequently, multithreading is a useful technique for obtaining high and scalable performance in systems such as FCC, if and only if the dynamic memory handler is optimized for SMP:s.

References

1. T.Connolly, C. Begg, A. Strachan, "Database Systems", Addison-Wesely, 1996.
2. R. Ford, D. Snelling and A. Dickinson, "Dynamic Memory Control in a Parallel Implementation of an Operational Weather Forecast Model", in Proceedings of the 7:th SIAM Conference on parallel processing for scientific computing, 1995.
3. D.Häggander and L. Lundberg, "Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor", in Proceedings of the ICPP 98, 27th International Conference on Parallel Processing, August, Minneapolis 1998.

4. C. Larman, "Applying UML and Patterns", Prentice Hall, 1998.

5. B. Lewis, "Threads Primer", Prentice Hall, 1996.

6. L. Lundberg and D. Häggander, "Multiprocessor Performance Evaluation of BillingGateway Systems for Telecommunication Applications", in Proceedings of the ISCA 9th International Conference in Industry and Engineering, December, Orlando 1996.

7. C. Lundin, B. Nguyen and B. Ewart, "Fraud management and prevention in Ericsson's AMPS/D-AMPS system", Ericsson Review No. 4, 1996.

8. J. Panttaja, M. Panttaja and J. Bowman, "The Sybase SQL Server -Survival Guide", John Wiley & Sons, 1996.

9. S. Roy and M. Sugiyama, "Sybase Performance Tuning", Prentice Hall, 1996.

10. B. Stroustrup, "The C++ Programming Language", Addison-Wesley, 1986.

11. W. Gloger, "Dynamic memory allocator implementations", http://www.cs.colorado.edu/~zorn/Malloc.html

Paper III

Maintainability Myth Causes Performance Problems in Parallel Applications

Daniel Häggander, PerOlof Bengtsson, Jan Bosch and Lars Lundberg

3rd International IASTED Conference on Software Engineering and Applications
Scottsdale, USA
October 1999

III

MAINTAINABILITY MYTH CAUSES PERFORMANCE PROBLEMS IN PARALLEL APPLICATIONS

DANIEL HÄGGANDER, PEROLOF BENGTSSON, JAN BOSCH, LARS LUNDBERG
Department of Software Engineering and Computer Science
University of Karlskrona/Ronneby
S-372 25 Ronneby, Sweden
[Daniel.Haggander | PerOlof.Bengtsson | Jan.Bosch | Lars.Lundberg] @ipd.hk-r.se

Abstract. *The challenge in software design is to find solutions that balance and optimize the quality attributes of the system. In this paper we present a case study of a system and the results of a design decision made on weak assumptions. The system has been assessed with respect to performance characteristics and maintainability and we present and evaluate an alternative design of a critical system component. Based on interviews with the involved designers we establish the design rationale. We analyzed the evaluation data of the two alternatives and the design rationale and conclude that the design decision is based on a general assumption, that an adaptable component design will increase the maintainability of the system. This case study is clearly a counter example to that assumption and we reject it as a myth. We also show how this myth is responsible for the performance problem in the case system.*

Keywords. Performance, Maintainability, SMP, Object oriented design, Frameworks, Multithreading.

1 INTRODUCTION

Software quality requirements are getting more and more attention with the dawn of the software architecture discipline. It is not always possible to maximize each quality attribute in a design, and in that case one has to do trade-offs. Maintainability and performance often serve as examples of inherently conflicting quality requirements where a trade-off is inevitable [13]. In our experience, this is the general opinion in research as well as in the software development industry.

We have assessed a parallel commercial telecommunication system from Ericsson, a fraud control centre (FCC), for performance and maintainability independently to see if there was a conflict. The system is designed using object-oriented techniques and the parallel execution is implemented using threads [10]. The system seemed very suitable since the two driving requirements during its development were performance, i.e. throughput & scalability, and maintainability, i.e. adaptation of the design to new requirements. Our assessments show that the design of one component, a parser, was a key factor for performance as well as for maintainability.

Fine grained adaptable object designs (e.g. design patterns [4]) are generally considered to be more maintainable than designs based on rigid but exchangeable components. Interviews with the designers related to the project showed that this was also their opinion in the design of the FCC, particularly for the parser component. However, fine grained adaptable designs tend to be larger than designs based on rigid but exchangeable components. This was the main reason why maintainability assessments of an alternative design showed that the assumption above does not apply universally, and we therefore reject it as a myth.

Performance evaluations showed that the performance loss due to choosing the fine grained adaptable design instead of the design based on rigid but exchangeable components was very large, particularly when using multiprocessor platforms. Interviews with the developers show that these performance problems were not anticipated. Consequently, the design decision did not provide the anticipated leverage with respect to maintainability, and it caused an unexpected and serious performance problem. Thus, this case presents an excellent instance of a general misconception about maintainability and performance.

The rest of this paper is structured in the following way. The next section presents the FCC system. Section 3 describes our method, consisting of implementation and evaluation of an alternative design and interviews with the designers at Ericsson. In Section 4 we present the result from the evaluations of the alternative design and compare those results with the assessments of the original design. In Section 5 we discuss myths and reality regarding maintainability and performance. Section 6 concludes the paper.

2 FRAUD CONTROL CENTER (FCC)

2.1 CELLULAR FRAUD

When operators first introduce cellular telephony into an area, their primary concern is to establish capacity, coverage and signing up customers. However, as their network matures financial issues become more important, e.g. lost revenues due to fraud.

The type of fraud varies from subscriber fraud to cloning fraud. Subscriber background and credit history check are the two main solutions to prevent subscriber fraud. In cloning fraud the caller uses a false or stolen subscriber identification in order to make free calls or to be anonymous. There are a large number of different approaches to identify and stop cloning. FCC is a part of an anti-fraud system which combats cloning fraud with real-time analysis of network traffic [12].

299-091

2.2 SYSTEM OVERVIEW

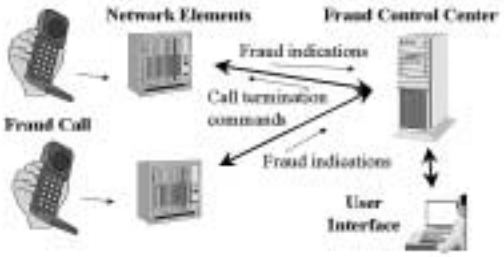


FIGURE 1. THE FCC SYSTEM.

Figure 1 shows an overview of the total FCC system. Software in the switching network centers provides real-time surveillance of suspicious activities associated with a call. The idea is to identify potential fraud calls and have them terminated. However, one single indication is not enough for call termination.

The FCC application allows the cellular operator to decide certain criteria that have to be fulfilled before a call is terminated. The criteria that can be defined are the number of indications that has to be detected within a certain period of time before any action is taken. It is possible to define special handling of certain subscribers and indication types. An indication of fraud in the switching network is called an *event* in the rest of this paper.

The events are continuously stored in files in the cellular network. With certain time intervals or when the files contain a certain number of events these files are sent to the FCC. In FCC, the events are stored and matched against pre-defined rules. If an event triggers a rule, a message is sent to the switching network and the call is terminated. It is possible to define an alternative action, e.g. to send a notification to an operator console.

The FCC application consists of five software modules, all executing on the same computer (see Figure 2). The TMOS module is an Ericsson propriety platform that handles the interaction with the switching network (2a), i.e. it is responsible for collecting event files and for sending messages about call terminations. The collected events are passed on to the next module (2b), which is the main module of FCC.

In the Main module the files of events are parsed and divided into separate events. The complete module, including the parser, is designed using object-oriented techniques and design patterns. The events are then stored and checked against the pre-defined rules using module three, the database (2c). If an event triggers a rule, the action module is notified (2d). This module is responsible for executing the action associated with a rule, e.g. a call termination. Standard Unix scripts are used to send terminating messages to the switching network via the TMOS module (2e). The last module is the graphical user interfaces from which the FCC application is controlled (2f).

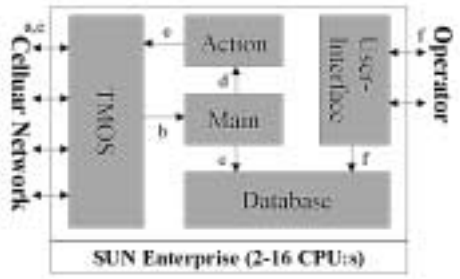


FIGURE 2. THE FIVE MODULES OF FCC.

2.3 PERFORMANCE ISSUES

The rapid growth in the telecommunication market increases the performance requirements on applications supporting telecommunication networks. In FCC, these performance requirements are meet by using a Symmetric Multiprocessor (SMP) [6], e.g. Sun multiprocessors. However, it is not trivial to increase the performance using SMP:s because bottlenecks can easily limit the performance significantly.

Performance evaluations showed that dynamic memory management becomes such a bottleneck in FCC [7]. The same evaluations showed that the parser was the limiting component. Dynamic memory management has in several evaluations of parallel applications been pointed out as a major bottleneck, especially for application implemented in C++ [3] [11] [8].

In C++ [15], dynamic memory is allocated using operator new which is an encapsulation of the c-library function malloc(). Allocated memory is deallocated using the c-library function free() via operator delete. Many implementations of malloc() and free() have very simple support for parallel entrance, e.g. using a mutex for the function code. Such implementations of dynamic memory result in a serialization bottleneck. Moreover, the system overhead generated by the contention for entrance can be substantial [8]. The bottleneck can be reduced using an optimized re-entrant implementation of malloc() and free(), e.g. ptmalloc [16]. However, even better performance can be achieved by decreasing the number of heap allocations [8]. A lower number of allocations will also improve the performance on uni-processors. Using a re-entrant implementation of malloc() and free() will only increase SMP performance.

In an object-oriented design, especially with design patterns, a large number of objects are used. An object often requires more then one dynamic memory allocations (call to operator new) in its construction. The reason for this is that each object often consists of a number of aggregated or nested objects. For example, a car can be represented as a number of wheel objects, a car-engine object and a chassis object. A car-engine-object may use a string object, usually from a third-party library, for its name representation and so on (see Figure 3).

It is not rare that these objects are combined at run-time in order to be as adaptable as possible. For example, a car

could have an unspecified number of wheels. Such a design requires that the each sub-object (Wheel) is created separately. As a result of this, dynamic memory are allocated and deallocated separately for each sub-object. The total number of allocations is, according to the discussion above, dependent on the composition of the object. Therefore every design decision which affects the composition of an object will also affect the number of memory allocations and deallocations during program execution.

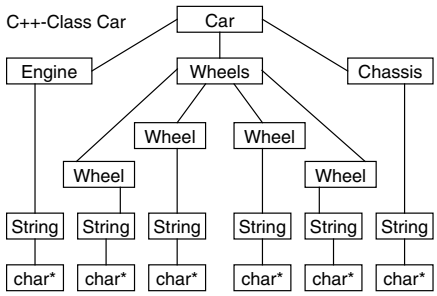


FIGURE 3. AN OBJECT REPRESENTATION OF CAR.

The parser component in FCC is similar to the car example above. The parser is constructed as a *State pattern* [4] where each possible state is represented by a class. A parser object holds one instance of every state class, in FCC's case nine. The nine instances are all created using operator new. All states also contain a third party software component; the "regular expression object", see Figure 4.

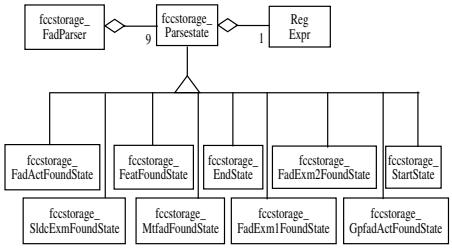


FIGURE 4. THE OBJECT DESIGN OF THE FCC PARSER.

Consequently, the performance of the FCC application is seriously limited by the large number of dynamic memory allocations and deallocations generated by the design of the parser.

3 METHODS

In Section 2.3 we described how congestion within the dynamic memory handling significantly limitates the performance of the FCC application. We also suggested that a fine grained adaptable object design can be the reason for this.

Interviews with the people responsible for the FCC parser design, showed that maintainability requirements were the main reason for the fine grained design of the parser component. The interviews also showed that the designers were quit pleased with the design result, and they did not anticipate any substantial performance problems.

With the knowledge that fine grained designs can cause performance problems, we performed an assessment addressing the maintainability aspects of the FCC application. Our intention was to find an alternative parser design which meets the maintainability requirements without causing performance problems. Based on the findings from this assessment, we suggested an alternative parser design which we implemented and evaluated with respect to performance and maintainability.

3.1 INTERVIEWS

We performed interviews with four persons acquainted to the FCC in order to determine the reasons for selecting the fine grained design of the parser. Each person was interviewed separately from the others. We selected four persons by the following criteria:

1. The designer responsible for the product.
2. One of the persons implementing the parser
3. A person in the project not responsible for the design or implementation of the parser.
4. A person outside the project but from the same product area within the company.

This selection enables us to find out; what was the intended design decisions from the responsible's point of view; does the person implementing the component have the same point of view, does another programmer in the project have the same understanding of the decision even when not directly involved and does this apply for other projects.

We performed the interviews using a questionnaire to help us remember all questions. The interview was divided into two parts. The first part was questions to be answered in a more discussive way. For example, one questions was "*Why did you choose to design the parser like you did?*". The second part contained very direct questions to be answered only yes or no, very rapidly. For example, "*Did you choose the parser design to reduce implementation time?*".

3.2 MAINTAINABILITY ASSESSMENT

We used a modified version of the scenario based software maintainability prediction method [2] on the software architecture of the FCC to assess the maintainability. In short the method consists of the following steps:

1. Identify categories of maintenance tasks
2. Synthesize scenarios
3. Assign each scenario a weight
4. Estimate the size of all elements
5. Impact analysis
6. Calculate the predicted maintenance effort

The only difference to the method as presented in [2] was that we did use a ordinal scale *{Not applicable, Low, Medium, High}* for estimating the amount of code that (1) needed to be modified and (2) the amount of new code that needed to be written.

A person closely related to the FCC project synthesized fourteen (14) scenarios to a scenario profile. Below is an example scenario from the profile.

"Event format changed to include several fraud indications"

Size estimations and impact analysis was combined into one activity and graded on the ordinal scale (above). A score was calculated by calculating a sum of the impact on each scenario in the scenario profile.

Besides the original design, a number of possible designs alternatives were evaluated using this scenario based maintainability assessment.

3.3 ALTERNATIVE PARSER DESIGN

The maintainability assessment of the FCC software architecture resulted in a proposal for an alternative design of the parser component which seemed promising, both from a maintainability and a performance point of view.

Instead of designing an object oriented adaptable parser, an alternative is to design a special purpose parser for each input format. When using this approach, instead of having one parser component the system would include one special purpose parser and determine what parser to run for the particular input that arrives. A parser is a well known component in the compiler community [1] and several tools for generating parser implementations from grammars in Backus-Naur format (BNF) exists, e.g. the standard Unix tools Lex and Yacc.

The FCC input format is suitable to describe as a context free grammar in BNF, i.e. a standard parser tool can be used. That would enable generation of several parsers for the time it takes to design, implement and test the flexible parser. Not mentioning the reduced complexity of maintaining the system. A new format would require some few changes to the syntax specification, a few lines of code and the generation of a new parser.

A developer was given the task to implement an alternative parser, using the architecture described above. The input to the developer was, besides the architecture strategy, the requirement specification from the original FCC project. Furthermore, the developer was instructed to implement the parser quick and simple, e.g. without considering the possibility that the source code has to be adapted to new requirements within the future.

The development of the experimental parser would incorporate two main activities, first learning to use the Lex and Yacc tools and then the parser was designed and implemented. The developer had no prior professional experience with Lex or Yacc before the assignment. The prototype development did not include design documentation and only limited verification of the parser was made. These two aspects have to be taken into account when comparing the time effort for the alternative parser with the original FCC parser. The new implementation did not used multithreading. Instead, ordinary UNIX processes was spawned of in order to be able utilize all processors on SMP:s hardware. This was a direct result of using the Lex and Yacc tools, which can not be mulithreaded.

3.4 PERFORMANCE COMPARISON

The source code of the original parser was extracted from the FCC application and then executed separately.

The performance of the two parser designs was evaluated on a SUN enterprise with eight processors, hosting a SUN Solaris operating system.

The throughput capacity was measured as the number of events parsed per second using one to eight processors. Since, the number of dynamic memory allocations has a large impact on the final FCC performance (see Section 2.3) the number of `malloc()` calls was counted for the two versions. The results of the measurements are presented in Section 4.

3.5 MAINTAINABILITY COMPARISON

We compare the two different designs based on the following research results:

- Software maintenance effort have been shown empirically to be strongly correlated to the code volume [9].
- Productivity for software development of new code is higher than the productivity of modification of existing code. Productivity being (>6x) higher when writing or generating new code (>250 LOC/Man-month in C [14]), compared to modifying legacy code (1,7 LOC/Man-day ~ <40 LOC/Man-month [5]).

We make the comparison by measuring the code volume for the two parser implementations, i.e. the original FCC parser and the alternative parser we implemented. Since the code volume is strongly correlated to the effort, we then calculate an effort prediction based on the productivity measures and the code volume measures.

If the code volume is significantly different in the two cases it is reasonable to assume that the comparison model will be valid. If the code volume measures of the two cases are very similar other factors might have to be involved in the comparison.

4 RESULTS

4.1 INTERVIEWS

These are the results from the interviews. We have concatenated the answers to the questions and in the cases where the interviewed persons have disagreeing answers, we present both.

These are the answers regarding the flexible parser:

1. *Why did you choose to design the parser like you did?*
Answer: We expected several input formats and late additions. It was a good way to introduce design patterns and it was a good object-oriented design.

2. *What alternatives were there?*
Lex & Yacc? Not familiar with the tools. Did not like to use generated code.
One hard-coded parser for every format? This was not adaptable enough. Some were personally against it.
A parser framework? Not considered.

3. *What consequences/advantages did you expect?*
Answer: More cost-effective, and less effort to implement support for new input formats even during the development project.

4. *What results have been achieved?*
Answer: The parser is fairly easy to adapt to new input formats.

5. *How would you solve the same problem today, based on your current experiences?*
Answer: Basically the same way, using the state-pattern, but the input formats have changed even more than anticipated. Therefore, more flexibility and adaptability would be good.

The results from the direct questions in the interviews are presented in Table 1. P1 through P4 is the persons interviewed, where P1 is the main designer for FCC, P2 is the parser programmer, P3 is another project programmer, and P4 is a designer from another project within the same product domain.

Question	P1	P2	P3	P4
Did you choose the parser design to reduce implementation time?	N	N	N	N
Did you choose the parser design to enable easy adaptation of the FCC to new or changed requirements?	Y	Y	Y	Y
Did you choose the parser design to improve the performance of the FCC?	N	N	N	N
Did you choose the parser design to minimize error sources?	Y	N	N	N
Did you anticipate any performance bottlenecks at all from these designs?	N	N	N	N

TABLE 1. The results from the interviews.

The result from the interviews show that the fine grained design was selected for maintainability reasons, and that no performance problems were anticipated. The answer to question 5 shows that the designers had themselves not been able to connect the performance problems with parser design, i.e. the connection is obviously non-trivial.

4.2 MEASUREMENTS ON THE ORIGINAL PARSER

Parser	Arch.	LOC	No of CPUs	Max events /sec.	No of malloc calls
FCC	multi-thread	3889	1	1800	100*ev. + 84
			8	680	
Alt.	UNIX -proc.	433	1	6800	11*ev. + 19
			8	55600	

TABLE 2. Experiment results.

The original implementation of the FCC parser has 3889 LOC (C++) and makes 100 calls to malloc() for each event file parsed. The maximum throughput is, using one processor 1800 events per second and using eight processors 680 (see Figure 5 and table 2). This gives a speed-up less than one, i.e. we get a slowdown. We have observed this kind of behavior before in previous projects, and the reason for the poor speed-up is congestions within the dynamic memory handling [8]. According to Ericsson the time effort to develop the original parser was 350 hours.

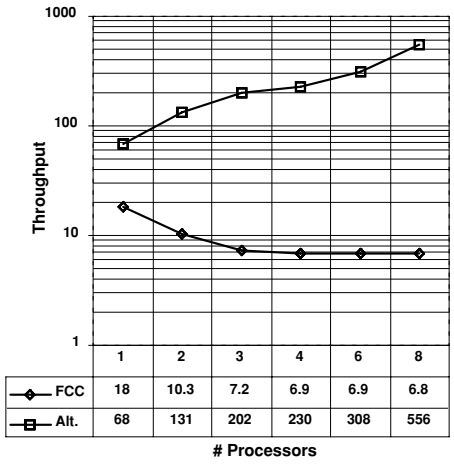


FIGURE 5. THE THROUGHPUT FOR THE ORIGINAL AND THE ALTERNATIVE PARSER.

4.3 MEASUREMENTS ON THE ALTERNATIVE PARSER

The experimental implementation was developed using the standard Unix tools, LEX and Yacc. The number of LOC (Lex, Yacc and C++) was 433. The parser is based on a

nine (1+8) UNIX processes architecture. The number of malloc() calls is eleven for each file of events parsed. The maximum throughput is, using one processor 6800 events per second and when using eight processors 55600 events per second (see Figure 5), i.e. somewhat more than eight times speed-up. The work load generator interferes more when only a few processors are used, and results in this super-linear speed-up. However, this effect is marginal and the speed-up is almost linear.

The time effort to develop the alternative parser was approximately 2 days. A comparison with the original FCC parser can be seen in Table 2. Consequently, the alternative parser has much better performance, particularly when using multiprocessors. Moreover, the development time for the alternative parser was much shorter than for the original, and the code size was also much smaller for the alternative parser.

5 DISCUSSION

Today, object oriented design and design patterns have become best practise in industry. Many designers are trained in state-of-the-art object oriented design and very familiar with design patterns. The object oriented design research community conveys that flexibility and adaptability are best achieved by designing components using design patterns. It is being taught and appreciated as a generally applicable assumption that guide the design of modern industrial applications.

Associated with this common assumption, it is assumed and accepted that a flexible and adaptable solution will have lower performance than the rigid exchangeable component design, but one expects that the performance reduction will be limited or even marginal.

In the Ericsson FCC project, the parser was identified as a key component for maintainability as well as for performance. In their efforts to produce a design that enabled them to adapt the parser component to new requirements the designers introduced a major performance bottleneck in their system. Interviews with the designers clearly show that the parser design was chosen to increase the adaptability, i.e. the designers adopted the general assumption that making a component adaptable would increase the overall system maintainability. By focusing on a single component in the system they overlooked an alternative. By making the system adaptable to changes and designing the component to be exchangeable new requirements could be met by replacing the component with a new one. The use of design patterns resulted in a component that is clearly adaptive. However, it requires nine classes and ~4 kLOC.

The results from the measurements of an alternative parser design show that the source code for that parser was about a tenth of the size of the original parser implementation. In [5] it is shown that the maintainability of a software module is highly correlated to its size in lines of code. Since the alternative design of the parser component was implemented in significantly less lines of code than the adaptable parser, we believe the maintainability effort for our alternative to be significantly less. Measurements of the implementation time for the original parser (350 h) and the alternative parser (16 h) strongly support this.

Hence, the assumption that an adaptable component design will increase the maintainability of the system proved to be invalid in this case. Therefore, the general assumption that fine grained adaptable designs are more maintainable than designs based on rigid but exchangeable components does not apply in all cases, and we reject it as a myth.

Performance comparisons between the original and the alternative design show that the alternative design has a much higher performance, particularly when using a multiprocessor. The throughput using eight processors is more than 80 times higher for the alternative design when using a multiprocessor with eight processors, and we expect that the performance difference will be even higher when using more than eight processors. The interviews show that the designers agreed unanimously that they had not anticipated the performance bottlenecks in their original design. The designers had themselves not been able to connect the performance problems with the parser design, i.e. the connection is obviously non-trivial.

Performance measurements of the FCC system showed that the parser component was the limiting bottleneck in the original design [7]. The original parser design resulted in a large number of object creations needed for parsing, thus increasing the of use of dynamic memory significantly. Consequently, the general assumption above, which we rejected as a myth, affected the performance of the FCC in a negative way. Hence, the performance problems in FCC were caused by a myth.

In this study we have found no characteristics or other evidence that distinguish this project from the general conception of other industrial development projects.

6 CONCLUSIONS

It is a challenge to find solutions that balance and optimize the quality attributes of a software system. It is not always possible to maximize each attribute and one has to make trade-offs. In this paper we have addressed the importance of the design decisions made during software development and the consequences of basing those decisions on assumptions or even prejudice.

We present a case, based on a real telecommunication application, where the performance has been significantly limited because of the assumptions that an fine grained adaptable object design (e.g. design patterns [4]) would give the application higher maintainability.

An alternative design was implemented and evaluated. The evaluation results show that the alternative parser has much better performance characteristics as well as higher maintainability. These findings show that the design decision was be based on a general assumption that proved invalid, i.e. the performance problems in FCC were caused by a myth.

The contribution of this papers is that the current prac-

tise in object oriented design and design patterns is not always the best alternative for neither performance nor maintainability. A design based on rigid but exchangeable components can give better performance as well as higher maintainability.

Acknowledgments

We would like to express gratitude to the persons in the FCC project who participated in the interviews and to Ericsson Software Technology AB for giving us the opportunity to study their work.

7 REFERENSES

[1] A. Aho, R. Sethi, A. V. Aho, J. D. Ullman, Compilers, *"Principles, Techniques And Tools"*, Addison Wesley Longman Inc., 1985, ISBN: 0201100886

[2] P. Bengtsson, J. Bosch, "Architecture Level Prediction of Software Maintenance", *in Proceedings of 3rd European Conference on Maintenance and Reengineering*, Amsterdam 1999.

[3] R. Ford, D. Snelling and A. Dickinson, "Dynamic Memory Control in a Parallel Implementation of an Operational Weather Forecast Model", *in Proceedings of the 7:th SIAM Conference on parallel processing for scientific computing*, 1995.

[4] E. Gamma, R. Helm, R. Johnson, J. Vilssides, *"Design Patterns"*, Addison-Wesley, 1997.

[5] Henry, J. E., Cain, J. P., "A Quantitative Comparison of Perfective and Corrective Software Maintenance", *Journal of Software Maintenance: Research and Practice,* John Wiley & Sons, Vol 9, pp. 281-297, 1997

[6] K. Hwang and Z. Xu, *"Scalable Parallel Computing"*, McGraw-Hill, 1998.

[7] D. Häggander and L. Lundberg, "Memory Allocation Prevented Telecommunication Application to be Parallelized for Better Database Utilization", *in Proceedings of the 6:th Annual Australasian Conference on parallel and Real.Time System*, Melbourne, Nov 1999.

[8] D.Häggander and L. Lundberg, "Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor", *in Proceedings of the ICPP 98, 27th International Conference on Parallel Processing*, August, Minneapolis 1998.

[9] W. Li, S. Henry, "Object-Oriented Metrics that Predict Maintainability", Elsevier Publishiing, *New York, Journal of Systems and Software*, v23, n2 , November, 1993, pp. 111-122.

[10] B. Lewis, *"Threads Primer"*, Prentice Hall, 1996.

[11] L. Lundberg and D. Häggander, "Multiprocessor Performance Evaluation of Billing Gateway Systems for Telecommunication Applications", *in Proceedings of the ISCA 9th International Conference in Industry and Engineering*, December, Orlando 1996.

[12] Catharina Lundin, Binh Nguyen and Ben Ewart, *"Fraud management and prevention in Ericsson's AMPS/ D-AMPS system"*, Ericsson Review No. 4, 1996.

[13] J.A. McCall, "Quality Factors", *Software Engineering Encyclopedia*, Vol 2, J.J. Marciniak ed., Wiley, 1994, pp. 958 - 971

[14] K. D. Maxwell, L. Van Wassenhove, S. Dutta, "Software Development Productivity of European Space, Military, and Industrial Applications", *IEEE Transactions on Software Engineering*, Vol. 22, No. 10: OCTOBER 1996, pp. 706-718

[15] B. Stroustrup, *"The C++ Programming Language"*, Addison-Wesley, 1986.

[16] W. Gloger, "Dynamic memory allocator implementations in Linux system libraries", *"http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html"* (site visited Aug the 3:th, 1999).

Paper IV

A Simple Process for Migrating Server Applications to SMPs

Daniel Häggander and Lars Lundberg
Journal of System and Software
2001

IV

ELSEVIER

The Journal of Systems and Software 57 (2001) 31–43

The Journal of Systems and Software

www.elsevier.com/locate/jss

A simple process for migrating server applications to SMP:s

Daniel Häggander, Lars Lundberg *

Department of Software Engineering, University of Karlskrona/Ronneby, S-372 25 Ronneby, Sweden

Received 4 February 2000; accepted 24 July 2000

Abstract

A strong focus on quality attributes such as maintainability and flexibility has resulted in a number of new methodologies, e.g., object-oriented and component-based design, which can significantly limit the application performance. A major challenge is to find solutions that balance and optimize the quality attributes, e.g., symmetric multiprocessor (SMP) performance contra maintainability and flexibility. We have studied three large real-time telecommunication server applications developed by Ericsson. In all these applications maintainability is strongly prioritized. The applications are also very demanding with respect to performance due to real-time requirements on throughput and response time. SMP:s and multithreading are used in order to give these applications a high and scalable performance. Our main finding is that dynamic memory management is a major bottleneck in these types of applications. The bottleneck can, however, be removed using memory allocators optimized for SMP:s or by reducing the number of allocations. We found that the number of allocations can be significantly reduced by using alternative design strategies for maintainability and flexibility. Based on our experiences we have defined a simple guideline-based process with the aim of helping designers of server applications to establish a balance between SMP performance, maintainability and flexibility.

Keywords: Symmetric multiprocessor (SMP); Guidelines; Performance; Multithreading; Real-time applications; Quality attributes

1. Introduction

Performance is not usually the most prioritized quality attribute when developing large and complex applications. Other quality attributes such as maintainability and flexibility are commercially more important, since more effort is spent on maintaining existing applications than developing new ones (Pigoski, 1997). Although performance is considered less important, some applications still demand high performance, e.g., real-time applications. For such applications, improving maintainability and flexibility is useless unless the performance requirements are fulfilled.

A simple, i.e., very cost-effective, way to improve the performance of an application is to increase the processing capacity of the hardware. The processing capacity can, for example, be increased by using a new and faster processor. Improving the application performance by using more powerful hardware may also be considered full-proof as regards the future since the peak performance of computers has grown exponentially from 1945 to the present, and there is little to suggest that this will change. The computer architectures used to sustain growth are changing, however, from the sequential to the parallel (Foster, 1995). Symmetric multiprocessor (SMP) is the most common parallel architecture in commercial applications (Hwang and Xu, 1998). It is thus reasonable to assume that developers of performance-demanding applications will be tempted to use SMP:s since their applications must meet new and high demands on performance. To improve the application performance using multiprocessors is not as simple as migrating to a faster processor. The software architecture and/or design must usually be adapted, i.e., parallelized and optimized for the multiprocessor architecture.

Multithreading (Lewis, 1996) is a programming method that allows parallelism within an application. It is characterized by low impact on the source code in C++ (Stroustrup, 1986) applications since it is based on shared address space and C-library calls. Multithreading can thus be considered a simple and cost-effective way to produce application parallelism. Designing efficient applications for multiprocessors using multithreading has, however, proven to be a far from trivial task that can easily limit the application performance (Lundberg and

* Corresponding author. Tel.: +46-457-385-833; fax: +46-457-271-25.
E-mail addresses: daniel.haggander@ipd.hk-r.se (D. Häggander), lars.lundberg@ipd.hk-r.se (L. Lundberg).

0164-1212/01/$ - see front matter
PII: S0164-1212(00)00115-1

D. Häggander, L. Lundberg / The Journal of Systems and Software 57 (2001) 31–43

Häggander, 1996; Häggander and Lundberg, 1998, 1999; Häggander et al., 1999; Larsson and Krishan, 1998; Ford et al., 1995).

Designing applications for multiprocessors is nothing new in high-performance computing. Efficient strategies of how to design and implement multiprocessor applications are well known (Foster, 1995). The strong connection between multiprocessors and high-performance computing has, however, raised standards and strategies strongly focused on execution time and scalability. Other quality attributes such as maintainability and flexibility have been more or less ignored. At the same time, traditional development has taken the opposite direction. A strong focus on maintainability and flexibility has resulted in new design and implementation methodologies, e.g., object-oriented design and third party class libraries. It is hardly surprising that an over-ambitious usage or a misapplication of these methodologies, seriously damages application performance, particularly for multithreaded applications executing on multiprocessors. It is thus not always possible to maximize each quality attribute in a design, hence trade-offs are often necessary. A great challenge is thus to find solutions that balance and optimize the quality attributes (Lundberg et al., 1999).

We have studied three server applications developed by the Ericsson telecommunication company. These are referred to as billing gateway (BGw), fraud control centers (FCC) and data monitoring (DMO). In all three applications maintainability and flexibility are strongly prioritized. The applications are also very demanding with respect to performance due to real-time requirements on throughput and response time. Symmetric multiprocessors and multithreaded programming are used in order to give these applications a high and scalable performance. This paper presents the studies and our findings. It concludes with a guideline-based process, assembled with the aim of helping designers of server applications to establish, in a more efficient manner, a balance between SMP performance, maintainability and flexibility.

The remaining pages are organized as follows. Section 2 presents the BGw, FCC and DMO applications. Section 3 describes our method which consists of evaluations, and interviews with the designers at Ericsson. In Section 4, we summarize our main findings. Related work is presented in Section 5. In Section 6, we discuss our findings from Section 4 in a wider context. Section 7 presents our guidelines, and Section 8 concludes the paper.

2. Server applications

In this section, we describe three industrial applications. All three applications are examples of a server application. We consider the applications to be large, both in terms of code size (10.000 to 100.000 lines of code) and variety. A more detailed description of the applications can be found in Häggander and Lundberg (1998, 1999).

2.1. Billing gateway (BGw)

BGw collects billing information about calls from mobile phones. The BGw is written in C++ using object-oriented design, and the parallel execution has been implemented using Solaris threads. The BGw transfers, filters and translates raw billing information from network elements (NE), such as switching centers and voice mail centers, in the telecommunication network to billing systems and other post processing systems (PPS). Customer bills are then issued from the billing systems. The raw billing information consists of call data records (CDRs). The CDRs are continuously stored in files in the NE. These files are sent to the BGw at certain time intervals or when the files have reached a certain size.

There is a graphical user interface connected to the BGw. In this interface the different streams of information going through the gateway are visualized. Fig. 1 shows an application where there are two NE producing billing information (the two leftmost nodes). These are called 'MSC-New York' and 'MSC-Boston' (MSC = Mobile Switching Center). The CDRs from these two MSCs are sent to a filter called 'isBillable'. There is a function associated with each filter, and in this case the filter function evaluates to true for CDRs which contain proper information about billable services. CDRs which do not contain information about billable services are simply filtered out. The other CDRs are sent to another filter called 'isRoaming'. The function associated with 'isRoaming' evaluates to true if the CDR contains information about a roaming call (a roaming call occurs when a customer is using a network operator other than his own, e.g., when travelling in another country). In this case, the record is forwarded to a formatter, and then to a billing system for roaming calls; the record is otherwise sent to a formatter and a billing system for non-roaming calls. The record format used by the billing systems differs from the format produced by the MSCs. This is why the CDRs coming out of the last filter must be reformatted before they can be sent to the billing systems. The graph in Fig. 1 is one example of how billing applications can be configured.

Fig. 2 shows the major threads for the application in Fig. 1. When there is no flow of data through the BGw, the system contains a number of static threads. When there is a flow of information going through the system, additional threads are created dynamically. When an NE sends a billing file to the BGw, a data

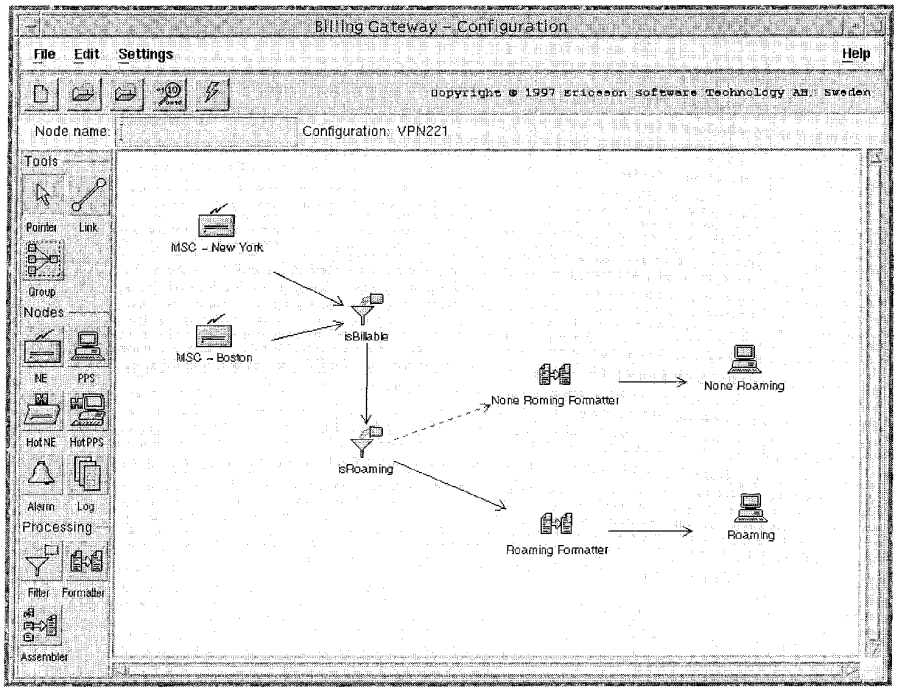


Fig. 1. BGw configuration window.

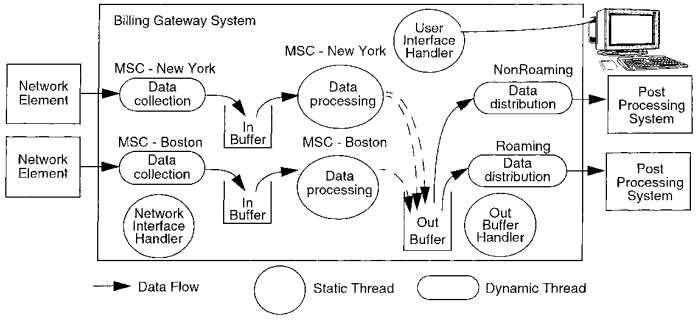


Fig. 2. The thread structure of BGw.

collection thread is created. This thread reads the file from the network and stores it on disk. When the complete file has been stored, the data collection thread terminates. Data processing, i.e., the part of the BGw that does the actual filtering and formatting, is implemented in a different way, i.e., there is one data processing thread for each NE node in the configuration (see Fig. 2).

The system architecture is parallel and we expected good multiprocessor speed-up. The speed-up was, however, very disappointing; in the first version the performance dropped when the number of processors increased because the dynamic memory management within the server process was a major bottleneck. The designers of the BGw wanted a flexible system where new CDR formats could be handled without changing

the system. One major component in the BGw is a very flexible parser that could handle data formats specified using ASN.1 (Larmouth, 1999). This parser uses a lot of dynamic memory. In order to increase the flexibility still further, a new language which makes it possible to define more complex filters and formatters was defined. The new language makes it easier to adapt the BGw to new environments and configurations. The introduction of the new language led, however, to a very intensive use of dynamic memory, even for small configurations. The excessive use of dynamic memory was a direct result from the effort of building a flexible and configurable application. It transpired that the performance problems caused by dynamic memory management could be removed relatively easily. By replacing the standard memory management routines in Solaris with a multiprocessor implementation called ptmalloc (Gloger, 2000), the performance was improved significantly. By spending one or two weeks on re-designing the performance was improved by approximately a factor of 8 for the sequential case, and a factor of more than 100 when using a multiprocessor. The major part of the redesign was aimed at introducing object pools for commonly used objects and to use stack memory when possible.

2.2. Fraud control centers (FCC)

When cellular operators first introduce cellular telephony in an area their primary concern is to establish capacity, coverage and signing up customers. However, as their network matures financial issues become more important, e.g., lost revenues due to fraud (Lundin et al., 1996).

Software in the switching centers provides real-time surveillance of irregular events associated with a call. The idea is to identify potential fraud calls, and have them terminated. One single indication is not enough for call termination. FCC allows the operator to decide certain criteria that must be fulfilled before a call is terminated, e.g., the number of indications that must be detected within a certain period of time. The events are continuously stored in files in the cellular network. These files are sent to the FCC at certain time intervals or when the files contains a certain number of events.

The FCC consists of four major software modules (see Fig. 3). The TMOS module is an Ericsson pro-

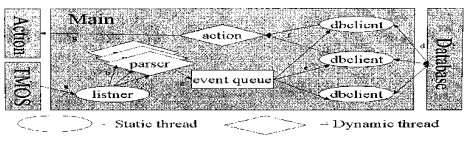


Fig. 3. The FCC application.

priety platform and handles the interaction with the switching network. The collected events are passed on to the Main module of the FCC. The event files are parsed and divided into separate events in the Main module. The events are then stored and checked against the pre-defined rules using the database. If an event triggers a rule, the action module is notified. This module is responsible for executing the action associated with a rule, e.g., to send terminating messages to the switching network.

A central part of FCC is data storage and data processing. A commercial RDBMS (Sybase) (Panttaja et al., 1996) was used in the implementation. In order to improve performance, FCC has implemented parallel execution using Solaris threads. The processing within the Main module is based on threads. Fig. 3 shows how the threads communicate. A listener thread receives the event file (Fig. 3(a)) and creates a parser thread (Fig. 3(b)). After it has created the parser thread, the listener thread is ready to receive the next file. The parser threads extract the events from the file and insert the separate events into an event queue (Fig. 3(c)), where they wait for further processing. When all events in a file have been extracted, the parser thread terminates. The number of simultaneous parser threads is dynamic. The parser in FCC is designed in such a way that it is flexible. It is very important for the FCC to support new types of events quickly when a new network release often introduces new ones or changes the format of old events. However, the flexible design results in frequent use of dynamic memory. The Main module has a configurable number of connections toward the database server (Fig. 3(d)). Each connection is handled by a dbclient thread. A dbclient thread handles one event at a time by taking the first event in the event queue (Fig. 3(e)), and processing it. The interaction with the database is made through SQL commands via a C-API provided by Sybase (Panttaja et al., 1996). Each SQL command is constructed before it is sent to the database module. Since the final size of a SQL command is unknown, dynamic memory is used for its construction. The dbclient thread is also responsible for initiating actions caused by the event (Fig. 3(f) and (g)) before it processes the next event.

One important conclusion from the industrial Ericsson FCC project is that the present and future performance requirements of this kind of telecommunication systems can only be met by multiprocessors. Dynamic memory management was also found to be a performance bottleneck for FCC. There are two reasons why FCC is an intensive user of dynamic memory: the object-oriented design of the parser, and the use of a string library for dynamic construction of database requests. By optimizing dynamic memory management the scaleup was increased significantly. We used two different approaches for optimizing the dynamic

memory handling in the FCC. One was to replace the standard memory handler with a parallel memory handler known as ptmalloc. The other approach was to split the client into two or three Unix processes (Unix processes have different memory images). The performance characteristics of these two approaches were very similar.

The impact of introducing multithreaded programming is considerable. One of the main problems is that multithreaded and regular code cannot be linked into the same executable since they use different library definitions. In the case of the FCC, this led to a separation of the Main and the TMOS modules. The interaction with TMOS was normally made via an API. Access via an inter-process communication became necessary. The designers decided to use a TCP/IP-based protocol. The interface towards the action module also had to be modified since the 'fork()' function in multithreaded program makes a complete copy of the parent process, including all existing threads (Lewis, 1996). It was subsequently necessary to order a new set of 'thread safe' libraries from Sybase since the ones previously used were 'none safe'. Since most software developers were unfamiliar with concurrent programing, they also lacked experience of documenting concurrent designs. An additional problem was that it is really difficult to test concurrent software, a problem which consumes work hours and decreases the quality of the product.

2.3. Data monitoring (DMO)

The growth in the mobile telecommunication market has been dramatic in the last years and there is little to suggest that this will change. Mobile operators constantly have to increase the capacity of their network in order to provide mobile services of good quality. Many network functions for operation and maintenance were originally designed to operate in networks much smaller than those of today. As the size of the network increases, these designs become more and more inadequate. One example is the management of performance data. Mobile networks today generate such large amounts of performance data that it can no longer be interpreted without help from computers.

DMO is an application that collects performance data and by using real-time analysis it detects, identifies and notifies performance problems in mobile networks (see Fig. 4). The performance data (counters) are continuously stored in files in the NE. These files are sent to the DMO at certain time intervals or when the files have reached a certain size. The received data is parsed into record objects in the DMO. Each object is then analyzed, i.e., matched against predefined rules. The DMO has two types of rules, static and/or historical thresholds. The historical thresholds are constantly updated for each data file received. When a performance problem is detected, the application sends a notification via an email or via a proprietary TCP/IP interface to the operator support center. The process is maintained and supervised via a graphical user interface.

Two student groups were assigned the task of developing two DMO applications. The requirements specification was assembled by the Ericsson telecommunication company. Each group consisted of four students working full-time for 10 weeks. Even though the two groups had identical requirements as regards functionality the requirements with respect to how to develop the applications differed. The first group (DMO 1) used third party tools liberally, while the other group (DMO 2) was not permitted to use third party tools, except for the operating system (Solaris) and for certain class libraries. Both applications were required, however, to use SMP in order to guarantee

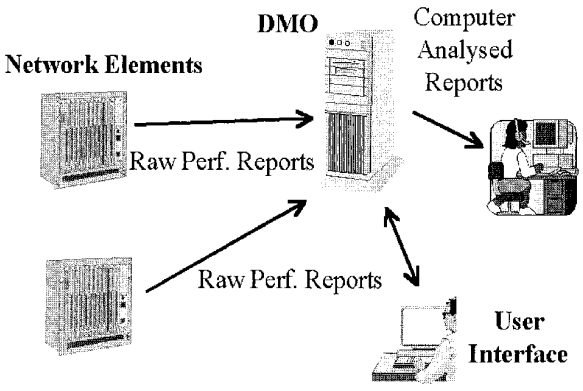


Fig. 4. The DMO application.

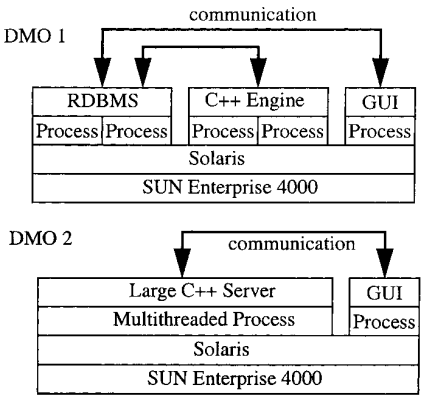


Fig. 5. Two architectures of DMO.

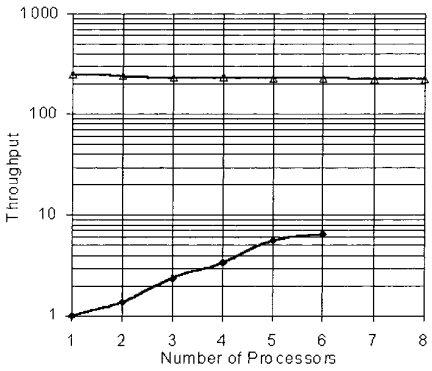


Fig. 6. DMO scale-up on an 8-way SMP.

high scalable application performance. The different requirements regarding how to use third party tools resulted in two different software architectures (see Fig. 5).

The DMO 1 architecture (the upper part of Fig. 5) is based on an RDBMS (Sybase) accessed via a small C++ server engine, while the DMO 2 architecture (the lower one in Fig. 5) is a large object-oriented and multithreaded C++ server using plain UNIX files for persistent storage. DMO 1 and DMO 2 are both maintained and supervised via a JAVA GUI, connected via TCP/IP.

DMO 2 is implemented using multithreaded programming and can thus be automatically distributed over the processors in an SMP via Solaris LWP:s (Lewis, 1996). The DMO 1 application uses a different technique for making use of multiple processors. In DMO 1, the parallelism is achieved by starting a multiple number of DMO 1 engines instances, each as separate UNIX processes. This makes it possible for DMO 1 to execute on multiple processors in parallel and to have simultaneous database connections. Multiple database connections usually give better RDBMS utilization, especially on an SMP (Häggander and Lundberg, 1999).

The performance of the DMO 1 was very poor. The main reason for this was the poor performance of the RDBMS. However, we expect that database optimizations, e.g., bulk copy and index tables, will increase the performance of the RDBMS dramatically. Even if the performance was poor, the application scaled-up rather well on an 8-way SMP. Fig. 6 shows that DMO 2 has significantly better throughput in comparison with the DMO 1. This application had, however, a most inefficient scale-up. The throughput decreased when using more than one processor. The bottleneck in the dynamic memory management was a major contributory factor. We believe that an optimized dynamic memory handling would make the application scale-up properly.

Both student groups had problems getting started. The reasons were, however, somewhat different. The DMO 1 group, the one with requirements on using third party tools, put a lot of effort into evaluating different tools. The DMO 2 group, the group not permitted to use third party tools, had a hard time choosing the 'right' architecture, a problem which followed the second group throughout the project. We imagine that the reason for this was simply that they had too many choices. A reflection made in the DMO 1 project also supports this hypothesis. It seemed as, the decision to use an RDBMS did not only define the database architecture. It also limited the number of possible architectures in other parts of the application, a limitation that seemed to guide the developers in designing the application.

While studying the DMO 2 project we found that the developers of DMO 2 could be divided into two categories: one category believed that multithreading simplifies both design and the implementation; the other believed multithreading makes design and implementation harder. How well developers adapt to parallel programming seems to be very individual.

3. Method

We have studied three large real-time telecommunication server applications (BGw, FCC and DMO), for a period of 5 yr. The applications have been studied in their industrial context as well as in an experimental environment, an 8-way Sun Enterprise multiprocessor system on which the three applications have been evaluated, optimized and then re-evaluated. The study also

includes a number of interviews with developers at Ericsson.

This report is a summary of our findings. We have identified negative and positive features common to the applications. These findings have been divided into five categories. Finally, we have assembled 10 design guidelines with the aim of helping designers of server applications to balance multiprocessor performance against other quality attributes such as maintainability and flexibility in a more efficient manner.

4. Lessons learned

We have divided our findings into the following five categories.

- Finding 1: when multiprocessors have been effective.
- Finding 2: when multiprocessors have caused problems.
- Finding 3: when multithreading has been effective.
- Finding 4: when multithreading has caused problems.
- Finding 5: actions which effectively increased the performance of multithreaded applications.

4.1. Multiprocessors have been effective...

Finding 1a: when applications consist of two or more independent sub-systems.

Comments: *Systems consisting of two or more sub-systems, which are normally time-sliced on a uni-processor or executed on two separate computers, usually scale-up well on a multiprocessor.*

Examples: *Server and GUI in the BGw, FCC and DMO.*

Finding 1b: when applications are based on a third party system that has shown itself to scale-up well on multiprocessors.

Comments: *Many third party systems, e.g., RDBMS have shown themselves to scale-up well on multiprocessors. It has also been shown that applications based on such systems scale-up efficiently on multiprocessors.*

Examples: *The database servers in the FCC and DMO* 1.

Finding 1c: when multiple instances of the same applications can be executed in parallel.

Comments: *In our experience, this is a simple way for a non-parallel application to utilize a multiprocessor. Executing multiple instances of the same application in parallel using separate processes has shown to be very effective. The method can be compared to Finding* 1a. *The 'sub-systems' have, however, became instanced from the same sequential application.*

Examples: *The servers in the process versions of the FCC and DMO* 1.

4.2. Multiprocessors have caused problems...

Finding 2a: for multithreaded applications.

Comments: *Even though a lot of effort was spent in preparing the applications for parallel execution, none of the multithreaded applications studied succeeded in scaling-up properly without modifications. Traditional development methods and skills are thus not enough to design an efficient multithreaded application for multiprocessors. Once the applications had been evaluated and optimized, however, they succeeded in scaling-up satisfactorily.*

Examples: *BGw server, FCC Main and DM0 2 server.*

Finding 2.b: because it differs from hardware traditionally used within this application domain.

Comments: *Multiprocessors (SMP:s) are a rather new concept for large real-time server applications. Third party tools, e.g., test tools and function libraries such as heap optimization libraries, are usually ineffective or even non-compliant with multiprocessors. We have, however, experienced a rapid improvement over the last* 2 *yr.*

Examples: *Third party class libraries and test-tools used in the BGw, FCC and DMO.*

4.3. Multithreading has been effective...

Finding 3a: for distributing applications over multiple processors.

Comments: *Our evaluations have shown that it is possible to achieve proper scalability using multithreading. This, however, requires extensive evaluations and optimization. Traditional development methods and skills are not enough to design an efficient multithreaded application for multiprocessors.*

Examples: *BGw with ptmalloc, BGw redesigned, FCC with ptmalloc.*

Finding 3b: for simplifying the design and implementation of parallel applications.

Comments: *Developers of multithreaded applications can be split into two categories: one who believes that multithreading simplifies both design and the implementation; the other thinks that multithreading makes design and implementation harder.*

Examples: *The designers of the BGw, FCC and DMO* 2.

4.4. Multithreading has caused problems...

Finding 4a: in applications which frequently allocate and de-allocate dynamic memory.

Comments: *In C and C++ dynamic memory is allocated and de-allocated via the C-library functions* `malloc()` *and* `free()`. *Many implementations of* `malloc()` *and* `free()` `have very simple support for parallel entrance, e.g.`, *they use a mutex for the function code. Such implementations of dynamic memory*

result in a serialization bottleneck in multithreaded applications. Moreover, the system overhead generated by the contention for entrance can be substantial on multiprocessors.

Examples: *BGw server*, *FCC Main and DMO* 2 *server.*

Finding 4b: in object-oriented applications.

Comments: *Applications design developed with object-oriented techniques* (*C++ applications*) *use dynamic memory more frequently than a structurally designed applications* (*C applications*).

Examples: *BGw server*, *FCC Main and DMO* 2 *server*.

Finding 4c: in applications using fine grained adaptable object design for maintainability or flexibility reasons.

Comments: *Current practice in object-oriented design and design patterns is not always the best alternative neither for performance nor maintainability. A fine grained adaptable object design can easily limit the performance significantly, especially for multithreaded applications executing on multiprocessors.*

Example: *FCC Main*

Finding 4d: because it differs from traditional software development.

Comments: *Multithreading is a rather new concept for this type of applications. Third party tools such as test tools, and function libraries, e.g., heap optimization libraries may be non-compliant or ineffective in multithreaded applications. We have experienced a rapidly improvement over the last* 2 *yr, however. An additional problem is that developers usually have limited education in writing parallel applications, especially multithreaded ones. Simple activities such as documentation and testing of parallel software thus make development more tricky compared to traditional development of sequential applications.*

Examples: *BGw server*, *FCC Main and DMO* 2 *server.*

4.5. Effective actions have been...

Finding 5a: to use rigid but exchangeable components instead of a fine grained adaptable objects.

Comments: *A design based on rigid but exchangeable components has been shown to give better performance as well as better maintainability* (Häggander et al., 1999).

Example: *FCC Main* (*architecture level*).

Finding 5b: to introduce specific object pools for complex and dynamic objects.

Comments: *Instead of de-allocating an object and its memory, it is returned to a pool of objects and when a new object of the same type is needed it can be reclaimed from this pool. The implementation is application specific and can be parallelized, i.e., object pools do not only reduce the number of allocations, they also improve the performance of the sequential execution.*

Example: *BGw server* (*object design level*).

Finding 5c: to change memory allocation from the heap to the stack.

Comments: *Applications often need temporary memory to perform an operation. A bad habit is to use the operator new, even if the size is constant* (*see the code examples*).

```
readAndPrint(int fd) {
unsigned char* buff = new unsigned char[16];
read(fd, buff, 16);
printf(''%s'', buff);
delete buff;}
```

An alternative implementation is to use memory from the stack.

```
readAndPrint(int fd) {
char buff [16];
read(fd, buff, 16);
printf(''%s'',buff);}
```

A typical situation where dynamic memory is used is when the size of the allocated memory has to be decided at run-time. In such cases, the C-library function, `alloca()` *can be used instead of the operator new. The function allocates a dynamic number of bytes from the stack.*

Example: *BGw server* (*implementation level*).

Finding 5d: to replace the standard memory management routines with new ones optimized for SMP:s

Comments: *By replacing the standard memory management routines with an SMP:s implementation, the performance can be improved significantly. Replacing the management routines is very simple since it does not require re-compilation, only re-linking of the application. Both freeware and commercial versions are available.*

Examples: *BGw server and FCC Main* (*operating system level*).

5. Related work

Performance trade-offs have always been made, and will continue to be made, in the design of software. Predictions and impact estimations are, however, often based on intuitive craftsmanship rather than rational engineering. When it comes to performance prediction, Smith (1990) has suggested a method known as the Software Performance Engineering method (SPE), in which she addresses the importance of having a process to ensure that all performance requirements are fulfilled. The approach is, however, very ambitious and will thus be used primarily on applications with high and very strict requirements on performance.

With the exception of ptmalloc and smartheap, we have not found many studies of parallel memory allocators. A relatively complete survey has, however, been written by Larsson and Krishan (1998).

Dynamic memory allocation is an important part of applications written in C++. High-performance algorithms for dynamic storage allocation have been, and will continue to be, of considerable interest. Zorn has addressed the subject in a number of papers, e.g., Detlefs et al. (1994) and Zorn and Grunwald (1994). These papers are, however, mostly focused on dynamic memory allocations in sequential applications. A handful of papers presenting new and adapted state-of-the-art solutions, e.g., patterns, have been written. Schmidt is one of the leading researchers in this area (Schmidt, 2000). The papers show that parallel software introduces the need for new patterns and idioms (Schmidt, 2001), but it also requires that existing, well-known and successful patterns are adapted or complemented in order to operate efficiently in the new context. One example is singleton, a very common pattern, which has proved to be an incomplete solution in parallel software since it only addresses the issue of creating objects (Schmidt, 1999). In order to work properly in parallel software, it also has to address the issue of object destruction.

6. Discussion

We have in Section 4 pointed out a number of occasions where we consider multiprocessors and multithreading to be an effective way of increasing the performance of server applications. We have not in any way, however, compared the effectiveness with other optimization possibilities such as optimization of the source code or cutting down on functionality. The BGw, e.g., had its sequential processing capacity increased by a factor of 8 by simple re-design. It will take an 8 way-SMP with perfect scale-up to match such an increase in performance. Are multiprocessors an effective way to increase the performance of the BGw? Furthermore, by starting six instances (processes) of a sequential FCC server, we reached the same processing capacity as that produced in our highly optimized multithreaded FCC version. Can we really claim that multithreading is an effective way of introducing parallelism for multiprocessor performance into the FCC? There is, apparently, no direct answer as to when to use or not to use multiprocessors or multithreading.

The usage of multiprocessors and multithreading creates a large number of new problems. Many of these problems can probably be avoided using more experienced developers. We also believe that problems with immature third party tools will decrease as multiprocessors become common in this type of applications. The design of highly efficient parallel applications for multiprocessors does, however, require a higher level of skill and a deeper knowledge of the hardware architecture as compared to designing sequential software for uni-processors. It is not reasonable to expect the average application developer to have such knowledge, and even if he or she has, it may still not be possible to find an efficient solution because of co-existing requirements on maintainability and flexibility.

A challenge is to find parallel design strategies which can offer scalable performance for software which is not primarily designed for multiprocessors or cannot be highly optimized for multiprocessors. We have ascertained that dynamic memory, i.e., memory allocation and de-allocation, is the most common bottleneck when executing these types of object-oriented applications on symmetric multiprocessors. The problem with dynamic memory can be dealt with in various ways and at different levels, e.g., using exchangeable instead of extendable components, object pools and parallelized heap implementations. As a result, both multiprocessors and multithreading are potentially important parts in server applications. Multiprocessors and especially multithreading are currently, however, two poorly developed technologies considering the scope for developing large real-time server applications. The lack of skills and in-depth knowledge of multiprocessors lead to large mismatches between the application software and the multiprocessor hardware; mismatches which limit the multiprocessor performance significantly. Choosing a parallel architecture which is simple and resilient to software/hardware mismatches can thus, in the long run, be more efficient than a more technically advanced architecture.

7. Guidelines

Consider a software company developing server applications. Over the years, the company has gradually and successfully adapted modern software engineering strategies, including object-oriented design (Booch et al., 1999), frameworks (Roberts and Johnson, 1996) and design patterns (Gamma et al., 1997), and has used a substantial amount of third party tools and software. Performance requirements are mainly met by using the latest in high-performance off-the-shelf uni-processor hardware. Imagine that new circumstances force the application designers to consider a change in hardware strategy, from a uni-processor to multiprocessor architecture, i.e., SMP.

The reasons for migrating to multiprocessor hardware may be higher requirements on performance or complementary requirements with respect to scalable performance. Irrespective of the reasons, the migration must be made in such a way that a high level of maintainability and flexibility is maintained, i.e., the applications must still utilize object-oriented design, third

party software, etc. The design of efficient applications for SMP:s is a vital task since communication and synchronization can easily limit the scale-up significantly. Efficient strategies for how to design and implement multiprocessor applications are well known from scientific and engineering supercomputer applications. In the type of application considered here the parallel performance must, however, be balanced with other quality attributes such as maintainability and flexibility.

For developers used to developing maintainable and flexible software, but unfamiliar with parallel hardware architectures and parallel software, making the right trade-offs for higher multiprocessor performance can be very difficult or even impossible. The goal of this study has thus been to define a set of design guidelines with the aim of helping designers of server applications to establish, in a more efficient manner, a balance between SMP performance, maintainability and flexibility.

The rest of this section presents 10 guidelines which are based on our findings. These guidelines are to be practised one by one, starting with number [I], until a sufficient SMP performance has been achieved (see Fig. 7). A more detailed flow chart, describing also the internal processes for each guideline, can be found together with the guideline definition. The guidelines can, furthermore, be divided into four groups (see Fig. 7). The first two guidelines are intended to help a designer to decide whether a supposed application design is scalable on an SMP without modifications, while guideline [III] encourages the designer to reconsider alternative optimizations once more. Guidelines [IV] and [V] are suggestions as to how to choose a parallel architecture. The last five guidelines concern how to design efficient multithreaded programs when maintainability and flexibility must also be taken into consideration. The ordering of the guidelines is assembled to support the main strategy: "improve the overall performance with minimum effort, i.e., minimum man-hours".

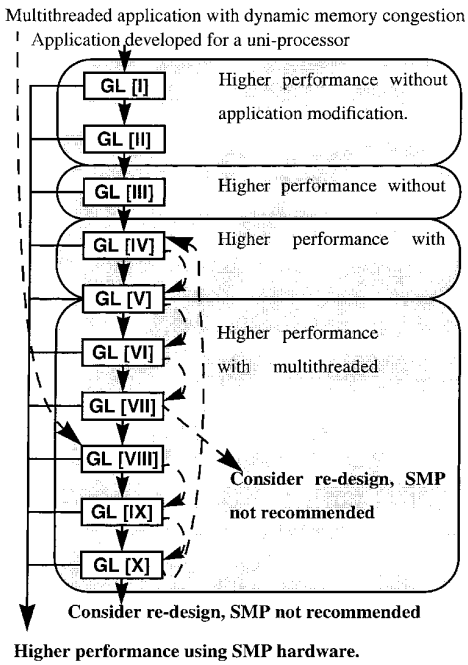


Fig. 7. The guideline-based process.

The process is primarily designed for existing applications which have been developed for a uni-processor system. The process can, however, be useful also when developing new applications. In these cases the development should start as if the application was to be developed for uni-processor hardware. However, on a very early stage, when the design still consists of models and diagrams, the first guidelines should be considered. Using the guidelines in such an early stage requires some minor adaptations. For example, the performance evaluations must be made using alternative evaluations techniques (i.e., estimation techniques), since an executable application does not yet exist. We believe that if the first five guidelines are applied in the early design, the last five guidelines will never have to be used.

An alternative scenario is that an application already has been parallelized using multithreaded programming (without these guidelines). It is likely that such an application suffers from dynamic memory congestion. The aim of the process is to improve the overall performance in a cost-effective way. Since the parallelism already is implemented, the first action is in this case to get as good performance as possible using the existing code and architecture, i.e., by temporarily removing the dynamic memory bottleneck. This implies that the optimal order in which the guidelines are to be practised differs. Instead of using the guidelines strictly from [I] to [X], we suggest an alternative order (shown with dashed arrows in Fig. 7).

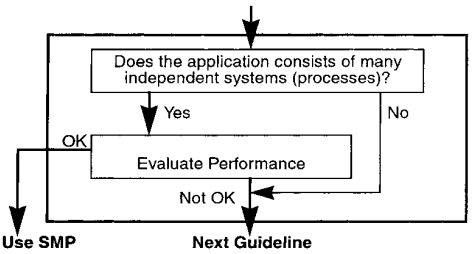
Using the guidelines in the order indicated by the dashed lines may fix a performance problem with limited effort. Experiences from the BGw, FCC and DMO projects show, however, that application specific optimizations (e.g., objects pools) are difficult to maintain and may therefore lose their effect after a number of product releases. When it comes to the long-term strategies we thus suggest that the original order, i.e., from [I] to [X], should be used also in these situations.

GL [I]: Consider a multiprocessor when the application design consists of many independent system instances.

Comments: *Our experience is that applications consisting of multiple system instances (processes), normally executing on separate computers or timed-sliced on a uniprocessor, scale-up well on multiprocessors. The system instances can be of different systems or copies of the same system. If the instances use shared resources, e.g., a database server, the synchronization must be supported by the resource side.*

Based on Findings: 1a and 1c.

Process flow:

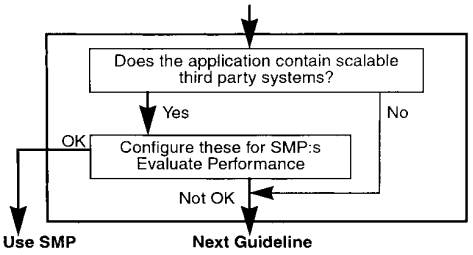


GL [II]: Consider a multiprocessor when the application design is based on third party systems which have shown themselves to scale-up well on multiprocessors.

Comments: *Our evaluations show that many third party systems scale-up efficiently on multiprocessors. Applications based on such systems also scale-up well on multiprocessors. This, however, usually requires special versions of the third party systems and, in most cases also reconfiguration of these systems.*

Based on Finding: 1b.

Process flow:

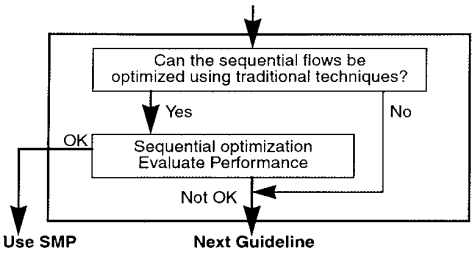


GL [III]: Reconsider using traditional performance optimization techniques instead of multiprocessors.

Comments: *We believe that the design of multiprocessor applications causes new types of problems. In some cases multiprocessors introduce more problems than they solve. Alternative solutions such as traditional code optimization should be considered first. If the multiprocessor domain is well known, however, and the potential gain in performance is essential, multiprocessor hardware is a competitive alternative for improving performance even in large real-time server applications.*

Based on Findings: 2a and 2b (see also Section 5).

Process flow:

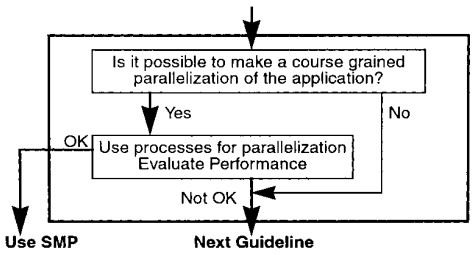


GL [IV]: Consider multiple processes for high performance and scalable first.

Comments: *We have found no loss in performance using multiple processes in comparison to multithreading programming. Rather, just the opposite. Misapplication of object-oriented design has led to a frequent usage of dynamic memory, thereby preventing the multithreaded application from scaling-up. In our experience, a parallel architecture based on multiple processors is more resilient to software/hardware mismatches. In some applications, however, e.g., application servers, multiple process solutions are not suitable.*

Based on Findings: 1a, 1c, 2a, 3a, 3b, 4a and 4d.

Process flow:

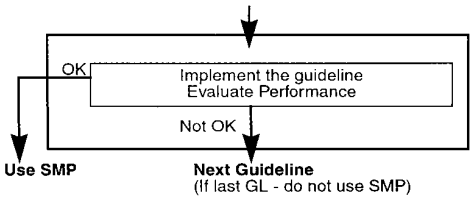


GL [V]: Consider multithreaded programing in combination with multiple processes.

Comments: *Multithreaded programing is a simple and cost-effective method for enabling application parallelism. This, together with aspects such as efficient scheduling and simplified communication, make multithreading a very competitive alternative to multiple processes. Designing multithreaded applications creates, however, a large number of problems, e.g., non-compliant function libraries and third party software. A multi-process architecture where only the most performance critical parts are multithreaded processes reduces the risk of having multithreading problems or software/hardware mismatches.*

Based on Findings: 1a, 1c, 2a, 3a, 3b, 4a and 4d.

Process flow:



GL [VI]: Avoid frequent allocations and de-allocations of dynamic memory.

Comments: *The bottleneck within the dynamic memory handler is a very common performance problem in multithreaded applications executing on multiprocessors, particularly applications designed with an object-oriented methodology. It is important not only to reduce the direct use of dynamic memory, but also to decrease the number of allocations by using less complex and course-grained object structures.*

Based on Finding: 4a.

Process flow: See process flow for GL [V].

GL [VII]: Meet maintainability requirements with exchangeable components instead of extendable components.

Comments: *Exchangeable components can be made less dynamic, i.e., they perform less dynamic memory allocations and de-allocations. Our evaluations show that applications using exchangeable, rather than extendable components are not necessarily less maintainable.*

Based on Findings: 4c and 5a.

Process flow: See process flow for GL [V].

GL [VIII]: Evaluate a heap implementation optimized for multiprocessors.

Comments: *We have found that heap implementations optimized for multiprocessors can be very effective. A heap implementation is also cheap and simple to install, i.e., one has everything to win and nothing to lose. Both freeware and commercial versions are available.*

Based on Finding: 5d.

Process flow: See process flow for GL [V].

GL [IX]: Utilize stack memory if possible.

Comments: *The applications studied have a tendency to use expensive heap memory where it would be possible to use stack memory. In multithreaded applications, the stack is thread specific and thus more efficient than the heap. Be very careful, however, not to exceed the stack size, especially when using dynamic stack allocation with* `alloca()`.

Based on Finding: 5c.

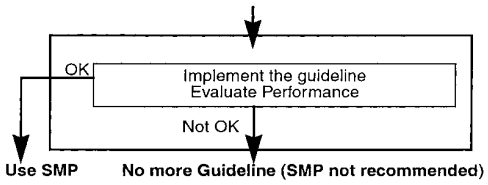
Process flow: See process flow for GL [V].

GL [X]: Introduce application specific dynamic memory management.

Comments: *While studying the three applications we found that the most effective way to increase the overall performance was to implement application specific dynamic memory management. Such implementations can be made more efficient and parallel, since they can be tailor-made for a certain application. The problem with this type of optimization is that it requires deep application knowledge, otherwise the implementation can be very time consuming or even impossible. We have also experienced that application specific optimizations tend to lose their efficiency when the applications grow.*

Based on Findings: 5b

Process flow:



8. Conclusion

It is not always possible to maximize each quality attribute in a design, thereby making trade-offs necessary. A major challenge in software design is thus to find solutions that balance and optimize the quality attributes. Server applications need to balance multiple quality attributes, i.e., maintainability, flexibility and high performance. Flexibility requirements with respect to application functionality must be balanced with highly adaptable object-orientation design strategies, and finally optimized for a parallel hardware architecture. This must be done in a cost-effective way, using third party software and off-the-shelf components to be competitive on the rapidly moving telecommunication market. For developers used to developing maintainable and flexible software but unfamiliar with parallel hardware and software, making the right trade-offs for higher multiprocessor performance can be very difficult or even impossible. Our investigations show, however, that the amount of information needed to find a significantly better solution is limited.

We have studied three telecommunication server applications. The applications have been studied in their industrial context as well as in an experimental environment. Our findings are transformed into a simple guideline-based process, which allows designers of server applications to balance multiprocessor performance against maintainability and flexibility.

References

Booch, G., Rumbaugh, J., Jacobson, I., 1999. The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA.

Detlefs, D., Dosser, A., Zorn, B., 1994. Memory allocation costs in large C and C++ programs. Software Practice and Experience 24 (6), 527–542.

Ford, R., Snelling, D., Dickinson, A., 1995. Dynamic memory control in a parallel implementation of an operational weather forecast model. In: Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing.

Foster, I., 1995. Designing and Building Parallel Programs. Addison-Wesley, Reading, MA.

Gamma, E., Helm, R., Johnson, R., Vilssides, J., 1997. Design Patterns. Addison-Wesley, Reading, MA.

Gloger, W., 2000. Dynamic memory allocator implementations in Linux system libraries. http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html (site visited 3 August).

Hwang, K., Xu, Z., 1998. Scalable and Parallel Computing. WCB/McGraw-Hill, New York.

Häggander, D., Lundberg, L., 1998. Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor. In: Proceedings of the ICPP 98, 27th International Conference on Parallel Processing, Minneapolis, August.

Häggander, D., Lundberg, L., 1999. Memory allocation prevented telecommunication application to be parallelized for better database utilization. In: Proceedings of the Sixth International Australasian Conference on Parallel and Real-Time Systems, Melbourne, Australia, November.

Häggander, D., Bengtsson, P., Bosch, J., Lundberg, L., 1999. Maintainability myth causes performance problems in parallel applications. In: Proceedings of SEA'99, the Third International Conference on Software Engineering and Application, Scottsdale, USA, October.

Häggander, D., Bengtsson, P., Bosch, J., Lundberg, L., 1999. Maintainability myth causes performance problems in SMP applications. In: Proceedings of APSEC'99, the Sixth IEEE Asian-Pacific Conference on Software Engineering, Takamatsu, Japan, December.

Panttaja, J., Panttaja, M., Bowman, J., 1996. The Sybase SQL Server – Survival Guide. Wiley, Chichester, UK.

Pigoski, T.M., 1997. Practical Software Maintenance, Wiley Computer Publishing, New York (p. 31, Table 3.1).

Larmouth, J., 1999. ASN. 1 Complete. Morgan Kaufmann, Los Altos, CA.

Larsson P., Krishan, M., 1998. Memory allocation for long-running server applications. In: Proceedings of the International Symposium on Memory Management, ISMM '98, Vancouver, BC, Canada, October.

Lewis, B., 1996. Threads Primer. Prentice-Hall, Englewood Cliffs, NJ.

Lundberg L., Häggander, D., 1996. Multiprocessor performance evaluation of billing gateway systems for telecommunication applications. In: Proceedings of the ISCA Ninth International Conference in Industry and Engineering, Orlando, December.

Lundberg, L., Bosch, J., Häggander, D., Bengtsson, P., 1999. Quality attributes in software achitecture design. In: Proceedings of SEA'99, the Third International Conference on Software Engineering and Application, Scottsdale, USA, October.

Lundin, C., Nguyen B., Ewart, B., 1996. Fraud management and prevention in Ericsson's AMPS/D-AMPS system. Ericsson Review No. 4.

Roberts, D., Johnson, R.E., 1996. Evolving frameworks: a pattern language for developing object oriented frameworks. In: Pattern Languages of Programming Design 3. Addison-Wesley, Reading, MA, pp. 471–486.

Schmidt, D., 1999. A complementary pattern for controlling object creation and destruction. In: Proceedings of the Sixth Pattern Languages of Programs, Monticello, IL, USA, August.

Schmidt, D., Patterns and idioms for simplifying multi-threaded C++ components. C++ Report Magazine (to appear).

Schmidt, D., 2000. "Douglas C. Schmidt". http://www.cs.wustl.edu/~schmidt (site visited 3 August).

Smith, C., 1990. Performance Engineering of Software systems. Addison-Wesley, Reading, MA.

Stroustrup, B., 1986. The C++ Programming Language. Addison-Wesley, Reading, MA.

Zorn, B., Grunwald, D., 1994. Evaluating models of memory allocation. ACM Transactions on Modeling and Computer Simulation 4 (1), 107–131.

Paper V

Quality Attribute Conflicts - Experiences from a Large Telecommunication Application

Daniel Häggander, Lars Lundberg and Jonas Matton
7th IEEE International Conference on Engineering
of Complex Computer Systems
Skövde, Sweden
June 2001

V

Quality Attribute Conflicts
- Experiences from a Large Telecommunication Application

Daniel Häggander, Lars Lundberg and Jonas Matton
Department of Software Engineering
Blekinge Institute of Technology
Box 520
S-372 25 Ronneby
Sweden
Daniel.Haggander@bth.se, Lars.Lundberg@bth.se and Jonas.Matton@epk.ericsson.se

Abstract

Modern telecommunication applications must provide high availability and performance. They must also be maintainable in order to reduce the maintenance cost and time-to-market for new versions. Previous studies have shown that the ambition to build maintainable systems may result in very poor performance. Here we evaluate an application called SDP pre-paid and show that the ambition to build systems with high performance and availability can lead to a complex software design with poor maintainability.

We show that more than 85% of the SDP code is due to performance and availability optimizations. By implementing a SDP prototype with an alternative architecture we show that the code size can be reduced with an order of magnitude by removing the performance and availability optimizations from the source code and instead using modern fault tolerant hardware and third party software. The performance and availability of the prototype is as least as good as the old SDP. The hardware and third party software cost is only 20-30% higher for the prototype. We also define three guidelines that help us to focus the additional hardware investments to the parts where it is really needed.

1. Introduction

A strong focus on quality attributes such as maintainability and flexibility has resulted in a number of new development methodologies, e.g., object-oriented and multi-layer design. Unfortunately, it has also resulted in less efficient applications [5][6], from a performance perspective that is.

One of the main reason for this is that applications developed using these new methodologies are more general and dynamic in their design. This is good for maintainability, but gives less application performance in comparison with static and tailored software. Consequently, the ambition of building maintainable systems often results in poor performance (some conflicts, but not all, are however based on myths and misconceptions and can thus be eliminated [7]).

Our first hypothesis is that the conflict is bidirectional, i.e., applications developed with high requirements on run-time qualities such as performance (and availability) may run into maintainability problems.

Performance is often gained by customizations, i.e., by using less generic solutions. Common examples are application specific buffering of persistent data, and bypassing layers in multi-layer architectures. Customization of functions tends to increase the amount of application code, since code which is normally located in the operating system or in third party software has to be dealt with on an applications level. Previous experience indicate that the maintenance cost is strongly related to the amount of application code. Consequently, there are reasons to believe that a too strong focus on performance may lead to large and complex software, and thus poor maintainability.

Our second hypothesis is that the best balance between performance and availability on the one hand and maintainability on the other is obtained by focusing on maintainability in the software design and using high-end hardware and software execution platforms for solving most of the performance and availability issues. It has at least been shown that it is rather simple to achieve high

performance in applications developed with a strong focus on maintainability by using SMP:s [8].

If our first hypothesis is true, it would be very important to find (hardware and software) architectures and design solutions which minimize the conflict between maintainability and performance. If our second hypothesis is true, we would like to estimate the additional cost (if any) for using the high-end (hardware and software) execution platforms. We would then like to compare the additional cost with the estimated reduction of maintainability costs and time to market for new versions of the products.

In order to test our hypothesis, we will look at a telecommunication application called SDP. SDP is a part of a system which in real-time rates pre-paid subscribers. The system has been developed for the cellular market by the Ericsson telecommunication company. Rating of pre-paid subscribers is a mission critical task, which requires extremely high performance and availability. The current software architecture of the SDP contains a number of design decisions for obtaining high performance and availability. The reason for this is that the designers thought that these optimizations were necessary in order to obtain the desired level of performance and availability.

The application code is very large (300,000 LOC) and complex. Functionality which normally is seen in the operating system or as licensed products, e.g. inter-process-communication and databases, is in SDP customized and "hard-wired" into the application source code.

We have, in order to test our second hypothesis, re-designed the SDP. The new prototype was designed to minimize the conflict between maintainability and performance, and according to our second hypotheses we preferred maintainability over performance. Customizations within the application code were removed and advanced functions were put back to an operating system level or to third partly software. The result was a prototype containing less than 7,000 LOC capable of handling three times more calls when the original SDP. The additional cost for the platform was 20-30% for the prototype compared to the old SDP.

The rest of the paper is structured in the following way. Section 2 gives a brief description of the SDP and the SDP architectures. In Section 3 we present our research method. The prototype architecture and results from measurements and benchmarks are presented in Section 4. In Section 5 we elaborate on our results and discuss some related work. Section 6 concludes the paper.

2. SDP

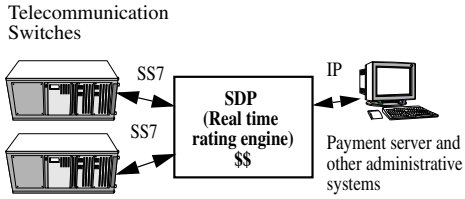


FIGURE 1. SDP OVERVIEW.

The SDP (Service Data Point) is a system for handling prepaid calls in telecommunication networks. The basic idea is simple; each phone number is connected to an account (many phone numbers can in some rare cases be connected to the same account), and before a call is allowed the telecommunication switch asks the SDP if there are any money left on the account. This initial message in the dialogue is called *First Interrogation*. The call is not allowed if there are no, or too little, money on the account.

If there are money on the account the call is allowed and the estimated amount of money for a certain call duration (e.g. three minutes) is reserved in a special table in the database (the account should not be reduced in advance). When the time period (e.g. three minutes) has expired, the telecommunication switch sends a request for an addition three minutes to the SDP. This type of message is called *Intermediate Interrogation*. The SDP then subtracts the previous reservation from the account and makes a new reservation for the next three minutes. This procedure is repeated every third minute until the call is completed. The telecommunication switch sends a message (*Final Report*) to the SDP when the call is completed, and at this point the SDP calculates the actual cost for the last period (which may be shorter than three minutes) and removes this amount from the account. Calls that are shorter than three minutes will only result in two messages, i.e., a First Interrogation message and a Final Report message. We refer to such a call as a *short call*.

The scenario above is the most common case, i.e., the case when there are enough money on the account. If there are money for a period shorter than three minutes when a First Interrogation or Intermediate Interrogation message arrives, the response to these messages will indicate that the call will only be allowed for a period x ($x < 3$ minutes), and that the call will be terminated after x time unites.

It is possible to use an *optimized dialogue* between the telecommunication switch and the SDP, and in that case no money is reserved in the First Interrogation if the amount

of money on the account exceeds a certain limit. This means that we do not need to calculate the cost for the first three minutes before the call is allowed to continue. This reduces the response time and the number of database accesses.

Figure 1 gives an overview of the SDP. The SDP communicates with the switches in the telecommunication network using the INAP and USSD protocols on top of the SS7 protocol. The most intense communication is via the INAP protocol, which is used for inquiring whether a certain call will be allowed (to continue) or not.

The USSD protocol is used for communication which makes it possible for the subscribers to obtain the current balance on their accounts. The balance is communicated in the form of a text message to the subscriber's phone. In some cases the subscriber will get a text message with the account balance automatically after a call has been completed.

The SDP communicates with administrative systems using standard TCP/IP. The most important administrative system is the Payment server from which the accounts can be refilled. Examples of other administrative system tasks are report generation and calculating statistics.

It is also possible to connect the SDP with traditional billings systems that are based on CDR (Call Data Records) processing. The SDP should therefore be able to process CDRs. Communication of CDRs to and from the SDP is done via TCP/IP.

2.1 SDP architecture

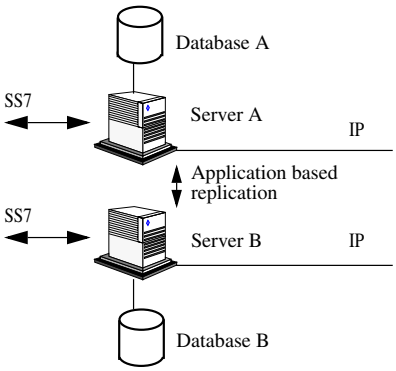


FIGURE 2. LOGIC VIEW OF THE SDP.

Figure 2 shows a logical view of the two computers (servers) in the SDP. The two Sun 4500 Enterprise multiprocessor servers [15] are connected in a mated pair. Each server has 4 processors and a SS7 connection on which SDP interacts with the telecommunication network. Performance evaluations of the SDP have shown that the performance will not increase if we put more than 4 processors in each server. Both servers are active during normal operation. If one of the servers goes down, the service is maintained by the other server.

Besides availability, performance was considered as a major potential problem. The software architecture of the SDP thus contains a number of design decisions for obtaining high performance. The reason for this is that the designers thought that these optimizations were necessary in order to obtain the desired level of performance, availability and capacity. At the time of the first design there was no time for evaluating different alternatives and the designers wanted to be sure that they would meet the performance and availability requirements. The safest way to guarantee this was to limit the use of third party software, such as commercial database systems, and instead provide the desired performance and availability by optimizations in the application code.

One performance optimization is that the application contains a cache for the entire database in RAM. The application programmer is responsible for guaranteeing that the RAM database is consistent with the one on disk. This complicates the application code. One similar example concerning availability is that the application is responsible for maintaining redundant copies of the database on the different servers. This also complicates the application code.

The mated pair solution may look rather simple. However, in order to reach the desired level of availability and performance a large amount of additional application software was needed. In SDP, this type of application code stands for over 85% of the total amount of code.

3. Method

This study consists of two parts. First, we have to find out if the conflict between maintainability and performance is bidirectional, i.e., if strong requirements on performance lead to larger and more complex application source code. We did this by studying the source code of the commercial SDP product. Secondly, we wanted to investigate if it was possible find alternative strategies which reduced the conflict thus making it possible to achieve high performance without causing maintainability problems. In order to do this, we built a new prototype of the SDP. The prototype was designed according to a three step process, which was assembled with the aim of helping the developers to achieve a high level of performance without adding more source code than necessary. Source code measurements on the prototype were compared with

identical measurements from the SDP. The attributes measured were maximum throughput and application code size, i.e., performance and maintainability respectively. Measuring maintainability via the code size is not an agreed method. However, previous experience indicate that the maintenance is strongly related to the amount of application code [11][3].

3.1 Source code measurements on the SDP

SDP consists of approximately 300.000 LOC. The measurements did, however, focus on a particular component in SDP. This was the rating engine which provides the core functionality, i.e., the part which handles calls and account withdraws. The size of this component is approximately 60.000 LOC. The source code was classified using three categories:

1. High Availability, concerning matters of system fail-over, handling of plug-in addition and removal at run time as well as on-line data replication of persistent storage.

2. Database, transforming transient storage to persistent storage.

3. Application Logic, the core business functionality. In this case the logic for rating and account handling.

The first category concerns availability. However, when the SDP was designed, it was the performance requirement which forced the developers to implement the availability facilities in the application source code instead of using a standard third party solution. The amount of code in this category can thus be considered as code improving the application performance. The next category of code is a direct result of a database optimization. Finally, in the third category, is code handling the application logic, e.g. rating calls.

The classification of the SDP source code was made by hand and with help from Ericsson personnel. The code size was measured using a software metrics tool called RSM (Resource Standard Metrics [1]). Only lines of code (LOC) were used in the results presented in this report.

3.2 The READ project

The prototype was developed in a research project called READ. The main objective of READ was to design a SDP prototype which had both high performance and maintainability. High performance was defined as fulfilling the hard performance requirement for the SDP. According to Ericsson personnel, SDP's major maintainability problem was the high complexity of the source code, which made it very difficult and time consuming to maintain the application, and the large amount of source code. Thus, less and simplified source code was considered as an effective way to increase the maintainability of the SDP.

A group consisting of six persons was assigned the task to find a new hardware and software architecture. Three out of the six came from Ericsson, two application experts and one database expert. One senior sales engineer and a free usage of Sun's and Oracles product portfolios were provided by Sun Microsystems. The project was managed and evaluated by researches at the Blekinge Institution of Technology.

READ used a three step iterative process for designing the prototype:

1. Start with the most simple architecture fulfilling the functional requirements, i.e., we do not consider any performance and/or availability requirements.

2. Start evaluating this solution. The first step of the evaluation usually consists of discussions and rough estimates based on expected performance bottle-necks, e.g., disk accesses, communication need etc. If these initial evaluations seem promising, a partial implementation is started. The implementation initially focuses on the parts of the design where one can foresee performance or availability problems. The performance and/or availability qualities of these parts are then evaluated. As long as there are no performance or availability problems, the rest of the system is gradually implemented, and if there are no problems along the way we will end up with a complete system which meets the performance and availability requirements. However, if some stage in the evaluation shows that we will not meet the performance and availability requirements, we go to step 3 in this design process.

3. Based on our accumulated experience from step 2 (we may have gone through step 2 more than once), we select a new architecture. The new architecture is the most simple architecture that meets the functional requirements, and based on what we know also the performance and availability requirements. Consequently, we do not select the minimal modification of the architecture last evaluated in step 2, instead we select the simplest architecture possible based on what we know. We now again go to step 2.

The intention of using such a process was to ensure that the most simple design solution possible was found. Each design alternative was evaluated with respect to performance (response-time and throughput), scalability

(number of subscribers for low-end and high-end) and availability (planned down-time, unplanned down-time, disaster recovery).

When defining step three of the process we discussed two alternative approaches: either to consider the minimal modification of the current architecture that would meet the performance and/or availability requirements, or to consider the simplest architecture possible (without considering the last architecture) that would meet those requirements. We expected that the first alternative would minimize the development time, whereas the last alternative would minimize the complexity and thus the maintainability of the resulting system. As shown in the process, we selected the last alternative, since maintainability was a major concern for us.

3.3 Source code measurements

The prototype has the same functionality as the rating engine component in the SDP. The SDP code measurements discussed above were therefor reused, and the corresponding measurements were done for the prototype.

3.4 Performance measurements

In order to get accurate and comparable performance measurements for the prototype a work load simulator had to be built. The simulator was designed according to the load case specified by Ericsson for the SDP. The simulator's load scenario was defined as follows:

- 17% subscriber/account inquiries via USSD
- 28% calls with no answer
- 50% successful calls with USSD notifications
- 5% low credits resulting in short calls or call refused.
- 1.000.000 different subscribers
- 1.000.000 different accounts

The maximum capacity was measured with and without optimized dialog. In our test case an optimized dialog means that 5% of the calls were made using pre-allocations setup and 95% without pre-allocations. Furthermore, the capacity was measured for long and short calls, respectively. We also evaluated the SDP for the load model described above.

4. Result

We have three major results, the code measurements on the SDP, the new architecture and the result from the source code and performance comparison measurements.

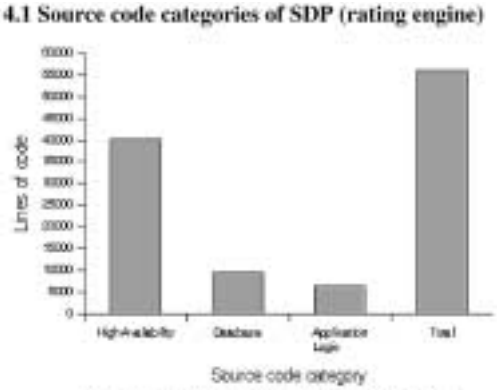


FIGURE 3. SOURCE CODE MEASUREMENTS ON SDP

The result of the source code measurements for the rating engine component in SDP is shown in Figure 3. The total amount of source code in the component is 60.000 LOC. It can be seen that less than 15% of the source code is related to functional requirements. Obviously, the hard requirement on performance (and availability) has in SDP resulted in seven times the amount of source code. The cost of maintaining an application is strongly related to the number of code lines, i.e., performance (and availability) stands for 85% of the total maintenance cost of the rating engine component.

4.2 Prototype architecture

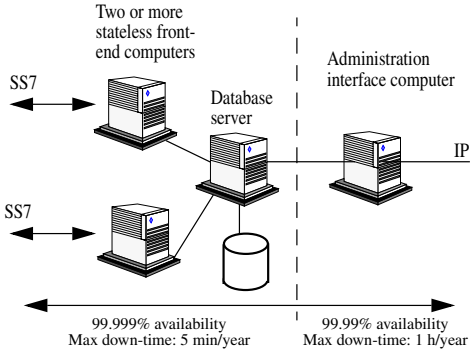


FIGURE 4. LOGICAL VIEW OF THE PROTOTYPE.

Figure 4 shows the logical view of the computers in the SDP prototype. The architecture supports two levels of availability; high and very high, 99.99% and 99.999% availability respectively. The most mission critical

functions, e.g. SS7 signalling and account withdraws, are executing on the higher level. Services less critical, e.g. report generation and administration, are running on the next highest level. The separation of a high and a very high availability reduced, in a cost-effective way, the conflict between maintainability and performance. One effect of this separation was that most of the code could benefit from standard third party software cluster solutions and thus did not have to be aware that it was running on a cluster.

There are two SS7 connections, each connected to a front-end computer. The messages from each telecommunication switch are under normal conditions split equally between these two front-end computers. If one front-end computer breaks down, the switch will send all messages to the other computer. The number of front-end computers is normally two, but may be more than two. The front-end computers are stateless, i.e. the user accounts, the subscriber database and the tariff trees are stored on the database server. Restarting or replacing a faulty front-end is thus uncomplicated. Separating the stateless parts from the rest of the application made it possible achieve high performance and high availability to a low cost.

The parts of the application which are state depended and require very high availability are hard to design. READ solved this problem by using a third party RDBMS on a fault-tolerant multiprocessor computer.

Figure 5 shows the physical view of the SDP prototype architecture. The administrative interface computer is implemented as two Sun Netra multiprocessors running Sun Cluster 3.0. Each front-end computer is a Sun Netra T1 [16]. The functionality of the system could still be maintained if one of these Sun Netra T1 computers breaks down, i.e., these computers are not a single point of failure. If one of the front-end computers breaks down, the SS7 communication bandwidth will be reduced to 50%. It one of the two computers implementing the administrative interface computer breaks down, there will be some transient disturbances during the switch over. This is, however, acceptable, since the response time requirements on this computer are soft, i.e., one can accept long response times occasionally.

The response time requirements from the SS7 side are, however, strict and they would be very difficult to meet using standard software clustering (e.g. Sun Cluster 3.0). The database server is therefore implemented as a fault tolerant multiprocessor, i.e., a Sun FT1800 [14].

The TCP/IP communication within the SDP prototype is based on two redundant networks, each network is based on a switch from Cisco.

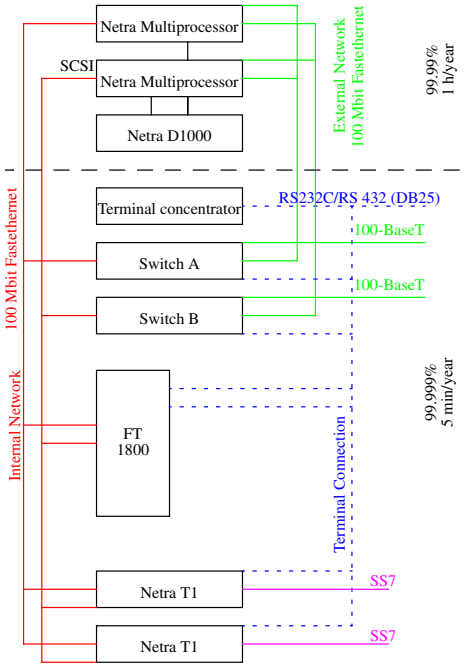


FIGURE 5. THE PHYSICAL VIEW OF THE PROTOTYPE.

The realization of the SDP prototype HW architecture ensures that there is no single point of failure, i.e., the prototype will still work if one component fails. The performance may, however, be degraded. Further, there is no single point of failure in the propriety software, apart from the store-procedures which are installed in the RDBMS. However, this “code” is executed in an encapsulated and “protected” environment. A programing error is most likely to result in that the RDBMS reports the error and continues with the next request.

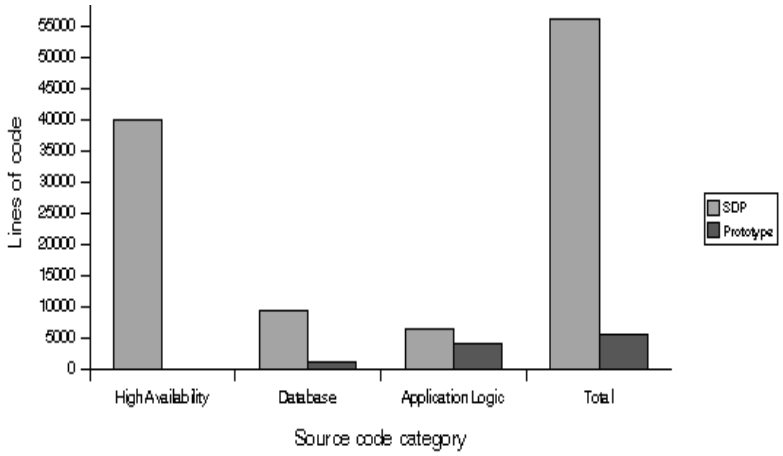


FIGURE 6. SOURCE CODE COMPARISON OF SDP AND PROTOTYPE.

4.3 Source Code Size Measurements

Figure 6 shows our result from the code size measurements. The chart displays lines of code measures for three different categories and a total. The categories are again: code used for obtaining high availability, code used for data handling and storage, and code used for application logic.

The total amount of source code in the prototype is significantly less compared to the SDP. In the SDP, the first two categories, i.e. High Availability and Database, were implemented in the application source code. In the prototype these code categorizes are, however, more or less located to third-party software or hardware. The source code used for the application logic in the prototypes is approximately 60-70% of the lines used in SDP. We believe the major reason for this is the simplified design from the READ project.

4.4 Performance Measurements

Figure 7 shows a performance comparison between SDP and the prototype. Instead of a Sun FT1800 we used a Sun Enterprise 4500 with four CPUs. The performance of the FT1800 and the Enterprise 4500 are the same; the difference is that the FT1800 provides fault tolerance. The *y*-axis in Figure 7 represents the throughput in calls per second. The maximum throughput was measured for four alternative load scenarios. These scenarios were:

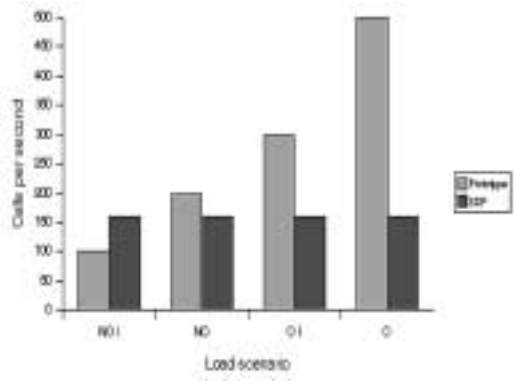


FIGURE 6. PERFORMANCE COMPARISON OF SDP AND PROTOTYPE

1. Non-Optimized dialogue with intermediate interrogations (NO I)
2. Non-Optimized dialogue without intermediate interrogations (NO)
3. Optimized dialogue with intermediate interrogations (O I)
4. Optimized dialogue without intermediate interrogations (O)

In reality, the most common case is number 4 (O), i.e. an optimized dialogue without intermediate interrogations.

Using optimized dialogue generates a significant speedup for the prototype. A reason for this is that the prototype performance is limited by the number of transactions per second. A optimized dialogue results in less transactions and thereby also a higher performance. The reason for the constant SDP performance is the architecture of the SDP. The persistent storage used by SDP is based on an object oriented database that utilizes the operating system's virtual memory algorithms for physical reads and writes. Each physical write or read always manipulates at least one whole memory page.

It is difficult to estimate the system level availability of the SDP and the prototype. The hardware availability is more or less the same for both solutions (in the SDP we have two multiprocessors, and in the prototype we have one fault tolerant multiprocessor). However, due to the large difference in code size between the SDP and prototype, we expect more errors during operation in the SDP than in the prototype. We therefore expect equal or better availability for the prototype compared to the SDP.

5. Discussion

We have in previous studies of large telecommunication applications shown that the ambition to obtain a maintainable system may result in (very) poor performance [5][6]. Our first hypothesis that we wanted to investigate in this project was if the ambition to obtain high performance (and availability) results in poor maintainability. By analyzing the code in the SDP we see that the ambition to obtain high performance and availability may result in very poor maintainability. The maintainability cost is strongly related to the code size. Further, 85% of the application code in the SDP is caused by performance and availability optimizations. We have thus been able to verify and quantify our first hypothesis, at least for this application.

The simple three step design process that we used when developing the prototype was based on our second hypothesis, i.e., that it is more difficult to afterwards increase the maintainability of a large and complex performance efficient application compared to increasing the performance of a maintainable design. The result of the three step process was the new prototype, which had a number of attractive qualities, e.g., one order of magnitude less code as well as comparable or better performance and maintainability. The experience from this project is thus a indication that our second hypothesis is true. The practical engineering implication of this is that one should use an optimistic approach and give higher priority to maintainability than performance when designing this kind of large real-time systems. One should only introduce performance optimizations when one knows that it is absolutely necessary.

The design of the prototype gave us some valuable experiences that can be formulated as three guidelines on how to handle high performance and availability requirements without introducing more application code than necessary:

- *Separate the functionality which requires (very) high performance and availability from the other functionality in the system.*

  We put the functions with high performance and availability requirements in the two front-end computers and the database server, whereas the less critical tasks are handled by the administrative interface computer.

  The application code for the less critical functionality can usually be implemented paying less attention to performance and maintainability. The desired level of maintainability and performance can usually be obtained be standard cluster software and standard multiprocessor hardware (in our case Sun cluster software and Sun multiprocessors).

- *Split the implementation of the performance demanding part of the system into a stateless and a state-full part, and try to put as much as possible of the functionality in the stateless part.*

  We have introduced two stateless front-end computers and a state-full database server computer. The state-full server executes only the Solaris operating system and the Oracle database server software, i.e., there is no "self developed" application software on this computer.

  It is generally much easier to obtain high availability and performance for stateless applications, and the cost (in terms of increases complexity and lines of code) for obtaining these qualities is thus very limited.

- *Solve as much of the availability and performance requirements for the state-full part of the performance (and availability) functionality by high-end hardware and/or third party software.*

  We introduced a fault-tolerant multiprocessor - the Sun FT1800 - for this part of the system. This machine was the key component for obtaining very high availability (99.999%) in the system.
  The state-full part of the performance (and availability)

critical functionality is the most challenging part of the system, and it is thus highly motivated to concentrate the economical resources in terms of hardware and third party software to this part.

One of the goals with the READ project was to investigate if one could buy reduced complexity (in terms of less lines of code) by using high capacity hardware, which would give us enough performance in term of capacity and switch over time to allow us to ignore (most of) the performance and availability requirements in our software design. The experience from the READ project shows that this was possible, and that the code size could be reduced with one order of magnitude.

One interesting question is how large the additional cost for the high capacity hardware really is. By using the three design guidelines above we were able to focus the economical resources to the critical parts of the system, thus reducing the additional cost to a minimum.

The cost for hardware and third party software for the prototype is approximately $300.000, provided that the Administrative interface computer is shared between a reasonably large number of database servers (e.g. between 5-10 servers). This is only 20-30% more than the hardware and third party software cost for the current SDP.

The original design of the SDP was based on the assumption that the performance and availability requirements had to be handled in the application code. The arguments for this were that the use of third party products (e.g. databases) would lower performance and one would not like to take that risk. Also, one underestimated the problem and did not anticipate the size and complexity of the resulting application code. At the start of the SDP project, the project managers were thus faced with a situation were they could either take the (presumably) safe alternative and handle the performance and availability requirements in the source code, or they could spend valuable time evaluating other alternatives which may have to be discarded due to insufficient performance and/or availability. As usual, there were very tough time-to-market requirements, and there were thus very little time to investigate alternative and less complex solutions. However, the experiences from this study show that it is important to take some time to evaluate alternative solutions, and to avoid handling the performance and availability requirements in the application code unless absolutely necessary.

We went through our three step process five times, i.e., we discarded four solutions before we found one that would meet the performance and availability requirements. Most of these solutions could, however, be discarded early without investing too much time evaluating them. We also found that the time spent evaluating these solutions contributed significantly to our understanding of the requirements of on the system. Consequently, very little, if any, time was actually wasted.

The need for considering a number of qualities (e.g. not only performance) during system design has been recognized by many researchers. In his article “The Future of Systems Research” in the 1999 August issue of IEEE Computer John Hennessy writes “Performance - long the centerpiece - needs to share the spotlight with availability, maintainability, and other qualities” [4]. Kazman et al have developed the Architecture Tradeoff Analysis Method (ATAM) which addresses the need for trade-offs between different quality attributes [9]. The main goal of ATAM is to illuminate risks early in the design process. However, ATAM does not contain any concrete guidelines regarding the software architecture and design. Lundberg et al have developed a set of guidelines and a simple design process that can be used for making design and architectural trade-offs between quality attributes [12]. Kruchten has suggested the “4+1 View Model” [10] which makes it possible to communicate and engineer different aspects of a software system, e.g., development as well as run-time aspects. Boehm and In have developed a knowledge based tool that helps the designers to identify conflicts already in the requirement phase [2].

For sure, there are a number of important issues which have to be addressed when developing application with high requirements on performance and availability. Most studies have, however, their focus on the development process or on the early development phases. This study focuses on the architectural trade-off in the design phase.

6. Conclusion

It is not always possible to maximize each quality attribute in a design, thereby making trade-offs necessary. A major challenge in software design is thus to find solutions that balance and optimize the quality attributes. Large real-time telecommunication applications is an application domain highly dependent on balancing multiple quality attributes, e.g., maintainability, performance and availability.

Earlier studies have shown that the ambition to obtain a maintainable system may result in (very) poor performance. It has, however, been shown that it is possible to achieve high performance also in applications with a strong focus on maintainability, by following a few simple guidelines which help the developers to minimize the performance problems in designs optimized for maintainability.

In this study we show that the ambition to build systems with high performance and availability may result in very poor maintainability, i.e., we have found support for our

first hypothesis (stated in the Introduction section). For the SDP application we estimate (based on code size) that 85% the maintainability cost was caused by performance and availability requirements.

The prototype developed in this project supports our second hypothesis, e.g., that it is more difficult to afterwards increase the maintainability of a large and complex application in comparison to afterwards improve the performance. We have thus found support for the two hypotheses we formulated in the Introduction section.

In the prototype, we used high-end hardware and third party software to reach the desired performance and availability levels. We also defined three guidelines that help us to focus our hardware investments to the parts where it is really motivated. These guidelines resulted in a cost-effective design, where we could obtain a dramatic reduction of time to market and maintainability costs for an additional hardware and third party software cost of 20-30% (Ericsson thought that an addition cost of 20-30% was very low considering the major reduction of the estimated maintenance cost and time to market for new versions of the system).

The original design of the SDP was based on the assumptions that the performance and availability requirements had to be handled in the application code, and there was no time to investigate alternative and less complex solutions. The experiences from this study show that it is important to take some time to evaluate alternative solutions, and to avoid handling the performance and availability requirements in the application code unless absolutely necessary. Our experience also shows that very little, if any, time was actually wasted evaluating alternative solutions.

Our experience from this project shows that hardware fault tolerant computers can indeed simplify the software design, i.e., the existing software cluster solutions are not sufficient in all cases [13].

7. References

[1] RSM, Resource Standard Metrics, http://mSquaredTechnologies.com (site visited January 2001).

[2] B. Boehm and H. In, "Identifying Quality-Requirement Conflicts", *IEEE Computer*, August 1999, pp. 27-33.

[3] J. C. Granja-Alvarez and J. Barrancp-Garcia, “A Method for Estimating Maintenance Cost in a Software Project: A Case Study”, *Journal of Software Maintenance: Research and Practice*, John Wiley & Sons, Volume 9, 166-175, 1997.

[4] J. Hennessy, "The Future of Systems Research", *IEEE Computer*, August 1999, pp. 27-33.

[5] D. Häggander and L. Lundberg, "Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor", *in Proceedings of the 27th International Conference on Parallel Processing*, August, Minneapolis, USA, 1998.

[6] D. Häggander and L. Lundberg, "Memory Allocation Prevented Telecommunication Application to be Parallelized for Better Database Utilization", *in Proceedings of the 6th International Australasian Conference on Parallel and Real-Time Systems*, Melbourne, Australia, November 1999.

[7] D. Häggander and P. Bengtsson, J. Bosch, L. Lundberg, "Maintainability Myth Causes Performance Problems in Parallel Applications", i*n Proceedings of the 3rd International Conference on Software Engineering and Application,* Scottsdale, USA, October 1999.

[8] D. Häggander and L. Lundberg, "A Simple Process for Migrating Server Applications to SMPs", *to appear in The Journal of Systems and Software*.

[9] R. Kazman, M. Barbacci, M. Klein, S.J. Carrière, "Experience with Performing Architecture Tradeoff Analysis", *in Proceedings of the International Conference on Software Engineering*, Los Angeles, USA, May 1999, pp. 54-63.

[10] P. Kruchten, "The 4+1 View Model of Architecture", *IEEE Software*, November 1995, pp. 42-50.

[11] W. Li and S. Henry, “Object-Oriented Metrics that Predict Maintainability”, *Journal of Systems Software*, 1993;23:111-122.

[12] L. Lundberg, J. Bosch, D. Häggander, and PO. Bengtsson, "Quality Attributes in Software Architecture Design", *in Proceedings of the 3rd International Conference on Software Engineering and Applications,* Scottsdale, USA, October 1999, pp. 353-362.

[13] G.F. Pfister, “*In search of clusters*”, Prentice hall, 1998.

[14] Sun Microsystems, "*Fault Tolerant Servers*", Netra ft 1800, http://www.sun.com/products-n-solutions/hw/networking/ftservers/ (site visited January 2001).

[15] Sun Microsystems, "*Midrange Servers*", Sun Enterprise 4500, http://www.sun.com/servers/midrange/e4500/ (site visited January 2001).

[16] Sun Microsystems, "*Netra t1 Server*", http://www.sun.com/netra/netrat/t1/article.html (site visited January 2001).

Paper VI

Attacking the Dynamic Memory Problem for SMPs

Daniel Häggander and Lars Lundberg
13th International ISCA Conference on Parallel and
Distributed Computing System
Las Vegas, USA
August 2000

VI

Attacking the Dynamic Memory Problem for SMPs

Daniel Häggander and Lars Lundberg
University of Karlskrona/Ronneby
Dept. of Computer Science
S-372 25 Ronneby, Sweden
{Daniel.Haggander, Lars.Lundberg}@ipd.hk-r.se

ABSTRACT

We have studied three large object oriented telecommunication server applications. In order to obtain high performance, these applications are executed on SMPs. Dynamic memory management was a serious serialization bottleneck in all applications. The dynamic memory problem was attacked in a number of ways for the three applications, and in this paper we summarize our experiences from these attacks. There are two basic ways of attacking the problem: either to reduce the cost of using dynamic memory or to reduce the usage of dynamic memory. The problem can also be attacked at different levels, i.e. the operating system level, the implementation level, and the software architecture and design level. Each of the investigated ways of attacking the dynamic memory problem has its advantages and disadvantages. We argue that the attack should focus on the operating system level when dealing with existing code and on the software architecture level when developing new applications.

Keywords

Symmetric Multiprocessor (SMP), Dynamic memory contention, Performance, Multithreading, Maintainability.

1 INTRODUCTION

Performance is usually not the primary quality attribute when developing large and complex applications. Other attributes such as maintainability and flexibility are often more important, since more effort is spent on maintaining existing applications than developing new ones [12]. Although performance is considered less important, some applications still require high performance. For such applications, improving maintainability and flexibility is useless unless the performance requirements are fulfilled. A simple and cost-effective way to improve the performance of an application is to increase the processing capacity of the hardware. The peak performance of computers has grown exponentially from 1945 to the present, and there is little to suggest that this will change. The computer architectures used to sustain growth are changing, however, from the sequential to the parallel [4]. Symmetric Multi-Processor (SMP) is the most common parallel architecture in commercial applications [6]. It is thus reasonable to assume that developers of performance-demanding applications will use SMPs since their applications must meet new and high demands on performance. To improve the application performance with multiprocessors is not as simple as migrating to a faster processor. The software architecture and/or design must usually be adapted, i.e. parallelized and optimized for the multiprocessor architecture.

Multithreading [15] allows parallelism within an application. It is characterized by low impact on the source code in C++ [22] applications since it is based on shared address space and C-library calls. Multithreading can thus be considered a simple and cost-effective way to obtain application parallelism. Designing efficient applications for multiprocessors using multithreading has, however, proven to be a far from trivial task that can easily limit the application performance [16][7][8][9][14][3].

Designing applications for multiprocessors is nothing new in high-performance computing [4]. The strong connection between multiprocessors and high-performance computing has, however, raised standards and strategies strongly focused on execution time and scalability. Other quality attributes such as maintainability and flexibility have been more or less ignored. At the same time, traditional development has taken the opposite direction. A strong focus on maintainability and flexibility has resulted in new design and implementation methodologies, e.g. object-oriented design and third party class libraries. It turns out that maintainable designs tend to use dynamic memory very frequently. The shared heap can thus become a serious performance bottleneck when running multithreaded programs optimized for maintainability on an SMP. The applications may suffer severely from dynamic memory contention [3][7][8]. We refer to this as *the dynamic memory problem for SMPs*.

We have studied three server applications developed by the Ericsson telecommunication company. These are referred to as BGw, FCC and DMO. In all three applications, maintainability and flexibility are strongly prioritized. The applications are also very demanding with respect to performance due to real-time requirements on throughput and response time. SMPs and multithreaded programming are used in order to give these applications a

high and scalable performance. The dynamic memory problem has been attacked in a number of different ways in these applications. In this paper we summarize our experiences from these attacks.

The remaining pages are organized as follows. The next section presents the BGw, FCC and DMO applications. Section 3 summarizes our findings regarding the dynamic memory problem. Related work and other aspects are discussed in Section 4, and Section 5 concludes the paper.

2 SERVER APPPLICATIONS

In this section we describe three industrial server applications. We consider the applications to be large, both in terms of code size (10.000 to 100.000 lines of code) and variety. A more detailed description of the applications can be found in [7][8].

Billing Gateway (BGw)

BGw collects billing information about calls from mobile phones. The BGw is written in C++ using object-oriented design, and the parallel execution has been implemented using Solaris threads. The BGw transfers, filters and translates raw billing information from Network Elements (NE), such as switching centers and voice mail centers, in the telecommunication network to billing systems and other Post Processing Systems (PPS). Customer bills are then issued from the billing systems. The raw billing information consists of Call Data Records (CDRs). The CDRs are continuously stored in files in the network elements. These files are sent to the BGw at certain times intervals or when the files have reached a certain size. There is a graphical user interface connected to the BGw. In this interface the different streams of information going through the gateway are visualized. Figure 1 shows an application where there are two network elements producing billing information (the two leftmost nodes). These are called “MSC - New York” and “MSC - Boston” (MSC = Mobile Switching Center). The CDRs from these two MSCs are sent to a filter called “isBillable”. There is a function associated with each filter, and in this case the filter function evaluates to true for CDRs which contain proper information about billable services. CDRs which do not contain information about billable services are simply filtered out. The other CDRs are sent to another filter called “isRoaming”. The function associated with “isRoaming” evaluates to true if the CDR contains information about a roaming call (a roaming call occurs when a customer is using a network operator other than his own, e.g when travelling in another country). In this case, the record is forwarded to a formatter, and then to a billing system for roaming calls; the record is otherwise sent to a formatter and a billing system for non-roaming calls. The record format used by the billing systems differs from the format produced by the MSCs. This is why the CDRs coming out of the last filter must be reformatted before they can be sent to the billing systems. The graph in Figure 1 is one example of how billing applications can be configured.

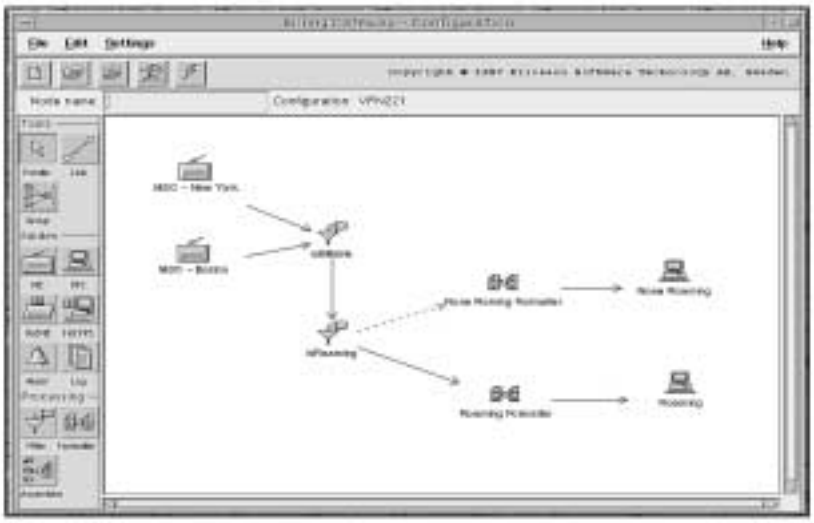


FIGURE 1. BGw CONFIGURATION WINDOW.

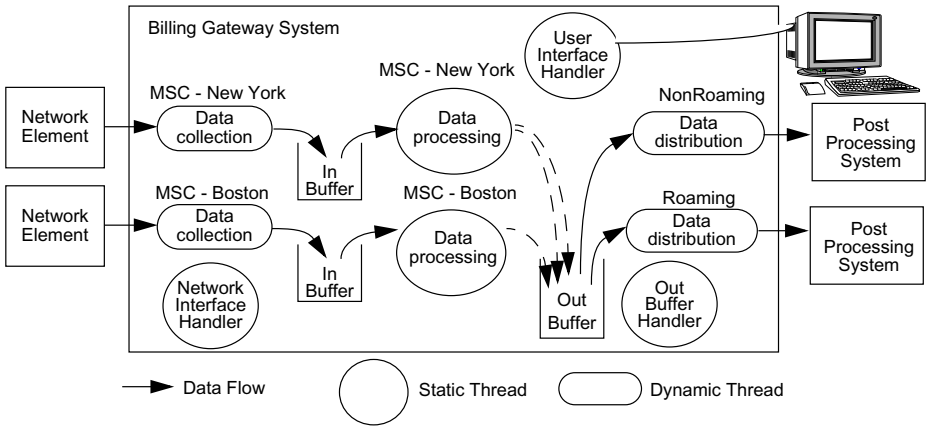


FIGURE 2. THE THREAD STRUCTURE OF BGW.

Figure 2 shows the major threads for the application in Figure 1. When there is no flow of data through the BGw, the system contains a number of static threads. When there is a flow of information going through the system, additional threads are created dynamically. When a NE sends a billing file to the BGw, a data collection thread is created. This thread reads the file from the network and stores it on disk. When the complete file has been stored, the data collection thread terminates. Data processing, i.e. the part of the BGw that does the actual filtering and formatting, is implemented in a different way, i.e. there is one data processing thread for each NE node in the configuration (see Figure 2).

The system architecture is parallel and we expected good multiprocessor speedup. The speedup was, however, very disappointing; in the first version the performance dropped when the number of processors increased because the dynamic memory management within the server process was a major bottleneck.

The designers of the BGw wanted a flexible system where new CDR formats could be handled without changing the system. One major component in the BGw is a very flexible parser that could handle data formats specified using ASN.1 [13]. This parser uses a lot of dynamic memory. In order to increase the flexibility still further, a new language which makes it possible to define more complex filters and formatters was defined. The new language makes it easier to adapt the BGw to new environments and configurations. The introduction of the new language led, however, to a very intensive use of dynamic memory, even for small configurations. The excessive use of dynamic memory was a direct result from the effort of building a flexible and configurable application. It transpired that the performance problems caused by dynamic memory management could be removed relatively easily. By replacing the standard memory management routines in Solaris with a multiprocessor implementation called ptmalloc [5], the performance was improved significantly. By spending one or two weeks on re-designing the performance was improved by approximately a factor of 8 for the sequential case, and a factor of more than 100 when using an SMP. The major part of the redesign was aimed at introducing object pools for commonly used objects and to use stack memory when possible.

Fraud Control Centers (FCC)

When cellular operators first introduce cellular telephony in an area their primary concern is to establish capacity, coverage and signing up customers. As their network matures, however, financial issues become more important, e.g. lost revenues due to fraud [17].

Software in the switching centers provides real-time surveillance of irregular events associated with a call. The idea is to identify potential fraud calls, and have them terminated. One single indication is, however, not enough for call termination. FCC allows the operator to decide certain criteria that must be fulfilled before a call is terminated, e.g. the number of indications that must be detected within a certain period of time. The events are continuously stored in files in the cellular network. These files are sent to the FCC at certain time intervals or when the files contains a certain number of events.

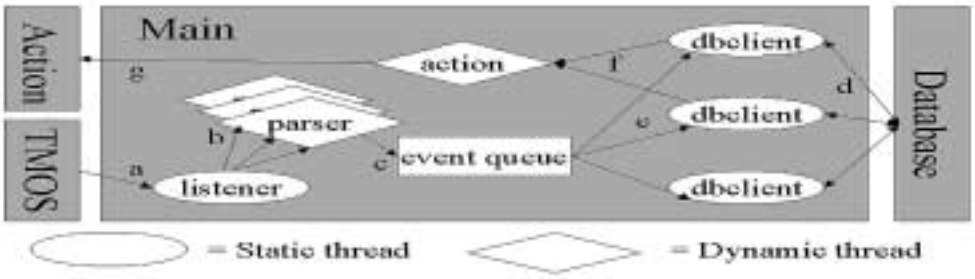


FIGURE 3. THE FCC APPLICATION.

The FCC consists of four major software modules (see Figure 3). The TMOS module is an Ericsson propriety platform and handles the interaction with the switching network. The collected events are passed on to the Main module of the FCC. The event files are parsed and divided into separate events in the Main module. The events are then stored and checked against the pre-defined rules using the database. If an event triggers a rule, the action module is notified. This module is responsible for executing the action associated with a rule, e.g. to send terminating messages to the switching network.

A central part of FCC is data storage and data processing. A commercial RDBMS (Sybase) [11] was used in the implementation. In order to improve performance. FCC has implemented parallel execution using Solaris threads. The processing within the Main module is based on threads. Figure 3 shows how the threads communicate. A listener thread receives the event file (3a) and creates a parser thread (3b). After it has created the parser thread, the listener thread is ready to receive the next file. The parser threads extract the events from the file and insert the separate events into an event queue (3c), where they wait for further processing. When all events in a file have been extracted, the parser thread terminates. The number of simultaneous parser threads is dynamic. The parser in FCC is designed in such a way that it is flexible. It is very important for the FCC to support new types of events quickly when a new network release often introduces new ones, or changes the format of old events. The Main module has a configurable number of connections toward the database server (3d). Each connection is handled by a dbclient thread. A dbclient thread handles one event at a time by taking the first event in the event queue (3e), and processing it. The interaction with the database is made through SQL commands via a C-API provided by Sybase [11]. Each SQL command is constructed before it is sent to the database module. Since the final size of a SQL command is unknown, dynamic memory must be used for its construction. The dbclient thread is also responsible for initiating actions caused by the event (3f,3g) before it processes the next event.

Dynamic memory management was found to be a performance bottleneck for FCC. There are two reasons why FCC is an intensive user of dynamic memory: the object oriented design of the parser, and the use of a string library for dynamic construction of database requests. By optimizing dynamic memory management the speedup was increased significantly. We used two different approaches for optimizing the dynamic memory handling in the FCC. One was to replace the standard memory handler with ptmalloc [5]. The other approach was to split the Main module into a number of Unix processes (Unix processes have different memory images). The performance characteristics of these two approaches were very similar [8].

Interviews with the designers showed that the reason for the frequent use of dynamic memory in the parser was that they wanted to obtain a maintainable and flexible design which would make it easy to adapt the parser to new input formats [9][10]. We implemented and evaluated an alternative parser design, which was rigid and “hard-coded”, i.e. this design was not at all optimized with respect to maintainability. The number of dynamic memory allocation was dramatically reduced in the rigid parser. The size of the rigid parser was a order of magnitude smaller than the original and flexible parser. When comparing the maintainability of these two approaches, using state-of-the-art techniques for maintainability evaluation, we concluded that the rigid parser was at least as maintainable as the flexible parser, i.e. it was at least as costly to adapt the flexible parser to a new input format as it was to write a new rigid parser. The performance of the rigid parser was more than one order of magnitude better than the performance of the flexible parser.

Data Monitoring (DMO)

The growth in the mobile telecommunication market has been dramatic in the last years and there is little to suggest that this will change. Mobile operators constantly have to increase the capacity of their networks in order to provide mobile services of good quality. Many network functions for operation and maintenance were originally designed to operate in networks much smaller than those of today. As the size of the network increases, these designs become more and more inadequate. One example is the management of performance data. Mobile networks today generate such large amounts of performance data that it can no longer be interpreted without help from computers.

FIGURE 4. THE DMO APPLICATION.

Data Monitoring (DMO) is an application that collects performance data and by using real-time analysis it detects, identifies and notifies performance problems in mobile networks (see Figure 4). The performance data (counters) are continuously stored in files in the network elements. These files are sent to the DMO at certain time intervals or when the files have reached a certain size. The received data is parsed into record objects in the DMO. Each object is then analyzed, i.e. matched against predefined rules. The DMO has two types of rules, static and historical thresholds. The historical thresholds are constantly updated for each data file received. When a performance problem is detected, the application sends a notification via an email, or via a proprietary TCP/IP interface to the operator support center. The process is supervised via a graphical user interface.

Two student groups were assigned the task of developing two DMO applications. The requirements specification was assembled by the Ericsson telecommunication company. Each group consisted of four students working full-time for 10 weeks. Even though the two groups had identical requirements as regards functionality the requirements with respect to how to develop the applications differed. The first group (DMO 1) used third party tools liberally, while the other group (DMO 2) was not permitted to use third party tools, except for the operating system (Solaris) and for certain class libraries. Both applications were required, however, to use SMPs in order to guarantee high scalable application performance. The different requirements as to how to use third party tools resulted in two different software architectures (see Figure 5).

The DMO 1 architecture (the upper part of Figure 5) is based on an RDBMS (Sybase) that is accessed via a small C++ server engine, while the DMO 2 architecture (the lower one in Figure 5) is a large object oriented and multithreaded C++ server using plain UNIX files for persistent storage. DMO 1 and DMO 2 are both supervised via a Java GUI, connected via TCP/IP.

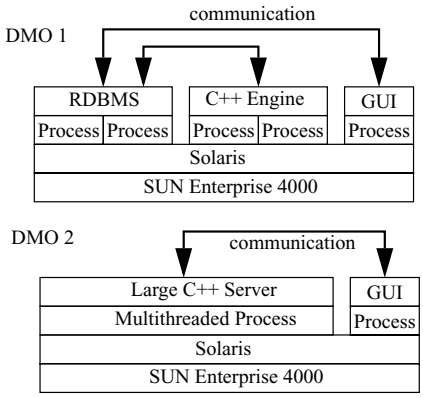


FIGURE 5. TWO ARCHITECTURES OF DMO.

DMO 2 is implemented using multithreaded programming and can thus be automatically distributed over the processors in an SMP via Solaris LWPs [15]. The DMO 1 uses a different technique for making use of multiple processors. In DMO 1, the parallelism is achieved by starting a number of DMO 1 engines instances, each as a separate UNIX process. This makes it possible for DMO 1 to execute on multiple processors in parallel.The DMO 1 application scaled-up well (almost linearly) on an 8-way SMP. The DMO 2 application had, however, a most inefficient scale-up. The throughput decreased when using more than one processors. The bottleneck in the dynamic memory management was the main reason for this. We believe that an optimized dynamic memory handling would make the application scale-up properly.

The DMO 1 application scaled-up well (almost linearly) on an 8-way SMP. The DMO 2 application had, however, a most inefficient scale-up. The throughput

decreased when using more than one processors. The bottleneck in the dynamic memory management was the main reason for this. We believe that an optimized dynamic memory handling would make the application scale-up properly.

While studying the DMO 2 project we found that the developers were divided into two categories: one category thought that multithreading simplifies both design and the implementation; the other thought that multithreading makes design and implementation harder. How well developers adapt to parallel programming seems to be very individual.

3 LESSONS LEARNED

The experiences from the BGw, FCC and DMO applications show that the design techniques used for obtaining high maintainability and flexibility result in very frequent allocation and deallocation of dynamic memory at run time. Our experience also shows that dynamic memory management has a serious impact on the SMP performance of these applications. The main reason for this is that the heap is a shared resource in a multithreaded program, i.e. the heap can easily become a serialization bottleneck when using an SMP.

As discussed in the previous section, the dynamic memory problem in SMPs has been attacked in a number of ways in the three applications, e.g. by using a parallel heap (ptmalloc), by using processes (with a private heap) instead of threads, or by introducing object pools. In this section we summarize the experience we have of attacking the dynamic memory problem in SMPs (see Table 1).

There are two basic ways of attacking the dynamic memory problem: either to reduce the cost of dynamic memory allocation and deallocation, or to reduce the usage of dynamic memory. For each of these two basic ways, the attack can be made on different levels. We have found it useful to distinguish between three levels: software architecture and design level, implementation level, and operating system level.

Starting from the bottom in Table 1 we see that using a parallel heap is a useful way of reducing the cost for dynamic memory allocation and deallocation. We have evaluated ptmalloc for BGw and FCC, and found that it solved the dynamic memory problem for these applications, at least when the number of processors was less than or equal to eight. Other studies [1] have, however, showed that for some applications even highly optimized parallel heaps such as ptmalloc and smartheap can suffer from serious dynamic memory problems even for a small number of processors, e.g. less than eight processors.

One very successful way of attacking the dynamic memory problem is to introduce objects pools. We have evaluated this in the BGw and found that it does not only provide very good multiprocessor speedup, it also improves the single-processor performance significantly. The reason for this is that allocation of memory from object specific pools can be done much faster than allocation of memory from a general heap. The major disadvantage with object pools is that they introduce additional complexity in the application program (i.e. we move the task of dynamic memory management from the operating system to the application programmer). The additional complexity makes it harder to maintain the program, thus negating the main reasons for using object orientation and maintainable design techniques in the first place. Experience from the BGw shows that the object pools tend to degenerate after a number of revisions of the application. The reason for this is that the designers that are changing the application from one revision to the next do not fully understand how to use the object pools. The result of this is that they start to use the ordinary heap instead.

A simple but effective way of attacking the dynamic memory problem at the implementation level is to use the stack instead of the heap. This is of course not possible in all cases, but experience from the BGw shows that a bad habit is to use the operator new, even if the size is constant (see the code examples on next side).

```
readAndPrint(int fd) {
unsigned char* buff =  new unsigned char[16];
read(fd, buff,  16);
printf("%s",buff);
delete buff;}
```

Table 1: Summary of our experiences of attacking the dynamic memory problem in SMPs

Level \ Type	Reduce cost	Reduce usage
Architecture and Design	Processes instead of threads	Rigid but exchangeable components
Implementation	Object pools	Use stack instead of heap
Operating System	Parallel heap	Not investigated by us

An alternative is to use memory from the stack.

```
readAndPrint(int fd) {
char buff [16];
read(fd, buff,  16);
printf("%s",buff);}
```

A typical situation where dynamic memory is used is when the size of the allocated memory has to be decided at run-time. In such cases, the C library function, alloca() can be used instead of the operator new. The function allocates a dynamic number of bytes from the stack.

The dynamic memory problem can also be attacked at the architecture and design level. One way of reducing the cost for dynamic memory allocation in SMPs is to split the application into a number of processes that each have their own heap. Some applications, e.g. applications that operate on one large data structure, may be difficult to split into processes. We have, however, found that it was possible to split the FCC and the DMO (i.e. DMO 1) applications into a number of processes without any problems. The versions of FCC and DMO that consist of a number of processes do not suffer from the dynamic memory problem, whereas the multithreaded versions do suffer from this problem.

The dynamic memory problem can also be attacked by changing the design style from very flexible components to more rigid ones. The detailed study of the parser component in the FCC, showed that a design based on rigid but exchangeable components can in some cases be at least as maintainable as a design based on components which are designed for maximum flexibility [9][10].

The techniques presented in Table 1 all have their advantages and disadvantages, and they can of course be combined. When working with existing code the best choice is probably to start with using a parallel heap and see if this solves the problem. If a parallel heap is not sufficient one should consider the techniques on the implementation level as a second step, and only as a very last resort consider altering the architecture and design of the system.

When developing new code we would, however, suggest that the designers start at the top in Table 1, i.e. they should consider using processes instead of threads when possible (and reasonable), and they should also consider using rigid but exchangeable components instead of flexible and extendable components.

We believe that it is a better long term solution to reduce the usage of dynamic memory compared to just reducing the cost for using dynamic memory. The reason for this is that it is generally easier to add on techniques for reducing the cost later, e.g. you can always relink the program with a parallel heap at a later stage.

4 RELATED WORK AND DISCUSSION

With the exception of Ptmalloc [5], Smartheap [18] and Hoard [1], we have not found many studies of parallel memory allocators. Larson and Krishan [14] also suggest a parallel heap and they include a relatively good survey.

High-performance algorithms for dynamic memory allocation have been, and will continue to be, of considerable interest. Benjamin Zorn has addressed the subject in a number of papers, e.g. [2][23]. These papers are, however, mostly focused on dynamic memory allocations in sequential applications. A handful of papers presenting new and adapted state-of-the-art solutions, e.g. design patterns, have been written. Douglas C. Schmidt is one of the leading researchers in this area [21]. The papers show that parallel software introduces the need for new patterns and idioms [20], and it also requires that existing, well-known and successful patterns are adapted or complemented in order to operate efficiently in the new context. One example is singleton, a very common pattern, which has proved to be an incomplete solution in parallel software since it only addresses the issue of creating objects [19]. In order to work properly in parallel software, it also has to address the issue of object destruction.

Our experience indicates that some designers like multithreading while others do not. It is, however, clear that the usage of multiprocessors and multithreading creates a number of new problems. We believe that problems with immature third party tools will decrease as SMPs become common in this type of applications; our positive experiences from the database management systems in FCC and DMO support this.

A challenge is to find design strategies which can offer scalable performance for software which is not primarily designed for SMPs or cannot be highly optimized for SMPs. We have ascertained that dynamic memory allocation and deallocation is the most common bottleneck when executing these types of object oriented applications on SMPs.

5 CONCLUSION

We have studied three large object oriented telecommunication server applications, for a period of five years. The applications have been studied in their industrial context as well as in an experimental environment - an 8-way Sun Enterprise multiprocessor system - on which the applications have been evaluated, optimized and then re-evaluated. The studies also include a number of interviews with developers at Ericsson.

One important finding is that dynamic memory management was a serialization bottleneck in all applications, i.e. there is a dynamic memory problem for SMPs for this type of applications. The large number of memory allocation and deallocation are caused by the designers' ambition to obtain a maintainable and flexible software design.

The dynamic memory problem was attacked in a number of ways for the three applications. Our experiences from these attacks could be summarized in a table (Table 1). This table shows that the problem can be attacked at different levels, i.e. the operating system level, the implementation level, and the software architecture and design level. The table also shows that there are two basic ways of attacking the problem: either to reduce the cost of using dynamic memory or to reduce the usage of dynamic memory. The investigated methods can be combined.

Each of the investigated ways of attacking the dynamic memory problem has its advantages and disadvantages. We argue, however, that the attack should focus on the operating system level when dealing with existing code and on the software architecture and design level when developing new applications.

REFERENCES

1. E. D. Berger and R. D. Blumofe, "Hoard: A Fast, Scalable, and Memory-Efficient Allocator for Shared Memory Multiprocessors", http://www.hoard.org.
2. D. Detlefs, A. Dosser, and B. Zorn ''Memory Allocation Costs in Large C and C++ Programs'' *Software Practice and Experience* 24(6):527-542, June 1994
3. R. Ford, D. Snelling and A. Dickinson, "Dynamic Memory Control in a Parallel Implementation of an Operational Weather Forecast Model", *in Proceedings of the 7:th SIAM Conference on parallel processing for scientific computing*, 1995.
4. I. Foster, "*Designing and Building Parallel Programs*", Addison Wesley, 1995.
5. W. Gloger, "Dynamic memory allocator implementations in Linux system libraries", *"http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html"*.
6. K. Hwang, Z. Xu, "*Scalable and Parallel Computing*", WCB/McGraw-Hill, 1998.
7. D. Häggander and L. Lundberg, "Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor", *in Proceedings of the ICPP 98, 27th International Conference on Parallel Processing*, August, Minneapolis 1998.
8. D. Häggander and L. Lundberg, "Memory Allocation Prevented Telecommunication Application to be Parallelized for Better Database Utilization", *in Proceedings of the 6th International Australasian Conference on Parallel and Real-Time Systems*, Melbourne, Australia, November 1999.
9. D. Häggander and P. Bengtsson, J. Bosch, L. Lundberg, "Maintainability Myth Causes Performance Problems in Parallel Applications", *in Proceedings of SEA'99, the 3rd International Conference on Software Engineering and Application, Scottsdale,* USA, October 1999.
10. D. Häggander and P. Bengtsson, J. Bosch, L. Lundberg, "Maintainability Myth Causes Performance Problems in SMP Applications", *in Proceedings of APSEC'99, the 6th IEEE Asian-Pacific Conference on Software Engineering,* Takamatsu, Japan, December 1999.
11. J. Panttaja, M. Panttaja and J. Bowman, "*The Sybase SQL Server -Survival Guide*", John Wiley & Sons, 1996.
12. T. M. Pigoski, "*Practical Software Maintenance*", Wiley Computer Publishing, 1997, table 3.1, page 31.
13. J. Larmouth, *"ASN.1 Complete"*, Morgan Kaufmann, 1999.
14. P. Larsson and M. Krishan, "Memory Allocation for Long-Running Server Applications", *in Proceedings of the International Symposium on Memory Management*, ISMM '98, Vancouver, Canada, October, 1998.
15. B. Lewis, "*Threads Primer*", Prentice Hall, 1996.
16. L. Lundberg and D. Häggander, "Multiprocessor Performance Evaluation of Billing Gateway Systems for Telecommunication Applications", *in Proceedings of the ISCA 9th International Conference in Industry and Engineering*, December, Orlando 1996.
17. C. Lundin, B. Nguyen and B. Ewart, "Fraud management and prevention in Ericsson's AMPS/D-AMPS system", *Ericsson Review No. 4*, 1996.
18. Microquill, "SmartHeap for SMP", http://www.microquill.com/smp.
19. D. Schmidt, "A Complementary Pattern for Controlling Object Creation and Destruction", *in Proceedings of the 6th Pattern Languages of Programs*, Monticello, Il, US, August, 1999.
20. D. Schmidt, "Patterns and Idioms for Simplifying Multi-threaded C++ Components", *to appear in the C++ Report Magazine*.
21. D. Schmidt, "Douglas C. Schmidt", http://www.cs.wustl.edu/~schmidt.
22. B. Stroustrup, "The C++ Programming Language", Addison-Wesley, 1986.
23. B. Zorn and D. Grunwald, ''Evaluating Models of Memory Allocation,'' *ACM Transactions on Modeling and Computer Simulation*, 4(1):pp.107-131, January 1994.

Paper VII

A Method for Automatic Optimization of Dynamic Memory Management in C++

Daniel Häggander, Per Lidén and Lars Lundberg
30th International Conference on Parallel Processing
Valencia, Spain
September 2001

VII

A Method for Automatic Optimization of Dynamic Memory Management in C++

Daniel Häggander, *Per Lidén and Lars Lundberg*
Department of Software Engineering and Computer Science
Blekinge Institute of Technology
SE-372 25 Ronneby, Sweden
{Daniel.Haggander, Per.Liden, Lars.Lundberg}@bth.se

Abstract

In C++, the memory allocator is often a bottleneck that severely limits performance and scalability on multiprocessor systems. The traditional solution is to optimize the C library memory allocation routines. An alternative is to attack the problem on the source code level, i.e. modify the applications source code. Such an approach makes it possible to achieve more efficient and customized memory management. To implement and maintain such source code optimizations is however both laborious and costly, since it is a manual procedure.

Applications developed using object-oriented techniques, such as frameworks and design patterns, tend to use a great deal of dynamic memory to offer dynamic features. These features are mainly used for maintainability reasons, and temporal locality often characterizes the run-time behavior of the dynamic memory operations.

We have implemented a pre-processor based method, named Amplify, which in a completely automated procedure optimizes (object-oriented) C++ applications to exploit the temporal locality in dynamic memory usage. Test results show that Amplify can obtain significant speed-up for synthetic applications and that it was useful for a commercial product.

1. Introduction

Most applications use a great deal of dynamic memory these days. As the trend towards larger applications continues, there will be even greater demands on the heap manager. Another trend that increases the importance of efficient heap management is the increasing popularity of C++ [12]. C++ programs by their nature use dynamic memory much more heavily than their C counterparts, and often for many very small, short-lived allocations [8]. Previous studies show that C++ programs invoke one order of magnitude more dynamic memory management than comparable C applications [3]. Modern software systems also tend to use various frameworks and design patterns. A main reason for this is to make the source code easy to maintain. Frameworks and design patterns introduce dynamic features. These features are to a large extent based on abstract object types and the introduction of new object types. In C++ this increases the use of dynamic memory even more. However, the dynamic features present in many frameworks and design patterns are mainly used to make the system adaptable to future changes. Thus, temporal locality will characterize the run-time behavior, where the same object structures tend to be created and used over and over again [9].

On multiprocessors, dynamic memory is an exclusive resource that must be protected against concurrent accesses by multiple threads running in the same multithreaded (C++) program. As a result of this, it is often not useful to add more computation power to a system by using multiple CPU:s, since frequent allocation and deallocation of dynamic memory becomes a bottleneck [11]. Several methods for solving this problem have been evaluated [10]. A common approach to make the memory management routines reentrant. This is, usually, solved by creating a dynamic memory manager that handles multiple heaps internally [1][4][5]. This way, the allocation and deallocation routines can work in parallel with a minimum need for mutual exclusion. Nevertheless, evaluations have shown that application specific design and implementation techniques, such as introducing handmade structure pools, is more beneficial in terms of performance [11]. However, this method suffers from some major drawbacks, which makes it a less attractive option. Using pools for reuse of object structures requires the programmer to incorporate these into every class by hand. This means that the programmer must be very familiar with the code. Further, these pools tend to be very tailored and optimized for a specific application, thus making them less reusable. As the system grows the pools must be maintained to reflect changes in the system. In the end, this all adds up to a time consuming activity requiring extensive knowledge of the system's internal structure. Also, since the pools are handmade there is a potential source for errors.

It would be beneficial if the process for incorporating structure pools in a system could be automated. Most of the identified drawbacks of this method could then be eliminated, or at least substantially reduced. We have approached this problem by implementing a pre-processor that inserts the optimizations into the code before it is compiled. The problems with high costs for maintaining a handmade version are eliminated, since it is only a matter of recompiling the code. Further, the potential errors caused by the programmer creating the pools are eliminated (as long as we assume that the pre-processor implementation is correct). Finally, there is no need for special expertise in this area among the developers in a team. Instead they can go on using the traditional programming and design methods and use the pre-processor when compiling the system.

Our results show that the performance speedup gained by using handmade structure pools can be preserved, however not completely, by using a pre-processor based automation. On synthetic programs, Amplify (our method) has shown to be up to six times more efficient, compared to the available, state of the art, C library allocators we have tested. This study also includes tests on a commercial product developed by the Ericsson Telecommunication Company, the Billing Gateway (BGw). BGw is a system for collecting billing information about calls from mobile phones. The application is written in C++ and is an example of a product dependent on a parallel memory management to scale-up on multiprocessors. The tests show a 17% performance increase when the BGw source code was pre-processed by a slightly modified version of Amplify.

The remaining pages are organized as follows. Section 2 describes in further detail problems surrounding dynamic memory. In Section 3 we describe the Amplify method. The test method is presented in Section 4. Our results can be found in Section 5, while Section 6 is a short presentation of related work. Finally, Section 7 concludes the paper.

2. Problems with dynamic memory

In C++ [12], dynamic memory is allocated using the operator `new`*, which normally is an encapsulation of the C-library function* `malloc()`*. Allocated memory is deallocated using operator* `delete`*, which normally is encapsulating the C-library function* `free()`*. Many implementations of* `malloc()` *and* `free()` *have very simple support for parallel entrance, e.g. using a mutex for the function code. Such implementations of dynamic memory result in a serialization bottleneck. Moreover, the system overhead generated by the contention of entrance can be substantial [11]. The bottleneck can be reduced by using an optimized reentrant implementation of* `malloc()` *and* `free()`*, e.g. ptmalloc [4] or Hoard [1]. However, even better performance can be achieved by decreasing the number of heap allocations, e.g. by implement structure pools [9]. A lower number of heap allocations will also improve the performance on uni-processors, whereas using a reentrant implementation of* `malloc()` *and* `free()` *will only improve performance on multiprocessors.*

In an object-oriented design, especially with design patterns and frameworks, a large number of objects are used. Creating an object often requires more than one dynamic memory allocation (call to operator `new`*). The reason for this is that each object often consists of a number of aggregated or nested objects. For example, a car can be represented as a number of wheel objects, an engine object and a chassis object. An engine object may use a string object, usually from a third-party library, for its name representation and so on (see Figure 1).*

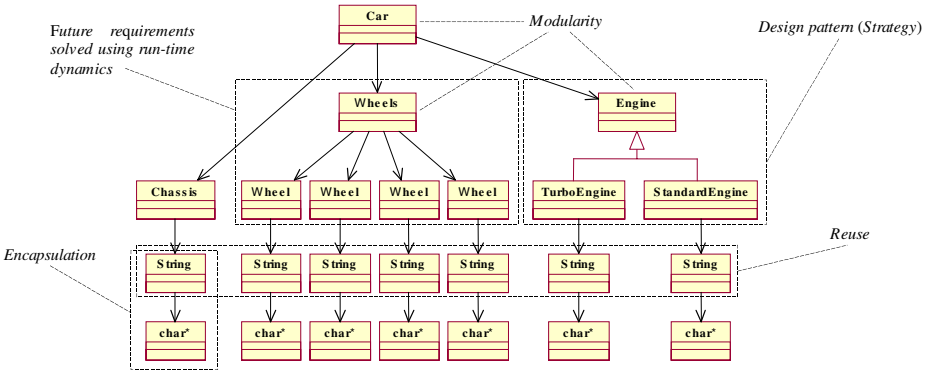


Figure 1. An object representation of a car

It is not rare that these objects are combined at run-time in order to be as adaptable as possible. For example, a car could have an unspecified number of wheels. Such a design requires each sub-object (Wheel) to be created separately. As a result of this, dynamic memory is allocated and deallocated separately for each sub-object. The total number of allocations is, according to the discussion above, dependent on the composition of the objects. Every design decision, which affects the composition of an object, will therefore also affect the number of memory allocations and deallocations during program execution. Large numbers of allocations and deallocations often become a big performance problem, especially on multiprocessor hardware. Actions must therefor often be taken, such as redesigning the application or using an alternative heap manager.

2.1. Structure pools

The concept of object pools, also known as memory pools, is a well-known memory management technique. An object pool acts as a layer between the application and the dynamic memory management subsystem. When allocating memory for an object, a call to an object pool is made instead of making direct calls to the memory manager, i.e. `malloc()`. *An object pool holds a free list containing objects of a specific type. When an object is requested an object is extracted from the free list and returned to the caller. Thus, no call to the memory manager is needed, instead an already allocated, but not currently used, object is reused. Further, when deallocating memory a call to the pool is made instead of directly calling the memory manager, i.e.* `free()`. *The pool will then insert the object into its free list for later reuse. This kind of strategy reduces the number of calls made to the memory management subsystem. However, in a multithreaded environment there is still a need for mutual exclusion during operations on the pool's free list.*

Using object structures, rather than single objects, as the reusable units in a free list leads to what we call structure pools. Each object that contains references to other objects is a potential root node in an object structure. When an object is deallocated it is placed in the free list, keeping its references to other objects. This means that if we would like to allocate a Car object (see Figure 1) we only need to access the Car's pool once to get the complete car with an engine, wheels and chassis. If we were using traditional object pools we would have had to access the pools of each and every class used by Car to build the same structure. By using pools to store object structures we not only reduce the number of calls made to the memory management subsystem, but we also reduce the number of calls made to pools.

The details on how these pools are implemented are discussed further in Section 3.

3. Amplify

When generalizing and automating the process of using structure pools we started by writing and analyzing handmade versions. During this process we found several points where the programmer makes decisions concerning the application source code or design. These decisions had to be made by the pre-processor when the process was automated. However, a pre-processor can never know as much about the source code as the programmer. Design decisions, rationale, unwritten coding rules, etc, are things a pre-processor can not comprehend. We therefore designed Amplify (which is the name of our method) to solve these problems in a general way, not depending on information unavailable to the pre-processor. By inspecting the input source code using a pattern matching approach we were able to modify the code to make use of structure pools.

3.1. Building structure pools by hand

In the case where a structure pool is handmade, the programmer selects the root object of the structure (i.e. Car in Figure 1). The programmer then creates a new class that handles a pool of such root objects. The new pool class typically has three static member functions, `init()`, `alloc()` *and* `free()` *(see Figure 2). These functions manage a free list. Whenever the programmer wants to create or destroy a new structure he/she must not use operator* `new` *or* `delete`, *but instead make calls to the pool's* `alloc()` *and* `free()` *functions, which removes respectively inserts an object into the free list. It is not only up to the programmer to do the right thing when allocating and deallocating a structure, he/she is also responsible for writing the pool's* `alloc()` *and* `free()` *member functions. Further, before starting to make calls to* `alloc()`, *the pool has to be initialized by a call to* `init()`. *This is done in order to let the pool pre-allocate a number of structures that is inserted into the free list.*

Since structures may not be identical every time they are used, the pool can only pre-allocate the objects that are common to every case (from now on referred to as template). In the case with the Car structure, a template would consist of a Car, a Chassis, and a Wheels object. Everything else is dependent on what kind of car is to be created. However, there are cases where the programmer knows that he will only create cars with a maximum of eight wheels. In these cases eight wheels are allocated and inserted into the template. Later, when a template is used, there are always eight wheels available. However, not all of them are used if the car only has four wheels. This memory overhead might be acceptable if it yields better performance. At this point the object structure has everything except an engine. The type of engine used can

```
class Car {
public:
   virtual void init(int numberOfWheels, ...); // Initialize
   virtual void destroy();                     // Clean up
...

class CarPool {
public:
   static void init(int numberOfCars); // Pre-allocate a number of structures
   static Car* alloc();                // Get a structure from the pool
   static void free(Car* car);         // Return a structure to the pool
...
```

Figure 2. Example of handmade pool

vary from time to time and must thus be allocated separately when the car is created.

Moreover, since the traditional memory management functions are bypassed when using these kinds of pools another addition is often made to the classes that compose the structures. It must be possible to initialize and destroy objects without calling operator `new` *and operator* `delete`*. Thus, each class in a structure gets two new member functions,* `init()` *and* `destroy()`*. These functions act as replacements for the traditional constructor and destructor (see Figure 2).*

Finally, the pools must be usable in a parallel environment. A solution to this would be to protect parts of the pools' code with mutual exclusion. However, this is often not a desirable solution since global locks are potential bottlenecks (and what causes `malloc()` *and* `free()` *to be inefficient). Instead the programmer keeps track of which pools are used by which threads and manually avoids simultaneous allocations/deallocations from the same pool.*

3.2. The Amplify method

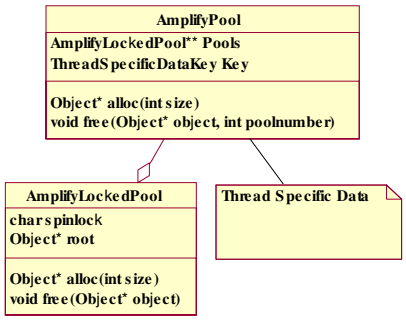


Figure 3. Structure of an Amplify pool

Automatically modifying the source code of an existing program requires the decisions made by the programmer to be generalized to a level comprehensible to the pre-processor. Therefor, we introduce a general structure pool (see Figure 3). The general structure pool must be flexible enough to fit every case and work completely transparent to the programmer. The following sections will describe the generalization.

When building a structure pool by hand, the programmer selects the root object of the structure. The programmer knows which structures are frequently used and can thus decide to only use structure pools in cases where he/she knows it will be beneficial. However, by just looking at the class structure in the source code we can not decide which objects are roots and which are not during execution. Since each object is a potential root node in a structure we can not during pre-processing treat some classes differently from others. Instead we treat every class as if it was a root in a structure, i.e. letting each class have its own pool.

The common parts of a structure were selected by the programmer and composed into a template. The characteristics of a template is based on design decisions and assumptions, e.g. "we will not create a car with more than eight wheels". It is obviously impossible for a pre-processor to make such assumptions and thus the contents of a template must be based on some other information.

One fundamental idea behind Amplify is to exploit temporal locality, which often characterizes object-oriented programs using frameworks and design patterns. In this case, it means that when the program creates a new structure we assume that it will be identical to the structure last deleted (which had the same type of root object). If we later find that the structures were not identical we will then take the overhead of reorganizing the structure to fit this specific case.

Before the programmer started to use a handmade pool he/she called `init()` *to pre-allocate a number of structures. The programmer knows that he/she will never use more structures simultaneously than present in the pool after the initialization.*

Instead of pre-allocating a number of structures, Amplify will start with empty pools. If a pool is empty when the application makes a memory request, the pool

itself will simply allocate more memory using the normal dynamic memory manager, i.e. malloc().

To allocate a structure using handmade pools, all traditional allocations (using operator new*) are exchanged with a call to the member function* alloc() *of the corresponding pool. Further, to be able to initialize an already allocated structure the member function* init() *is called, instead of the constructor.*

Amplify solves this by overloading operator new *of each class that is associated with a pool. Operator* new *redirects all memory requests to the pool's member function* alloc(). *This function will extract a structure from the pool's free list and return it. Only if the free list is empty a new piece of memory is allocated on the heap.*

If the pre-processed class already has a new *operator defined, the pre-processor will respect that and not generate another one.*

One problem with the standard malloc() *and* free() *is the critical code regions that must be protected by mutual exclusion. A pool has the same fundamental problem. Only one thread can manipulate the pool's free list at the same time, else there would be a potential source for race conditions that could cause a program to fail. When developing handmade pools, the programmer often avoids this problem by simply not sharing a pool between two threads. A pool could of course be reused, but only when the thread currently using it has been terminated.*

One goal when developing Amplify was to avoid expensive locking schemes. In particular, we would like to minimize the situations where a thread has to wait for a lock that is already occupied. This was done in several ways. First, having one pool for each class leads to a more parallel behavior since two threads will only compete with each other if they both try to allocate an object of the same type. Secondly, we used known strategies, mainly from ptmalloc, to spread the threads over a number of pools to avoid lock contention on a multiprocessor.

Freeing a structure when using handmade pools is done by calling the member function destroy() *of the root object. This function acts as the destructor for the object and will in turn call the* destroy() *function of its children. When this is completed all objects in the structure have done their necessary clean up actions (releasing resources, closing files, etc) and the structure can be considered dead. The programmer now calls the* free() *function of the pool associated with the root. This will insert the structure into the pool's free list for later reuse. It is important to note that the relation between objects in the structure is kept intact, i.e. no pointers are deleted or changed.*

The Amplify pre-processor is faced with two problems here. First, when deleting a structure the root object must be placed in the corresponding pool's free list. Second, to be able to reuse the structure later, the object structure must be preserved. Preserving the structure means that child objects of the root must not be deleted or placed in other pools.

These problems are solved by the following two actions. First, operator delete *is overloaded on all classes associated with a pool. The* delete *operator will redirect all deletion requests to the pool of the class. This is done to avoid letting the memory be passed back to the heap manager. Instead* delete *adds the object to the corresponding pool's free list. Doing this, we allow the memory occupied by the object to be reused the next time such an object is to be created.*

The second step is to preserve the object structure, e.g. avoid deletion of child objects of the root. Traditionally when an object structure is deleted, references (pointers) to other objects are deleted as well. Thus, we have to store information about the structure that outlives a deletion of its objects. This information must make it possible to rebuild the same object structure again (preferably with no or little effort). To solve this we introduce what we call shadow pointers. Shadow pointers are additional fields in a class, replicating all pointer fields in a class. These additional fields are added by the pre-processor, thus they are completely invisible to the programmer.

```
class Root {
public:
   // Methods
private:
   Child* left;
   Child* right;
   int data;
   Child* leftShadow;
   Child* rightShadow;
};
```

The shadow pointers are used in the following way. When a new Root *object is allocated on the heap (i.e. the pool was empty) all shadows are set to* 0 *(null). In* Root*'s methods where the following code appears*

```
delete left;
```

the pre-processor will change it to

```
if (left) {
   left->~Child();
   leftShadow = left;
}
```

This is done to avoid deletion of left*. Instead the object's destructor is called and then the address to the memory occupied by the object is saved in its shadow pointer. By doing this we can keep track of the object structure after it has been deleted. The next time the same structure is created we can reuse the previous one by just reading the shadow pointers. Thus, in* Root*'s methods*

where the following code appears

```
left = new Child(...);
```

the pre-processor will change it to

```
left = new(leftShadow) Child(...);
```

By also overloading operator new *we can add type checking to ensure that there is enough space for the new object.*

4. Method

As a first step, we developed three synthetic test programs to evaluate the potential of Amplify. All test programs used dynamic memory extensively and had a run-time behavior characterized by temporal locality, i.e. the program frequently created the same object structures over and over again. The three programs differed from each other by how deep object structures they where creating. The test programs were executed and compared using three alternative memory management solutions, ptmalloc [4], Hoard [1], and Amplify.

The test suite used was based on a program with 100% temporal locality behavior, i.e. creating the same structure over and over again. This was done by creating a number of threads, which allocates, initializes and then destroys and deallocates binary trees. Each node was 20 bytes (28 bytes when "amplified") in size, holding two pointers to its children and some "dummy" data.

The program was set up to vary its behavior with respect to the number of threads created, the number of binary trees created and destroyed, as well as the depth of the binary trees. No system calls were made during execution, thus making it theoretically possible for ideal scalability.

Three test cases (see Table 1) were chosen based on the following:

Test case	*Tree depth*	*Number of objects*
1	*1*	*3*
2	*3*	*15*
3	*5*	*63*

Table 1. Size of data structures in test cases.

- *Test case one was chosen as a worst case, to see if the Amplify-approach would cause a large overhead when data structures are shallow. Since Amplify tries to reuse data structures it is clearly of interest to see how well it performs if there are no large structure to reuse.*
- *Test case two can be mapped to the Car-paradigm (see Figure 1) and thus represents something that could, based on the discussion in sections 1 and 2, be considered as a normal case for an application.*
- *Test case three could be thought of as the best case, where deep structures could be reused.*

The tests were executed under Solaris 2.6 on a Sun Enterprise 4000 equipped with 8 processors. Solaris default memory manager implementation was used as the baseline for calculating speedup. In a second step, we applied Amplify on a real application, Billing Gateway (BGw). By doing this we identified a number of necessary adaptations (see Section 5.2).

The Billing Gateway (BGw) is a system for collecting billing information about calls from mobile phones. The system is a commercial product and has been developed by the Ericsson Telecommunication Company. BGw is written in C++ (approximately 100,000 lines of code) using object-oriented design, and the parallel execution has been implemented using Solaris threads. The system architecture is parallel and Ericsson therefore expected good speedup when using a multiprocessor. However, the actual scale-up of BGw was very disappointing.

In the first version of BGw, the sequential implementation of dynamic memory allocation and false memory sharing resulted in performance degradation when using more than one processor. The major reason was the standard sequential dynamic memory management combined with a large number of dynamic memory allocations. The large number of allocations is a result of how the object-oriented design was used. The number of heap allocations in BGw was reduced by introducing object-pools for commonly used object types, and by replacing heap variables with stack variables. The reduction of allocations made the application to scale linear.

However, after a few years of continuous development, BGw once again encountered scalability problems. The reason was the same as the last time. New functionality and program flows had made the object-pools inefficient. At this time, resources for updating the pools were lacking. The BGw is thus nowadays depending on a parallel allocator (Smart Heap for SMP) to scale on SMP:s.

The problem with dynamic memory is in the BGw mainly located in a certain component. This component, approximately 45,000 LOC, was extracted from the rest of the application source code. A test program design to behave identical to the original application was implemented. The test program was executed on a SUN Enterprise 10000 equipped with 8 processors. The time it took to process 5,000 CDR:s was measured.

5. Results

5.1. Synthetic program results

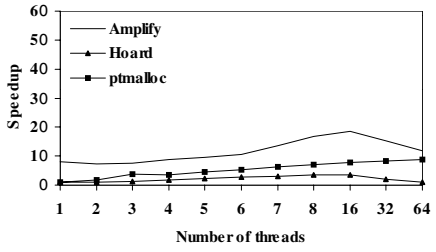


Figure 4. Speedup graph for test case 1

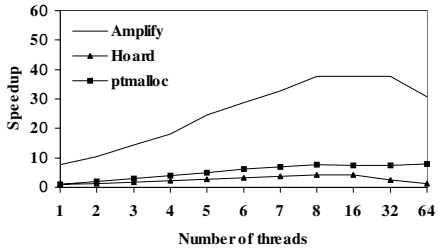


Figure 5. Speedup graph for test case 2

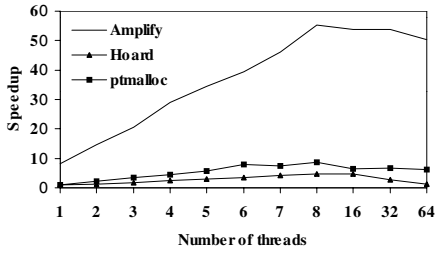


Figure 6. Speedup graph for test case 3

Figures 4, 5, and 6 show the speedup of each test case using three different methods: ptmalloc, Hoard and Amplify. As discussed above, the definition of a speedup value of 1 is the execution time using one thread and the standard Solaris heap manager. In all our tests Amplify outperforms both Hoard and ptmalloc, even when the data structure is shallow. The drop when using 2 threads compared to 1 thread in Figure 4 is caused by the fact that Amplify can operate more efficiently in a non-threaded environment. This since the pre-processor automatically removes all unnecessary locks.

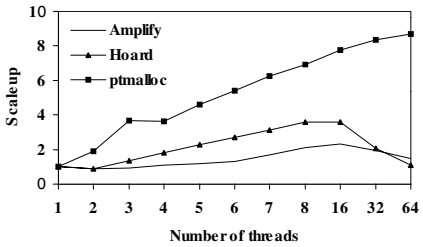


Figure 7. Scaleup graph for test case 1

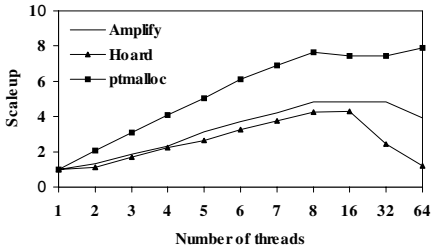


Figure 8. Scaleup graph for test case 2

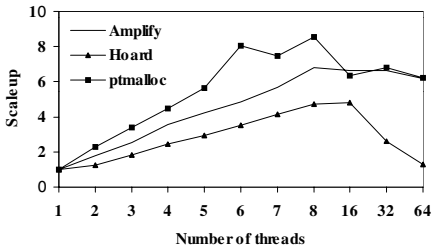


Figure 9. Scaleup graph for test case 3

Figures 7, 8 and 9 show the scaleup for our three test cases. The scaleup is defined as the speedup where the speedup value for one thread is normalized to one. Interesting to note is the fact that Amplify does not scale very well in test case 1 (see Figure 7). The main focus for Amplify is to avoid the contention caused by mutual exclusion during memory allocation. From this point of view the results from test case 1 are successful. When monitoring the locks protecting critical code regions within the pools we noticed a very low number of failed lock attempts. This result has lead us to believe that the ptmalloc approach (i.e. choose another heap/pool if a thread becomes blocked too often) no longer is a favorable solution. Using several heaps/pools has two advantages: it reduces lock contention and it improves cache hit ratio, provided that the threads that share a

heap/pool run on the same processor. The time to lock, insert/remove an object into a free list, and then unlock is very short in Amplify (compared to ptmalloc). Thus, threads will seldom or never be blocked when trying to obtain a lock. Since the frequency of failed lock attempts is the axiom for choosing another pool this can lead to a situation where there are no failed locks, but undesirable cache effects, e.g., false memory sharing. Thus, our conclusion is that the poor scalability in test case 1 is not caused by a poor locking mechanism, but rather caused by other factors, such as false memory sharing. The test programs are constructed in such a way that the memory consumption is kept low also when using Amplify. However, the intention in this test was not to stress the memory consumption. There are two potential sources for memory consumption overhead in Amplify.

The first, and most obvious, case is when there are a lot of unused objects structures in the pools. This problem can be handled by returning memory from the pools to the operating system on demand, or when the pools exceed a certain limit.

The second case is when we reuse an unnecessary large data structure for a smaller structure. The Car in Figure 1 may for instance, in a later instantiation, chose not to create an Engine object at all. In this situation the memory, used for the Engine object in the former Car, will be unused. However, the memory can be reused in later Car instantiations. The designer may, chose not to "amplify" objects that can cause this type of overhead. Another alternative is to use the same strategy as the one described for the BGw version of Amplify (see below), i.e., and not reusing unnecessary large memory blocks. Consequently, if memory consumption overhead is critical, one can take precautions by limiting the number of root objects in the pools and/or by (statically) not applying the technique on root objects that can cause serious memory overhead and/or by (dynamically) not reusing unnecessary large memory blocks. The shadow pointers also introduce memory overhead. However, each shadow pointer could be replaced with one bit, which indicates if the original pointer is logically deleted or not. If the original pointer is logically deleted it has the role of the shadow pointer, and if it is not deleted the shadow pointer has no role. This strategy would, however, make the pre-processor somewhat more complex, and we have therefore not implemented it in this prototype.

Figure 10 shows a comparison between ptmalloc, Hoard, Amplify, and a handmade structure pool when executing test case 2. Note that Hoard does not scale when the number of threads is larger then the number of processors, which is a common case for server applications (as discussed before, we use 8 processors). A possible explanation for this can be that when threads start to migrate from one processor to another Hoard runs into lock contention. The publicly available

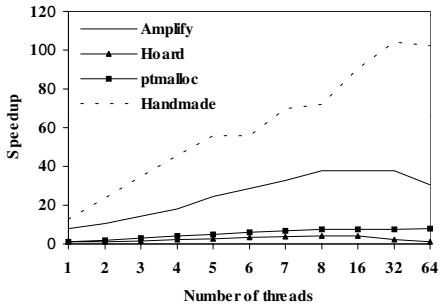


Figure 10. Speedup graph for test case 2 (including handmade structure pool)

implementation of Hoard uses a modulation based on thread id to assign threads to heaps. The handmade version could be seen as the theoretical maximum of what an optimizing pre-processor could do. As discussed before, the programmer knows things about the source code that are not available to the pre-processor. We believe, however, that it would be possible to improve the pre-processor even further by making a more sophisticated source code analysis, causing the speedup to come closer to the handmade version.

5.2. Result from tests on BGw

When Amplify was applied on BGw we encountered some (expected) problems. The first problem was that only half of the allocations in BGw are made from the application source code. The other half comes from tool libraries, such as Tools.h++. Since Amplify must be able to pre-process the code in order to insert the optimizations, Amplify's effect on BGw was limited. We can thus identify this problem to be a major drawback of our method and conclude that it will still be necessary to use some kind of parallel allocator, at least as long as we do not have access to the source code for the libraries.

The second problem encountered was that Amplify only handles objects. In BGw the large number of allocations were made for data type arrays, e.g. char[] and int[]. In order to deal with this problem we extended Amplify to also include data types. If the parent object is allocated from the object pool the following code

```
buffer = new char[length];
```

is replaced with

```
buffer = realloc(bufferShadow, length);
```

and

```
delete buffer;
```

with

```
bufferShadow = buffer;
```

where buffer is an attribute in the parent object.

Amplify works with the standard `realloc()`. *However, by overloading the* `realloc()` *function it is possible to gain performance, and also to better control the memory consumption. We implemented a* `realloc()` *of our own in order to avoid unnecessary locks and it works in the following way:*

If the allocated memory is smaller than the shadow memory but not smaller than half the shadow memory, then the shadow memory is reused. The function will guarantee that, if an allocation is made repeatedly, the maximum memory consumption is twice the normal.

We also introduced a maximum size for shadowed memory, i.e., if memory that should be shadowed is larger than the defined maximum, the memory is deleted (as normal). This function will prevent large single allocations to waste large memory chunks. We also introduced a maximum number of objects for each pool.

Consequently, for repeatedly created objects the memory consumption is limited to twice the amount of memory. However, there can still be memory overhead consumption if an object is created one or more times and after that not re-created. There are a number of solutions that reduce this problem, e.g. garbage collection. We have, however, not yet implemented any of these.

The results from the BGw show that the application is scaleable with SmartHeap. Amplify alone, i.e. without help from Smartheap, did not make BGw scalable. However, when Amplify was used in combination with SmartHeap, BGw was capable of processing CDR:s 17% faster (see Figure 11). The same result was measured if only data type arrays were shadowed or if all objects were shadowed, i.e., the shadowing of data types contributed with the major part of the allocations. The result shows that Amplify speeds up parts of the memory management where SmartHeap for SMP:s fails.

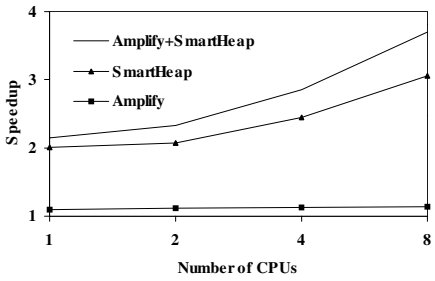


Figure 11. Speedup graph for BGw

6. Related work

The allocator designed by Doug Lea [6] is fast and efficient. However, the original version is neither thread-safe nor scalable on SMP:s. Wolfram Gloger created a thread-safe and scalable version called ptmalloc [4]. The allocator is based on a multiple number of sub-heaps. When a threads is about to make an allocation it "spins" over a number of heaps until it finds an unlocked heap. The thread will use this heap for the allocation and for allocations to come. If an allocation fails, the thread "spins" for a new heap. Since the operating system normally keeps the number of thread migrations low this implementation works fine for many applications. However, ptmalloc can when the number of threads is larger than the number of processors be unpredictable, in terms of efficiency. We have successfully used this heap implementation in our studies. For a more detailed description of ptmalloc see [11].

There is also an SMP version of SmartHeap available [7]. We have not had the opportunity to evaluate this implementation on our synthetic programs since the software is not freely accessible. During our test on BGw, however, we were given the opportunity to make some benchmark tests against Amplify.

Hoard [1] is another allocator optimized for SMPs, which we successfully used in our study. This allocator focuses on avoiding false memory sharing and blowup in memory consumption. The allocator is scalable. However, we have found that Hoard has problems when threads frequently migrate between processors. The Hoard allocator has been evaluated in a number of benchmarks. However, none of the benchmarks have the characteristics found in object-oriented software. Thus, to run these benchmarks on Amplify would be pointless.

LKmalloc [5], developed by Larsson and Krishnan, is yet another parallel memory allocator. (Not investigated by us). It focuses on being fast and scalable in both traditional applications and long-running server applications, which are executed on multiprocessor hardware.

Before ptmalloc, Hoard, LKmalloc etc., which were presented relatively recently, there is not much work available on parallel memory allocators. A survey on prior work in dynamic memory management can be found in Larson and Krishan's paper "Memory Allocation for Long-Running Server Applications " [5] *There are also reports on hardware support for dynamic memory management [2]. The temporal locality characteristics exploited by Amplify are not specific to applications written in C++, but rather the behavior of object-oriented applications using frameworks and design patterns. Thus, using Amplify together with other object-oriented languages than C++ would most likely be beneficial. We are currently evaluating Amplify for Java.*

7. Conclusion

The traditional solution to performance problems caused by heavy use of dynamic memory is to optimize the C library memory allocation routines. However, attacking these problems on source code level (i.e. modify the application code) makes it possible to achieve more efficient, application specific, memory management.

Optimizing the application by integrating customized structure pools into the source code has shown to be very beneficial in terms of performance. It is, however, a manual procedure and thus carries maintainability drawbacks, which makes it less attractive. By automating the procedure of adding these structure pools, most of these drawbacks are eliminated, making it possible to combine maintainability and performance. We have implemented a pre-processor based method, named Amplify, which in a completely automated procedure integrates structure pools into applications written in C++. By exploiting temporal locality, often characterizing applications developed using object-oriented techniques, Amplify can make very efficient optimizations resulting in low overhead. Amplify has shown to be up to six times more efficient in speeding up synthetic applications, compared to the available, state-of-the-art, C library allocators we have tested. It is also interesting to note that Amplify increases the performance of sequential as well as parallel programs.

Our tests on BGw verify the existents of temporal locality in a real commercial application. However, Amplify's possibility to make optimizations was reduced by the limited access to application source code. Nevertheless, a modified version of Amplify improved the performance of BGw with 17%. It was interesting to note that the relative performance increase due to Amplify was (more or less) the same with and without a parallel heap manager, i.e., the performance improvements of Amplify seem to be orthogonal to the performance improvements of parallel heap managers.

Neither the BGw nor the synthetic test programs suffered from the increased memory consumption. Amplify can, however, in some situations suffer from this problem. We have therefor discussed a number of techniques to limit the memory consumption overhead of Amplify.

Sine all source code may not be available, the Amplify method may have to be complemented with parallel heap managers to be useful in a realistic situation. However, the results show that Amplify has high potential and that it was useful on a commercial product.

8. References

[1] *E. Berger K. McKinley, R. Blumofe, and P. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications", in Proc. of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), Cambridge, MA, November 2000.*

[2] *J.M. Chang, W. Srisa-an, C.D. Lo, and E.F. Gehringer, "Hardware support for dynamic memory management", in Proc. of the Workshop for Solving the Memory-Wall Problem, at the 27th International Symposium on Computer Architecture, Vancouver, BC, June 2000.*

[3] *D. Detlefs, A. Dosser, and B. Zorn, "Memory allocation costs in large C and C++ programs", Software – Practice and Experience, pp. 527-542, June 1994.*

[4] *W. Gloger, "Dynamic memory allocator implementations in Linux", http://www.dent.med. uni-muenchen.de/~wmglo/malloc-slides.html (site visited January 2001).*

[5] *P. Larson and M. Krishan, "Memory Allocation for Long-Running Server Applications", in Proc. of the International Symposium on Memory Management, ISMM '98, Vancouver, British Columbia, Canada, October, 1998.*

[6] *D. Lea, "A memory allocator", http://g.oswego.edu /dl/html/malloc.html (site visited January 2001).*

[7] *MicroQuill Software Puplishing, Inc., "SmartHeap for SMP", http://www.microquill.com/smp, (site visited January 2001).*

[8] *MicroQuill Software Publishing, Inc., "SmartHeap – Programmer's Guide", March 1999.*

[9] *D. Häggander, PO. Bengtsson, J. Bosch, and L. Lundberg, "Maintainability Myth Causes Performance Problems in Parallel Applications", in Proc. of the 3rd International Conf. on Software Engineering and Applications, Scottsdale, USA, pp. 288-294, October 1999.*

[10] *D. Häggander and L. Lundberg, "Attacking the Dynamic Memory Problem for SMPs", in Proc. of the 13th International Conf. on Parallel and Distributed Computing Systems, Las Vegas, Nevada, USA, August 8-10 2000.*

[11] *D. Häggander and L. Lundberg, "Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor", in Proc. of the 27th International Conf. on Parallel Processing, Minneapolis, USA, August 1998.*

[12] *B. Stroustrup, "The C++ Programming Language", Addison-Wesley, 1997.*

Paper VIII

Recovery Schemes for High Availability and High Performance Cluster Computing

Lars Lundberg and Daniel Häggander
Research report 2001:06
ISBN:1103-1581

VIII

Recovery Schemes for High Availability and High Performance Cluster Computing

Lars Lundberg and Daniel Häggander

Department of Software Engineering and Computer Science, Blekinge Institute of Technology,
S-372 25 Ronneby, Sweden
lars.lundberg@bth.se, daniel.haggander@bth.se

Abstract. Clusters and distributed systems offer two important advantages, viz. fault tolerance and high performance through load sharing. When all computers are up and running, we would like the load to be evenly distributed among the computers. When one or more computers break down the load on these computers must be redistributed to other computers in the cluster. The redistribution is determined by the recovery scheme. The recovery scheme should keep the load as evenly distributed as possible even when the most unfavorable combinations of computers break down, i.e. we want to optimize the worst-case behavior. In this paper we define recovery schemes, which are optimal for a number of important cases.

1 Introduction

One way of obtaining high availability and fault tolerance is to distribute an application on a cluster or distributed system. If a computer breaks down, the functions performed by that computer will be handled by some other computer in the cluster. A lot of cluster vendors support this kind of fault recovery, e.g. Sun Cluster [9], MC/ServiceGuard (HP) [5], TruCluster (DEC) [10], HACMP (IBM) [1], and MSCS (Microsoft) [6,11].

Consider a cluster with two computers. A common way to obtain high availability is to have one active (primary) computer that executes the application under normal conditions and one secondary computer that executes the application when the primary computer breaks down. This approach can be extended to systems with more than two computers. In that case there may be a third computer that executes the application when the primary and secondary computers are both down, and so on. The order in which the computers are used is referred to as the *recovery order*.

An advantage of using clusters, besides fault tolerance, is load sharing between the computers. When all computers are up and running, we would like the load to be evenly distributed. The load on some computers will, however, increase when one or more computers are down, but even under these conditions we would still like to distribute the load as evenly as possible.

Consequently, clusters offer two important advantages: fault tolerance and high performance through load sharing.

The distribution of the load in case of a fault, i.e. a computer going down, is decided by the recovery orders of the processes running on the faulty computer. The set of all recovery orders is referred to as

the *recovery scheme*, i.e. the load distribution in case of one or more faults is determined by the recovery scheme.

We have previously defined recovery schemes that are optimal for some cases [4]. In this paper we define new recovery schemes which are better in the sense that they are optimal for a larger number of cases.

2 Problem Definition

We consider applications that are executed on a cluster consisting of n identical computers. There is one process running on each computer, e.g. n identical copies of the same application (in Section 5 we extend our results to cases when there are more than one process on each computer). The work is evenly split between these n processes.

There is a recovery order list associated with each process. This list determines where the process should be restarted if the current computer breaks down. Fig. 1 shows an example of such a system for $n = 4$. We assume that processes are immediately moved back as soon as a computer comes back up again. In most cluster systems this can be configured by the user [5,9,10], i.e. in some cases one may not want immediate and automatic relocation of processes when a faulty computer comes back up again.

The left side of the figure shows the system under normal conditions. In this case, there is one process executing on each computer. The recovery order lists are also shown; one list for each process. The set of all these recovery order lists is referred to as the recovery scheme.

The right side of Fig. 1 shows the scenario when computer zero breaks down. The recovery order list for process zero, which executes on computer zero, shows that process zero should be restarted on computer one when computer zero breaks down. Consequently, there will be two processes executing on computer one, i.e. processes one and zero.

If computer one also breaks down, process zero will be restarted on computer two, which is the second computer in the recovery order list. The first computer in the recovery order list of process one is computer zero. However, since computer zero is down, process one will be restarted on computer three. Consequently, if computers zero and one are down, there are two processes on computer two (processes zero and two) and two processes on computer three (processes one and three).

If computers zero and one break down the maximum load on each of the remaining computers is twice the load compared to normal conditions. This is a good result, since the load is as evenly distributed as it can be. However, if computers zero and two break down, there will be three processes on computer one (processes zero, one and two), and only one process on computer three. In this case the maximum load on the most heavily loaded computer is three times the normal load. This means that, for the recovery scheme in Fig. 1, the combination of computers zero and two being down is more unfavorable than the combination of computers zero and one being down.

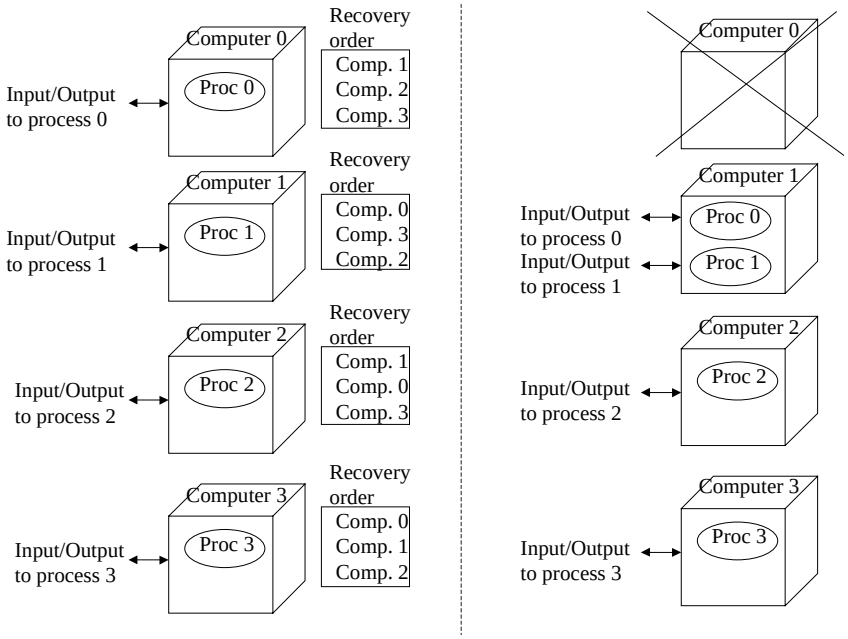


Fig. 1: An application executing on a cluster with four computers. The left side of the figure shows the system under normal conditions, and the right side shows the system when computer zero is down.

Our results are also valid when there are *n* external systems feeding data into the cluster, e.g. one telecommunication switching center feeding data into each of the *n* computers in the cluster (see Fig. 2). If a computer breaks down, the switching center must send its data to some other computer in the cluster, i.e. there has to be a "recovery order list" associated with each switching center. The fail-over order can alternatively be handled at the communication protocol level, e.g. IP takeover [8]. In that case, redirecting the communication to another computer is transparent to the switching center.

Many cluster vendors offer not only the user defined recovery order lists considered here, but also dynamic load balancing schemes in case of a node going down. The unpredictable worst-case behavior and relatively long switch-over delays of such dynamic schemes may, however, be unattractive in many real-time applications; it is often easier to control the worst-case behavior by using static recovery schemes. Also, external systems cannot use dynamic load balancing schemes when they need to select a new node when the primary destination for their output is not responding (see Fig. 2). Consequently, the results presented here are very relevant for systems where each node has its own network address. This is the case in distributed systems and it can also be the case in cluster systems (depending on how you define the word "cluster" [8]).

In Section 5 we discuss the case when there are a number of processes on each computer, and these processes can be redistributed independently of each other. However, until then we assume that the work performed by each of the *n* computers must be moved as one atomic unit. Examples of this are systems where all the work performed by a computer is generated from one external system (see Fig. 2) or when all the work is performed by one process (see Fig. 1).

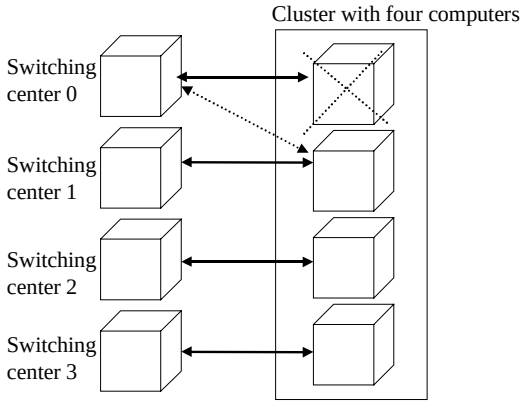


Fig. 2: Load redistribution in a telecommunication system when one computer in the cluster crashes.

We now introduce some notations. Consider a cluster with n computers. Let $L(n,x,\{c_0,\ldots,c_{x-1}\},RS)$ ($n > x$) denote the load on the most heavily loaded computer when computers $c_0,\ldots,c_{x-1}$ are down and when using a recovery scheme RS. $L(4,2,\{0,1\},RS) = 2$ for the example in Fig. 1, whereas $L(4,2,\{0,2\},RS) = 3$. However, if we change the recovery order for process two to (Comp. 3, Comp. 0, Comp. 1) and the recovery order of process three to (Comp. 2, Comp. 1, Comp. 0), we obtain a new recovery scheme RS' such that $L(4,2,\{0,2\},RS') = 2$.

Let $L(n,x,RS) = max\ L(n,x,\{c_0,\ldots,c_{x-1}\},RS)$ for all vectors $\{c_0,\ldots,c_{x-1}\}$, i.e. for all possible combinations of x computers being down. Consequently, $L(n,x,RS)$ defines the worst-case behavior when using recovery scheme RS. Worst-case behavior is particularly important in real-time systems. For the recovery scheme in Fig. 1, $L(4,2,RS) = 3$.

The recovery scheme should distribute the load as evenly as possible for any number of failing computers x. Small values of x are (hopefully) more common than large values of x. We introduce the notation $V(L(n,RS)) = \{L(n,1,RS),\ldots,L(n,n\text{-}1,RS)\}$ (N.B. $V(\ldots)$ is a vector of length n-1), and say that $V(L(n,RS))$ is smaller than $V(L(n,RS'))$ if and only if $L(n,y,RS) < L(n,y,RS')$ for some $y < n$, and $L(n,z,RS) = L(n,z,RS')$ for all $z < y$ (if $y = 1$, it is enough that $L(n,y,RS) < L(n,y,RS')$).

Let $VL = min\ V(L(n,RS))$ for all recovery schemes RS (VL is a vector of length n-1).

We have previously defined a lower bound B on VL, i.e. $B \leq VL$ (B is a vector of length n-1), and for $n \leq 11$, we defined a recovery scheme RS such that $V(L(n,RS)) = B$ [4]. We refer to such recovery schemes as *optimal recovery schemes*. We have also defined recovery schemes that are optimal when at most $\lfloor {}^2\log n \rfloor$ computers break down. In order to make this paper self-contained, the previous results are summarized in Section 3. In Section 4 we improve the previous results by defining recovery schemes that are optimal also when a significantly larger number of computers break down.

3 The Bound *B*

In order to obtain *B* we start by defining *bound vectors*: A bound vector of length $l = 2+3+4+\ldots+k$ has the following structure: {2,2,3,3,3,4,4,4,4,5,5,5,5,5,…,*k*,*k*,…,*k*} (the vector ends with *k* entries with value *k*). For instance, a bound vector of length 2 + 3 + 4 + 5 + 6 = 20 looks like this: {2,2,3,3,3,4,4,4,4,5,5,5,5,5,6,6,6,6,6,6}. We now extend the definition of bound vectors to include vectors of any length by taking any arbitrary bound vector and truncating it to the designated length, e.g. a bound vector of length 18 looks like this: {2,2,3,3,3,3,4,4,4,4,5,5,5,5,5,6,6,6,6}.

Theorem 1: *VL* cannot be smaller than a bound vector of length *n*-1 (the meaning of *VL* being "smaller than" than some other vector was defined in the previous section).

Proof: We will use proof by contradiction.

Assume that *VL* is smaller than a bound vector of length *n*–1. In that case there is a value *y* such that the first *y*-1 values in the bound vector are the same as the first *y*-1 values in *VL* and that *VL*(*y*) is smaller than entry *y* in the bound vector.

Obviously, $VL(y\text{-}1) \leq VL(y)$. Moreover, the bound vector only increases at entries $x_k = 2+3+4+\ldots+k\text{-}(k\text{-}1)$, for *k*=2, 3, 4,…, i.e. $x_2 = 2\text{-}(2\text{-}1) = 1$, $x_3 = 2+3\text{-}(3\text{-}1) = 3$, $x_4 = 2+3+4\text{-}(4\text{-}1) = 6$, …. Consequently, the difference between *VL* and the bound vector must occur at an entry x_k, i.e. $y = x_k$ This means that, $VL(x_k) < k$ for some x_k.

If $k = 2$ ($x_k = 1$), one computer is down, and *VL*(1) is obviously equal to 2, i.e. *VL*(1) is equal to entry one in the bound vector.

Since *VL*(2) = 2 (remember that we assume that *VL* is smaller than a bound vector of length *n*-1), we know that no two recovery order lists starts with the same computer. (If two lists corresponding to processes c_1 and c_2 start with the same computer c_3, there will be three processes on computer c_3 if computers c_1 and c_2 break down.)

If $k = 3$ ($x_k = 3$), three computers are down. Since all recovery order lists start with different computers, there must be at least one pair of recovery order lists l_1 (corresponding to process c_1) and l_2 (corresponding to process c_2) such that computer c_3 is first l_1 and c_3 is the second alternative in l_2. (Since *VL*(2) = 2 we know that c_1 is not the first alternative in l_2.) If computers c_1, c_2 and the first alternative in l_2 break down, there will be three processes on c_3. Consequently, *VL*(3) = 3.

Since *VL*(5) = 3, we know that no two recovery order lists have the same computer as the second alternative. If two lists corresponding to c_1 and c_2 have the same computer c_3 as the second alternative, then there will be four processes on computer c_3 if computers c_1, c_2, the first alternative in l_1 and l_2 and the computer with c_3 as the first alternative break down.

If $k = 4$ ($x_k = 6$), six computers are down. Since all recovery order lists start with different computers and have different computers as the second alternative, there must be at least one triplet of recovery order lists l_1, l_2 and such that computer c_4 is first l_1 and the second alternative in l_2 and the third alternative in l_3. If computers c_1, c_2, c_3, the first alternative in l_2, the first and the second alternative in l_3 break down, there will be four processes on c_4. Consequently, *VL*(6) = 4.

Since *B*(9) = 4, we know that no two recovery order lists have the same computer as the third alternative. If two lists corresponding to c_1 and c_2 have the same computer c_3 as the third alternative, there will be four processes on computer c_3 if computers c_1, c_2, the first alternative in l_1 and l_2 and the computer with c_3 as the first alternative break down.

This procedure can be repeated for all *k*. From this we can conclude that there is no *k* such that, $VL(x_k) < k$ for some x_k, where $x_k = 2+3+4+\ldots+k\text{-}(k\text{-}1)$. Consequently, *VL* cannot be smaller than a bound vector of length *n*-1. •

Theorem 2: $VL(i) \geq \lceil n/(n\text{-}i) \rceil$.

Proof: If i computers are down, there are i processes which must be allocated to the remaining n-i computers. The best one can hope for is obviously to obtain a load of $\lceil n/(n\text{-}i) \rceil$ processes on the most heavily loaded computer. •

Based on Theorems 1 and 2, we define $B(i)$ = max(entry i in the bound vector, $\lceil n/(n\text{-}i) \rceil$).

By using exhaustive testing, we have previously defined a recovery scheme RS such that $V(L(n,RS)) = VL = B$ for $n \leq 11$, i.e. we have defined an optimal scheme, and thus also showed that B is an optimal bound when $n \leq 11$. The exponential complexity of the problem makes it infeasible to calculate optimal recovery schemes for a larger set of computers. For arbitrary n we have, however, previously defined recovery schemes that are optimal when at most $\lfloor {}^2\log n \rfloor$ computers break down. In the next section we improve these results by defining recovery schemes that are optimal also when a significantly larger number of computers break down.

4 Optimal Recovery Schemes

The first recovery scheme that comes to mind is R_i = {(i+1) mod n, (i+2) mod n, (i+3) mod n,…, (i+n-1) mod n}, where R_i denotes the recovery order list for process i. If n = 4 we would get the following four lists: R_0 = {1,2,3}, R_1 = {2,3,0}, R_2= {3,0,1}, and R_3 = {0,1,2} (R_0 denotes the recovery order list for process zero and R_1 denotes the recovery order list for process one etc.).

The problem with this scheme is that the maximum number of processes may exceed the bound B. Consequently, the intuitive scheme R_i = {(i+1) mod n, (i+2) mod n, (i+3) mod n,…, (i+n-1) mod n} is not optimal for n = 4 (in that case $B(2)$ = 2 but there are three processes on computer two if computers zero and one are down). For n = 8 we will obtain six processes on computer five if computers zero to four are down. However, when n = 8 we know that $B(5)$ = 3. Consequently, the worst-case behavior of the intuitive scheme is twice as bad as an optimal scheme for clusters with eight computers. The difference between the intuitive scheme and an optimal scheme can be arbitrarily large for large values of n.

As discussed above, the combinatorial complexity explosion of the problem makes it very difficult, or even impossible, to find optimal recover schemes for large n. We have, however, previously shown that recovery lists (for process zero) that start with: R_0 = {1, 2, 4, 8,…, $2^{\lfloor {}^2\log n \rfloor}$,…} are optimal as long as at most $\lfloor {}^2\log n \rfloor$ computers break down. The recovery list for process i is obtained from R_0 by using the following relation: $R_i(x) = (R_0(x) + i) \bmod n$, where $R_i(x)$ denotes element number x in list R_i.

We will now define a recovery scheme called *the improved single-process recovery scheme*. We call it single-process because we consider the case where there is only one process on each computer when all computers are up and running, and we call it improved because we can now guarantee optimal behavior also when more than $\lfloor {}^2\log n \rfloor$ computers break down. The improved single-process recovery scheme is defined for any arbitrary value of n. We will show that this recovery scheme is optimal for a number of important cases. In order to define the improved single-process recovery scheme, we start by defining R_0, i.e. the recovery list for process zero:

$\mathbf{N} = \{1,2,3,\ldots\}$, $f(i,k,\{r(1),\ldots,r(x)\}) = \sum_{l=k-i}^{k-1} r(l)$.

$r(x) = \min\{j \in \mathbf{N}\}$, such that $\{f(i,k,\{r(1),\ldots, r(x\text{-}1)\}) : i = 1,\ldots,x\text{-}1;\ k = i\text{+}1,\ldots,x\} \cap$ $\{f(i,k,\{r(1),\ldots, r(x)\}) : i = 1,\ldots,x;\ k = x\text{+}1\ \} = \varnothing$.

If $\sum_{l=1}^{x} r(l) < n$, then $R_0(x) = \sum_{l=1}^{x} r(l)$, else $R_0(x) = \min\{j \in \mathbf{N} - \{R_0(1),\ldots, R_0(x\text{-}1)\}\}$

Again, we obtain all other recovery lists from R_0, by using $R_i(x) = (R_0(x) + i) \bmod n$.

From the definition above we see that for large values of n (i.e. $n > 289$) first 16 elements in R_0 for the improved single-process recovery scheme are: 1, 3, 7, 12, 20, 30, 44, 65, 80, 96, 122, 147, 181, 203, 251, 289. For $n = 16$, R_0= {1, 3, 7, 12, 2, 4, 5, 6, 8, 9, 10, 11, 13,14, 15}.

In the definition of R_0 we use the *step length* vector r, where $r(x)$ (i.e. element number x in this vector) denotes the difference between $R_0(x\text{-}1)$ and $R_0(x)$ when $\sum_{l=1}^{x} r(l) < n$ (we define $R_0(0) = 0$).

Consider a system with $n \geq 8$. In that case $r(1) = 1$, $r(2) = 2$ and $r(3) = 4$. Fig. 3 illustrates this example by showing what happens if:

(a) All computers are up and running.
(b) Computer zero first breaks down, i.e., in this case process zero ends up on computer one ($r(1) = 1$).
(c) Computer one then breaks down, i.e., in this case process zero ends up on computer two ($r(2) = 2$) and process one end up on computer one ($r(1) = 1$).
(d) Computer two then breaks down, i.e., in this case process zero ends up on computer seven ($r(3) = 4$) and process two ends up on computer three ($r(1) = 1$).

In general, process x ($0 \leq x \leq n$) will in the x:th step end up on computer $(x + \sum_{i=1}^{x} r(i)) \bmod n$.

Table 1 below shows the first 10 entries in the step length vector and in the recovery list for process zero, assuming that $n > 96$.

Entry no.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Step length r	1	2	4	5	8	10	14	21	15	16
R_0	1	3	7	12	20	30	44	65	80	96

Table 1: The relation between the step length and the recovery order lists.

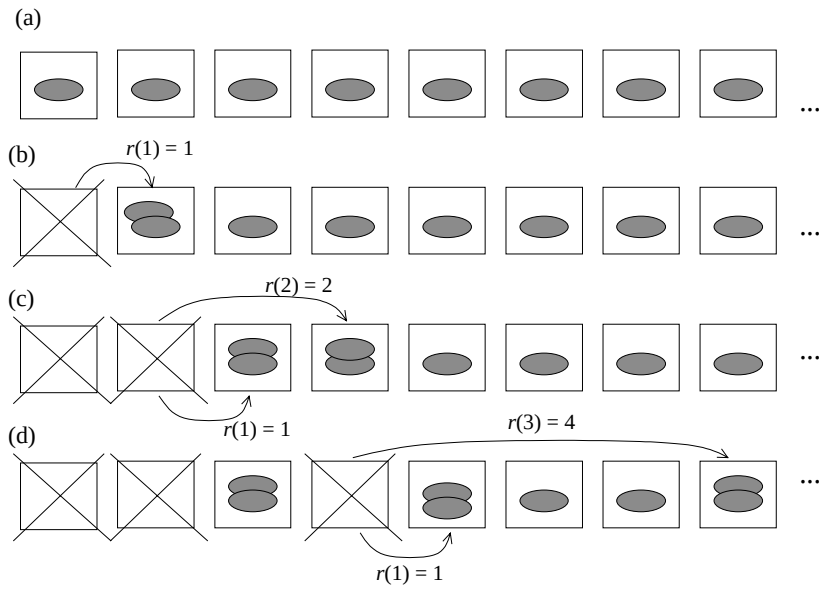


Fig. 3: An example showing the start of the step length vector for the improved single-process recovery scheme, and $n \geq 8$.

We now define a new vector of length x. This vector is called the reduced step length vector and it consists of the first x entries in the step length vector r, where $x = \max(i)$, such that $R_0(i) < n$. Let r' denote the reduced step length vector. From the definition of the improved single-process recovery scheme we know that all sums of sub-sequences in r' are unique. Table 2 illustrates this for $n = 97$, resulting in a reduced step length vector of length 10 ($10 = x = \max(i)$, such that $R_0(i) < n = 97$).

Entry no. (*i*)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
R_0	1	3	7	12	20	30	44	65	80	96
Reduced step length vector r'	1	2	4	5	8	10	14	21	15	16
Note that there are only unique values in the triangular matrix below.										
Sum of subsequences of length 1 starting in position *i*.	1	2	4	5	8	10	14	21	15	16
Sum of subsequences of length 2 starting in position *i*.	3	6	9	13	18	24	35	36	31	
Sum of subsequences of length 3 starting in position *i*.	7	11	17	23	32	45	50	52		
Sum of subsequences of length 4 starting in position *i*.	12	19	27	37	53	60	66			
Sum of subsequences of length 5 starting in position *i*.	20	29	41	58	68	76				
Sum of subsequences of length 6 starting in position *i*.	30	43	62	73	84					
Sum of subsequences of length 7 starting in position *i*.	44	64	77	89						
Sum of subsequences of length 8 starting in position *i*.	65	79	93							
Sum of subsequences of length 9 starting in position *i*.	80	95								
Sum of subsequences of length 10 starting in position *i*.	96									

Table 2: All sums of subsequences of the reduced step length vector for $n = 97$.

Theorem 3: The improved single-process recovery scheme is optimal as long as x computers or less have crashed, where $x = \max(i)$, such that $R_0(i) < n$.

Proof: Let y ($0 \le y \le n$) be the heaviest loaded computer when x computers have crashed, where $x = \max(i)$, such that $R_0(i) < n$. The lists below show the x possible sequences of computers that need to be down in order for an extra process to end up on computer y, i.e. if computer $(y\text{-}r(1)+n) \bmod n$ is down one extra process will end up on computer y, if computers $(y\text{-}r(1)\text{-}r(2)+n) \bmod n$ and $(y\text{-}r(2)+n) \bmod n$ are down another extra process will end up on computer y, and so on.

1. $(y - \sum_{i=1}^{1} r'(i) + n) \bmod n$
2. $(y - \sum_{i=1}^{2} r'(i) + n) \bmod n$, $(y - \sum_{i=2}^{2} r'(i) + n) \bmod n$
3. $(y - \sum_{i=1}^{3} r'(i) + n) \bmod n$, $(y - \sum_{i=2}^{3} r'(i) + n) \bmod n$, $(y - \sum_{i=3}^{3} r'(i) + n) \bmod n$

…

x. $(y - \sum_{i=1}^{x} r'(i) + n) \bmod n$, $(y - \sum_{i=2}^{x} r'(i) + n) \bmod n$,…,$(y - \sum_{i=x}^{x} r'(i) + n) \bmod n$

From the previous discussion we see that no computer is included into more than one of the lists one to x. By looking at the proof of Theorem 1 we see that the scheme is optimal as long as no computer is included in two such lists. The theorem follows. •

As discussed above, we have previously defined a recovery scheme (the single-process recovery scheme) which is optimal as long as at most $\lfloor {}^2\log n \rfloor$ computers break down. In Fig. 4 we compare the performance of the improved single-process recovery scheme with the performance of the (old) single-process recovery scheme as a function of the number of computers. The performance is defined as the number of crashes that we can handle while still guaranteeing an optimal load distribution. Fig. 4 shows that the behavior of the improved single-process recovery scheme is significantly better for large values of n, e.g., for $n = 100$ the improved scheme guarantees optimal behavior even if 10 computers break down while the old version can only guarantee optimal behavior as long as at most 6 computers break down. Fig. 4 shows that for $n = 1000$ we can guarantee optimal behavior for up to 26 crashed computers using the improved single-process recovery scheme, whereas the (old) single-process recovery scheme only guarantees optimal behavior up to 9 crashed computers.

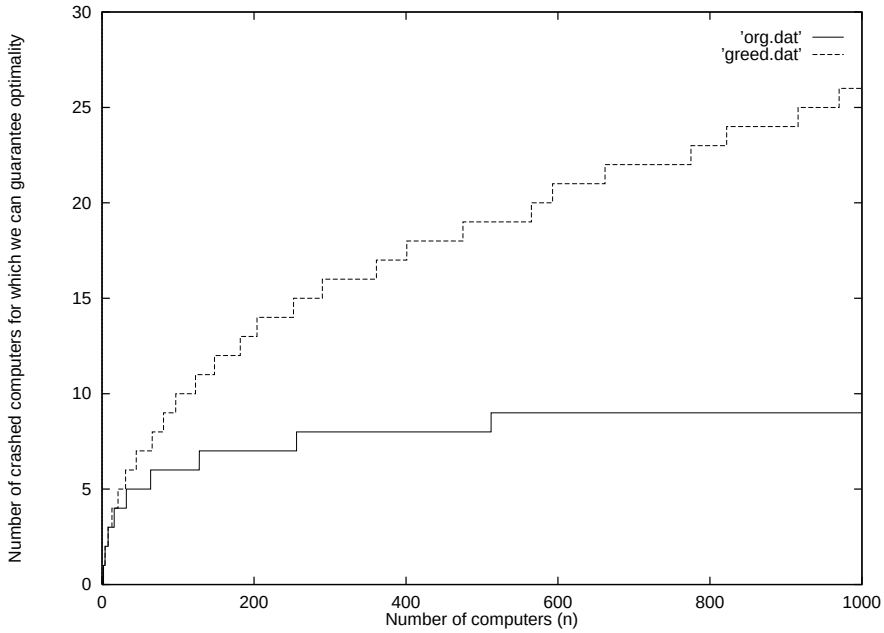


Fig. 4: The "performance" difference between the improved version of the single-process recovery scheme ('greed.dat') and the old version of the single-process recovery scheme ('org.dat').

To recapitulate our problem formulation: *Given a certain number of computers (n) we want to find a recovery scheme that can guarantee optimal worst-case load distribution when at most x computers are down, and we are interested in the schemes that have as large x as possible.*

By looking at the proof of Theorem 3, we see that the problem can be reformulated as: *Given a number of computers (n) we want to find the longest sequence of positive integers such that the sum of the sequence is smaller than or equal to n and such that all sums of subsequences (including subsequences of length one) are unique.*

The sequence of positive integers is the reduced step length vector. The improved single-process recovery scheme is a greedy way of generating the reduced step length vector (hence the name 'greed,dat' in Fig 4). By greedy we mean that we generate the numbers in the sequence one by one and select the smallest possible number in each step. It turns out that this greedy approach is not the best strategy for all values of *n*. Fig. 5 compares the improved single-process recovery scheme ('greed.dat') with a greedy approach where we exclude the value one (which is the first element in the reduced step length vector). The reduced step length vector when we exclude value one is (n = 107): 2, 3, 4, 6, 8, 11, 16, 12, 24, 20. Consequently, excluding value one affects the entire sequence. Fig. 5 shows that for some values of *n*, the improved single-process recovery scheme is better than the sequences obtained when we exclude the value one ('greed-1.dat'), whereas the sequence where one is excluded is better for other *n*, and for some *n* it does not matter.

Fig. 6 shows a similar comparison, where 'greed-5.dat' denotes the sequence obtained when excluding all values smaller than or equal to five. The reduced step length vector when we exclude all values smaller than or equal to five is (n = 122): 6, 7, 8, 9, 10, 12, 11, 14, 18, 26. Fig. 6 shows that

'greed-5.dat' is (as one might expect) not as good as the improved single-process recovery scheme for small values of *n*. However, for larger *n* the difference is marginal and in one interval 'greed-5.dat' actually outperforms 'greed.dat'.

Our conclusion from this is that it seems to be NP-hard to find the optimal sequence, i.e., the longest sequence, given a certain *n*. As for other NP-hard problems, one can use heuristic search methods in order to find good solutions. The limited experiments that we have done this far (see Fig. 5 and Fig. 6) indicates that the performance difference between different greedy approaches seems to be rather small.

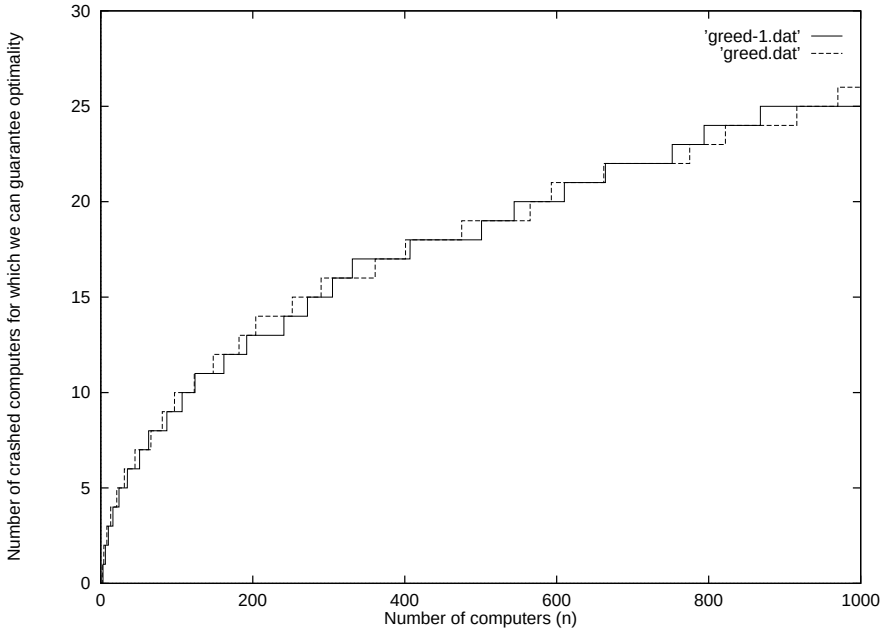


Fig. 5: The "performance" difference between the improved version of the single-process recovery scheme ('greed.dat') and a version where we have we have excluded the value 1 as a valid step length ('greed-1.dat').

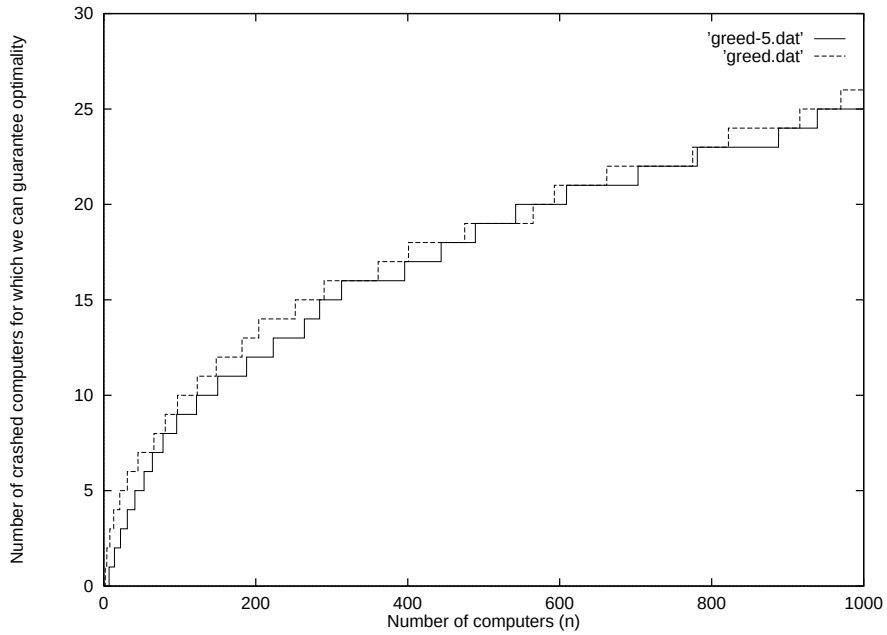


Fig. 6: The "performance" difference between the improved version of the single-process recovery scheme ('greed.dat') and a version where we have we have excluded all values smaller than or equal to five as valid step lengths ('greed-5.dat').

5 Non-Atomic Loads

Hitherto, we have considered the case when all work performed by a computer must be moved as one atomic unit. Obviously, there could be more than one process on each computer and these processes may in some cases be redistributed independently of each other, and in that case there is one recovery order list for each process.

Previous studies show that, if there is a large number of processes on each computer, the problem of finding optimal recovery schemes becomes less important [4]. The reason for this is that the intuitive solutions become better. Consequently, the importance of obtaining optimal recovery schemes is largest when p is small compared to n. In the previous sections we have obtained results for $p = 1$. The techniques used for obtaining those results are also useful when we consider $p > 1$.

We now define Bound vectors of type p. The Bound vectors discussed in Section 3 were of type one, i.e. $p = 1$. A bound vector of type p is obtained in the following way (the first entry in the Bound vector is entry 1):

1. $t = p$; $i = 1$; $j = 1$; $r = 1$; $v = 1$
2. If $r = 1$ then $t = t+1$; $r = j$; $v = v+1$ else $r = r-1$
3. Let entry i in the Bound vector of type p have the value t/p
4. $i = i+1$
5. If $v = p$ and $r = 1$ then $j = j+1$; $v = 0$
6. Go to 2 until the bound vector has the desired length.

A Bound vector of type two looks like this (note that the load on each computer is normalized to one when all computers are up and running):

{3/2, 4/2, 4/2, 5/2, 5/2, 6/2, 6/2, 6/2, 7/2, 7/2, 7/2, 8/2,…

A Bound vector of type three looks like this:

{4/3, 5/3, 6/3, 6/3, 7/3, 7/3, 8/3, 8/3, 9/3, 9/3, 9/3, 10/3, 10/3,…

We now extend the definition of *VL* to cover not only the case when $p = 1$, but also the case when $p > 1$. Consequently, $VL(x)$ $(1 \leq x < n)$ denotes the maximum number of processes divided with p (in order to normalize the original load to one) on the most heavily loaded computer when using an optimal recovery scheme and when x computers are down, not only when $p = 1$ but also when $p > 1$.

Theorem 4: *VL* cannot be smaller than a bound vector of type p of length n-1 (the meaning of *VL* being "smaller than" than some other vector was defined in Section 2).

Proof: By using the same proof technique as for Theorem 1 we see that there must be at least p processes that has the same computer first in their recovery order list. This is the reason why the first p entries in a Bound vector of type p increase with one. We also see that there must be at least p processes with the same computer as the second alternative in their recovery lists. This is the reason why a Bound vector of type p increases with one for every second entry from entry p to $3p$. There must also be at least p processes that have the same computer as the third alternative in their recovery lists. This is the reason why a Bound vector of type p increases with one every third entry from entry $3p$ to $6p$, and so on. •

Based on p and n we now define B in the following way: $B(i)$ = max(entry i in the bound vector of type p, $\lceil pn/(n-i) \rceil / p$). When $p = 2$ and $n = 12$ we get: B = {3/2, 4/2, 4/2, 5/2, 5/2, 6/2, 6/2, 6/2, 7/2, 12/2, 24/2}.

For $p \leq n$, we have previously defined a recovery scheme - the p-process recovery scheme - that is optimal as long as at most $\lfloor {}^2\log \lfloor n/p \rfloor \rfloor$ computers break down. We call it p-process because we consider the case where there are p processes on each computer when all computers are up and running.

We now define *the improved p-process recovery scheme*. The scheme is defined when $p \leq n$ (remember that optimal recovery schemes are most important when p is small compared to n). We will show that this recovery scheme is optimal also when significantly more than $\lfloor {}^2\log \lfloor n/p \rfloor \rfloor$ computers break down. In fact, the results presented in Section 4 correspond to the case when $p = 1$.

Let $R_{i,j}$ denotes the recovery order list for process j $(0 \leq j < p)$ on computer i $(0 \leq i < n)$. We will now define $R_{i,j}$ based on R_0, i.e. the recovery scheme for process zero in the single-process recovery scheme defined in the previous section.

If $R_0(k) + j\lfloor n/p \rfloor < n$, then $R_{i,j}(k) = (R_0(k) + i + j\lfloor n/p \rfloor) \bmod n$, else $R_{i,j}(k) = (R_0(k) + i + j\lfloor n/p \rfloor + 1) \bmod n$, where $R_{i,j}(k)$ denotes integer number k in the lists for process j on computer i. The table below shows the recovery lists when $n = 5$ and $p = 2$.

$R_{0,0}$	1,3,2,4
$R_{0,1}$	3,1,4,2
$R_{1,0}$	2,4,3,0
$R_{1,1}$	4,2,0,3
$R_{2,0}$	3,0,4,1
$R_{2,1}$	0,3,1,4
$R_{3,0}$	4,1,0,2
$R_{3,1}$	1,4,2,0
$R_{4,0}$	0,2,1,3
$R_{4,1}$	2,0,3,1

Theorem 5: Let RS_n denote the improved p-process recovery scheme for n computers with p processes on each computer, and let B be a Bound vector of type p. In that case the first entry in $V(L(n,RS_n))$ is equal to $B(1)$, the second entry in $V(L(n,RS_n))$ is equal to $B(2)$, the third entry in $V(L(n,RS_n))$ is equal to $B(3)$, …, the x:th entry in $V(L(n,RS_n))$ is equal to $B(x)$, where $x = \max(i)$, such that $R_0(i) < \lfloor n/p \rfloor$.

Proof: The proof is similar to the proof of Theorem 3.

Let y ($0 \le y \le n$) be the heaviest loaded computer when x computers have crashed, where $x = \max(i)$, such that $R_0(i) < \lfloor n/p \rfloor$. The lists below show the xp possible sequences of computers that need to be down in order for an extra process to end up on computer y.

1.1 $(y- \sum_{i=1}^{1} r'(i) + n) \bmod n$

1.2 $(y- \sum_{i=1}^{2} r'(i) + n) \bmod n$, $(y- \sum_{i=2}^{2} r'(i) + n) \bmod n$

…

1.x $(y- \sum_{i=1}^{x} r'(i) + n) \bmod n$, $(y- \sum_{i=2}^{x} r'(i) + n) \bmod n$,…,$(y- \sum_{i=x}^{x} r'(i) + n) \bmod n$

2.1 $(y- \sum_{i=1}^{1} r'(i) - \lfloor n/p \rfloor + n) \bmod n$

2.2 $(y- \sum_{i=1}^{2} r'(i) - \lfloor n/p \rfloor + n) \bmod n$, $(y- \sum_{i=2}^{2} r'(i) - \lfloor n/p \rfloor + n) \bmod n$

…

2.x $(y- \sum_{i=1}^{x} r'(i) - \lfloor n/p \rfloor + n) \bmod n$, $(y- \sum_{i=2}^{x} r'(i) - \lfloor n/p \rfloor + n) \bmod n$,…,
$(y- \sum_{i=x}^{x} r'(i) - \lfloor n/p \rfloor + n) \bmod n$

…

p.1 $(y- \sum_{i=1}^{1} r'(i) - (p\text{-}1)\lfloor n/p \rfloor + n) \bmod n$

p.2 $(y- \sum_{i=1}^{2} r'(i) - (p\text{-}1)\lfloor n/p \rfloor + n) \bmod n$, $(y- \sum_{i=2}^{2} r'(i) - (p\text{-}1)\lfloor n/p \rfloor + n) \bmod n$

…

$p.x$ $(y- \sum_{i=1}^{x} r'(i) - (p\text{-}1)\lfloor n/p \rfloor + n) \bmod n$, $(y- \sum_{i=2}^{x} r'(i) - (p\text{-}1)\lfloor n/p \rfloor + n) \bmod n$,…,
$(y- \sum_{i=x}^{x} r'(i) - (p\text{-}1)\lfloor n/p \rfloor + n) \bmod n$

Since $x = \max(i)$, such that $R_0(i) < \lfloor n/p \rfloor$, we see that no computer is included into more than one of the lists one to x. By looking at the proof of Theorem 4 we see that the bound is optimal as long as no computer is included in two such lists. The theorem follows. •

6 Conclusion

In many cluster and distributed systems, the designer must provide a recovery scheme. Such schemes define how the workload should be redistributed when one or more computers break down. The goal is to keep the load as evenly distributed as possible, even when the most unfavorable combinations of computers break down, i.e. we want to optimize the worst-case behavior which is particularly important in real-time systems.

We consider n identical computers, which under normal conditions execute the same number of processes (p) each. All processes perform the same amount of work. Recovery schemes that guarantee optimal worst-case load distribution when x computers have crashed are referred to as optimal recovery schemes for the values n and x. We have previously obtained recovery schemes that are optimal when at most $\lfloor {}^2\log \lfloor n/p \rfloor \rfloor$ computers are down (i.e. when $x \leq \lfloor {}^2\log \lfloor n/p \rfloor \rfloor$) [4]. The main contribution in this paper is that we have defined an improved recovery scheme that is optimal also when a significantly larger number of computers are down. For large n and small p the difference in terms of the number of crashed computers for which we can guarantee optimality can be significant. For instance, for $n = 1000$ and $p = 1$ the improved scheme guarantees optimality for almost three times as many crashed computers compared to the previous scheme (i.e. 9 crashed computers vs. 26 crashed computers, see Fig. 4).

We have shown that the problem of finding optimal recovery schemes for a system with n computers, with p processes on each computer, corresponds to the mathematical problem of finding the longest sequence of positive integers such that the sum is smaller than or equal to $\lfloor n/p \rfloor$ and the sums of all subsequences are unique. Some initial experiments indicate that this problem is NP-hard. The technique we use for generating such sequences is based on a greedy approach, but it turns out that our greedy approach will not result in the longest possible sequence in all cases. As discussed above, the sequence that we generate with the greedy approach is, however, significantly better than the previous result (e.g. almost three times better for $n = 1000$ and $p = 1$).

Our recovery schemes can be immediately used in commercial cluster systems, e.g. when defining the list in Sun Cluster using the *scconf* command. The results can also be used when a number of external systems, e.g. telecommunication switching centers, send data to different nodes in a distributed system (or a cluster where the nodes have individual network addresses). In that case, the recovery lists are either implemented as alternative destinations in the external systems or at the communication protocol level, e.g. IP takeover [8].

The difference between the schemes that we suggest and an intuitive scheme can be arbitrarily large.

References

1. G. Ahrens, A. Chandra, M. Kanthanathan, and D. Cox, *Evaluating HACMP/6000: A Clustering Solution for High Availability Distributed Systems*, in Proceedings of the 1995 Fault-Tolerant Parallel and Distributed Systems Symposium, 1995, pp. 2-9.

2. D. Häggander and L. Lundberg, Memory Allocation Prevented Server to be Parallelized for Better Database Utilization, in Proceedings of the 6th Australasian Conference on Parallel and Real-Time Systems, Melbourne, Nov/Dec, 1999, pp. 258-271.
3. L. Lundberg and D. Haggander, *Multiprocessor Performance Evaluation of Billing Gateway Systems for Telecommunication Applications*, in Proceedings of the ISCA 9th International Conference on Computer Applications in Industry and Engineering, Orlando, December, 1996.
4. L. Lundberg and C. Svahnberg, *Optimal Recovery Schemes for Fault Tolerant Distributed Computing*, in Proceedings of the 6th Annual Australasian Conference on Parallel and Real-Time Systems, Melbourne, November 1999 (Springer-Verlag), pp. 153-167. An updated version of the article will also be published in Journal of Parallel and Distributed Computing, November/December 2001 (special issue on cluster computing).
5. *Managing MC/ServiceGuard for HP-UX 11.0*, http://docs.hp.com/hpux/ha/index.html
6. *Comparing MSCS to other products*, http://www.microsoft.com/ntserver/ntserverenterprise/exec/compares/CompareMSCS.asp
7. S Mullender, *Distributed Systems (Second Edition)*, Addison-Wesley, 1993.
8. G.F. Pfister, *In Search of Clusters*, Prentice-Hall, 1998.
9. Sun Cluster 2.1 System Administration Guide, Sun Microsystems, 1998.
10. TruCluster, Systems Administration Guide, Digital Equipment Corporation, http://www.unix.digital.com/faqs/publications/cluster_doc
11. W. Vogels, D. Dumitriu, A. Agrawal, T. Chia, and K, Guo, Scalability of the Microsoft Cluster Service, Department of Computer Science, Cornell University, http://www.cs.cornell.edu/rdc/mscs/nt98.