Multiprocessor Performance Evaluation of a Telecommunication Fraud Detection Application

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

Abstract

The Fraud Control Center (FCC) is an object oriented C++ application running on Sun Solaris. Performance and scalability requirements are meet by having the application executing on Symmetric Multi-Processors (SMP). FCC is based on a commercial Relation DataBase Management System (RDBMS). Multithreaded programming is used to distribute the application over the SMP processors and to enable parallel database requests.

Performance evaluations using an 8 processor SMP showed that the original implementation did not scale up well. As a first step in the performance improvement process, the number of threads (database clients) was optimized. However, the speedup was still limited even after these optimizations, and our experiments showed that dynamic memory management was the major bottleneck. Therefore, two versions with optimized memory management were evaluated. Due to these optimization efforts, the speedup was increased from 2.7 to 4.3, using a SMP with 8 processors.

Keywords: Symmetric multiprocessors, Relation database management systems, Multithreading, Object oriented design, Dynamic memory, Scalability, Cellular fraud.

1. Introduction

Fraud is one of the cellular operators 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 [6].

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++ [9] (approximately 10,000 lines of code), using object-oriented design.

The rapid growth in the telecommunication market increases the performance requirements on FCC. One way of improving performance is using Symmetric Multi-Processors (SMP:s). However, it is not trivial to increase the performance by adding more processors; performance bottlenecks can easily limit the performance increase significantly [1][5]. A previous performance study of another multithreaded telecommunication application showed that the performance can actually decrease when the number of processors increases [2].

This paper includes a case study of the industrial FCC project at Ericsson. A project which was followed for over one year. It also includes an experiment where FFC is evaluated using a SMP with 8 processors.

Earlier evaluations have identified problems with dynamic memory, especially in applications using object oriented design. Therefore, two new FCC versions (besides the one implemented by Ericsson) with modified dynamic memory handling were implemented and evaluated. In the first one, the dynamic memory handler of the operating system was replaced. In the second, the application was divided into a number of Unix processes. The purpose of these experiments was to verify the hypothesis that the FCC suffered from performance problems due to dynamic memory management.

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 experiments are described. The results of the experiments are presented in Section 4. Section 5 discusses the experiences from the industrial Ericsson FCC project. 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 solution to prevent subscriber fraud. In subscriber 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.

2.2. System Overview

Figure 1 shows an overview of the total 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.

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. 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 predefined 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.

Graphical user interfaces are used to define, turn off and turn on rules, as well as for generating statistics reports.

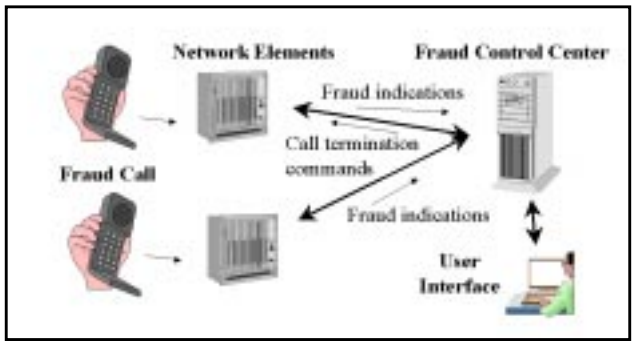

Figure 1. The FCC system.

The FCC application consists of five software modules, all executing on the same SMP (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. 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). The rest of this paper will mostly focus on the Main and Database modules.

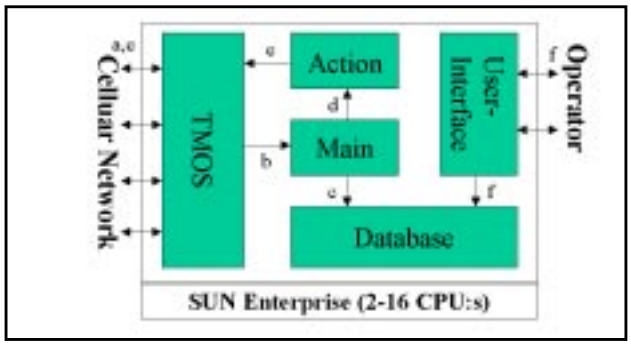

Figure 2. The five modules of FCC.

2.3. The Client/Server Architecture in FCC

A central part of FCC is data storage and data processing. A commercial RDBMS (Sybase [7]) that allows SMP execution was used in the implementation. Using a commercial RDBMS reduces the development and maintenance cost compared to implementing the database 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. Each database request is generally processed by an engine. A RDBMS executing on an 8-way SMP normally has 8 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.

There are a number of techniques which increase the performance of a RDBMS with a single client.

None-blocking requests is one such technique (see Figure 3 A). When using none-blocking requests the clients do not have to wait for the result from an issued request. Instead, it can send the next request, which can be processed in parallel with the previous ones. However, the use of none-blocking requests requires more advanced programming and design of the application software.

A more straight forward solution is to introduce multiple database connections. This can be done by simply starting a number application instances, each executing within its own Unix process (see Figure 3 B). Such a distribution does, however, imply that the application global functionality and shared resources may need to be synchronized with inter-process communication. Communication which can be both inefficient and complex.

A third alterative is to use multithreaded programming. Multithreaded programing makes it possible to write parallel applications which benefit from the processing capacity of SMP:s. It also gives the possibility to accomplish multiple database connections without dividing the application into processes or to implement none-blocking request handling (see figure 3 C).

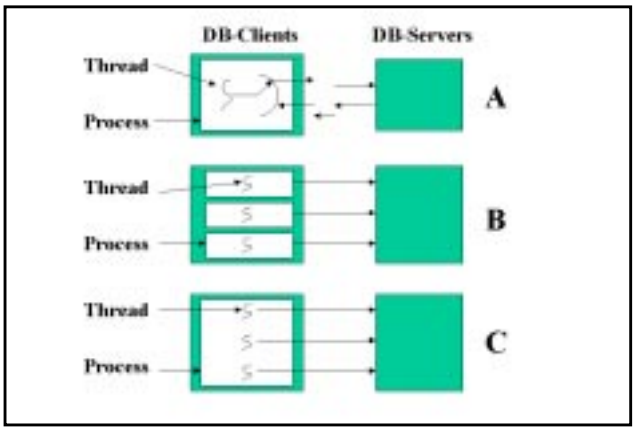

Figure 3. Three possible solutions for improving the performance of a single-client database system.

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 speed-up other parts of the application which are not directly related to the RDBMS.

2.4. The Implementation

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 itself. The number of simultaneous parser threads is dynamic. However, FCC gives the possibility to define an upper limit. The parser in FCC is designed in a way which makes it highly adaptable. It is very important for FFC to quickly supports 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 design results in frequent use of dynamic memory.

The Main module has a configurable number of permanent connections towards the database module (4d), i.e. the database server. Each connection is handled by a dbclient thread. A dbclient thread handles one event at a time by taking the first event off the event queue (4e) and then processing it according to the scheme in Figure 5. The interaction with the database is made with SQL commands [7] via a C-API provided by the database vendor, i.e. Sybase. Each

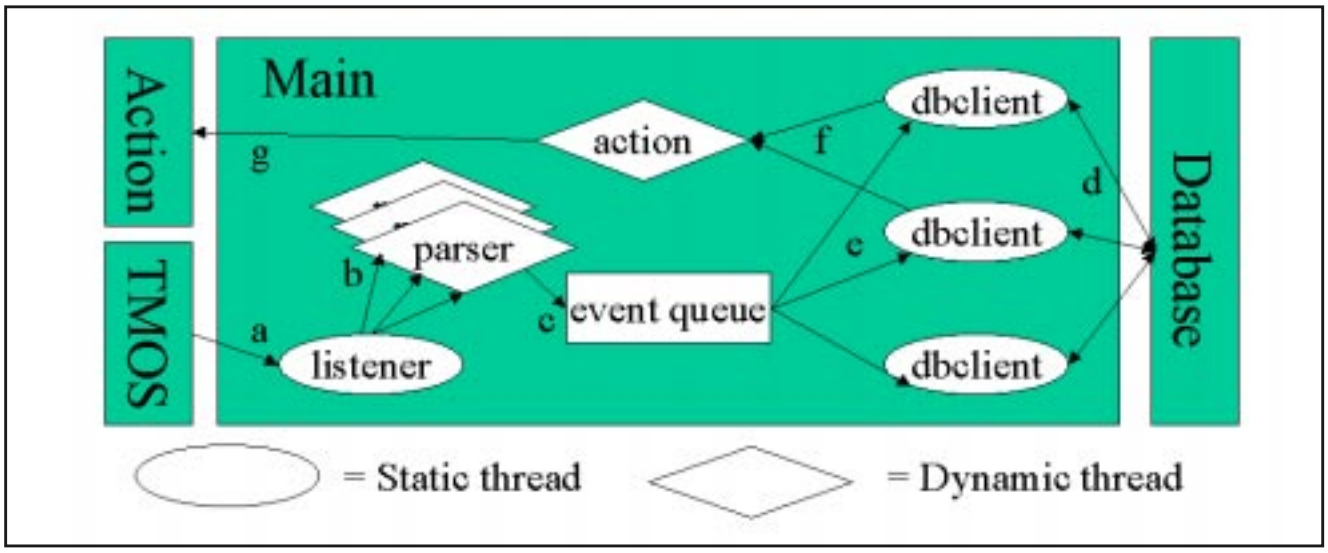

Figure 4. The tread architecture of FCC.

SQL command is constructed before it is send 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.

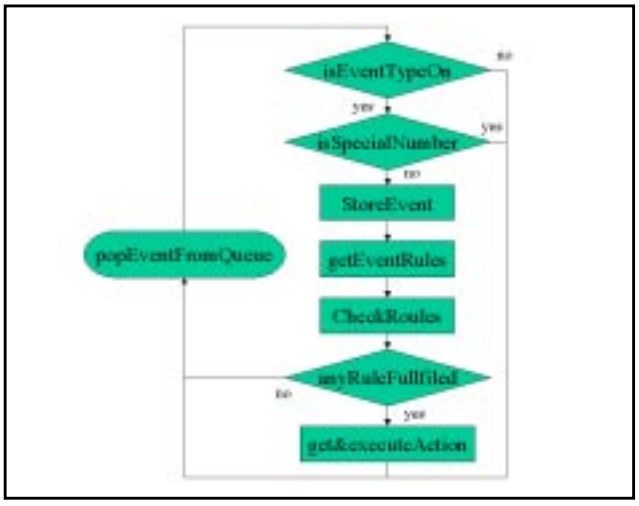

Figure 5. Processing scheme.

2.5. The Database Optimizations

Performance was prioritized in the design. In order to achieve high performance a database expert from Sybase was consulted. However, the consulting was limited to the earlier stages of the design. The actions taken in order to improve the performance at that point were:

- Separate disk for the transaction log.
- Index tables on 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 [8]). Database optimization techniques are not a subject for this study.

2.6. Symmetric Multiprocessing in FCC

Two of FCC's five modules are more CPU consuming than the others. 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 not immediate request waiting to be processed. As a result, each engine more or less allocates a processor. The remaining processors are mainly used by the Main module, via Solaris.

3. The Experiment Setup

The intention with the experiment was to:

- Quantify the increase in performance when using multithreaded programming to obtain multiple client connections to the database server.
- Find out if the maximum performance was bounded by the multithreaded database client or the by the database server when FCC is executing on an 8-way SMP.

3.1. Two optimized FCC versions

Earlier experiments have indicated that object-oriented techniques dramatically increase the number of dynamic memory allocations for an application. 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 [3]. 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 in 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 [2]).

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 are the system overhead generated by the contention for entrance.

Our hypothesis is that contention in the dynamic memory management is a bottleneck in the multithreaded FCC database client (i.e. the Main module). This hypothesis was tested by doing measurements on two optimized FCC versions. In one of these versions the ptmalloc [10] was replacing 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 Volfram 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.

In the second optimized version, the FCC Main module was divided into two and three processes, i.e. the application used two and three dynamic memory resources, respectively. Dynamic memory contention is only generated by real parallelism, i.e. simultaneously memory allocations from two or more processors. The multithreaded client in FCC utilized at most two or three processors (see section 2.6). Therefore, three processes would be enough to avoid dynamic memory contention. In FCC the effort of splitting the client into a number of processes was very limited. However, for other applications this approach could be costly or even impossible.

3.2. The experiment environment

The hardware in the experiment 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 twelve disks. All disks were paritioned 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 FFC was installed in the experiment environment.

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

One of the intentions with the experiment was to leave the database installation and configuration intact. However, disk tracing during the initial phase of performance evaluation indicated heavily write activity on the disk handling the database transactions. This disk became a direct bottleneck in the system (see Figure 10). 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. A part from this, no changes within the installation or the configuration of the database were made.

3.3. The simulation of workload

It would be extremely difficult to use a real telecommunication network in the experiment. The solution was to develop an event generator which 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 6). The same simulator was used in the industrial Ericsson project.

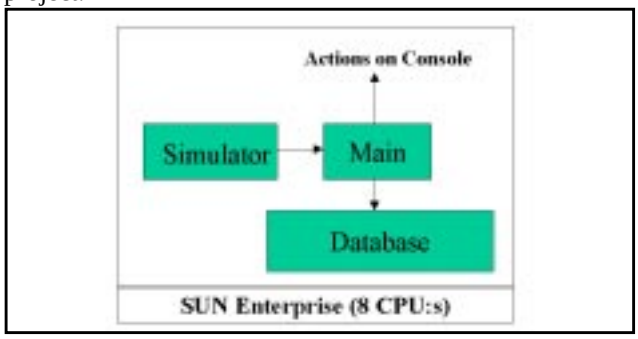

Figure 6. The experiment environment.

It is common practice in the telecommunication community to use simulated workloads based on an exponential

distribution functions.The function used for event generation can be seen in Figure 7. 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 to events.

$$F(t) = 1 - e^{-\lambda t}$$

$$t = \frac{1}{\lambda}\ln(x)$$

x = random[0..1]

Figure 7. The exponential distribution function used to generate the workload.

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

These settings are by Ericsson considered 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, the FCC application was configured in following way:
- All event types were processed (see Figure 5), i.e. no events were turned off.
- All events were matched against ten friendly subscriber id:s, i.e. no subscriber id:s had special treatment.
- All events were tested the three rules described below.

The three rules were:
- 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.

4. The Results of the Experiment

The results of the experiments are shown in three speed-up diagrams (see Figures 8, 9 and 10). Figure 8 shows the increase in performance when using multiple threads on a single engine database server. Figure 9 shows the speed-up for a database server with 6 engines. In Figure 10, the number of thread is 3 per engine and the figure shows the increase in performance when using up to 6 database server engines on an 8 processor SMP. The reason for having 3 threads per engine was that the thread speed-up diagram indicated 3 threads to be optimal (see Figure 8).

When comparing the results from the speed-up test for a server with 6 engines (Figure 9) with the results for the total FCC speed-up test (Figure 10), the first test comes up with a somewhat better peak performance. Our intention in the first test was to quantify the speed-up for a multithreaded client. 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.

4.1. Thread speed-up for a single engine (Fig. 8)

A reason for the increase in performance for a single engine, i.e. a database sever which only can benefit from the power of one processor, is that idle time created by disk assesses and latency in the communication protocol between the client and the server can be utilized. A maximum speed-up of 1.3 was measured for 3 threads.

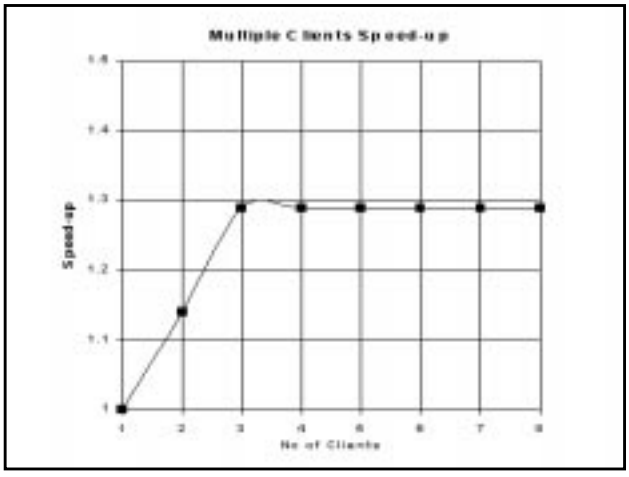

Figure 8. The performance of a single database engine.

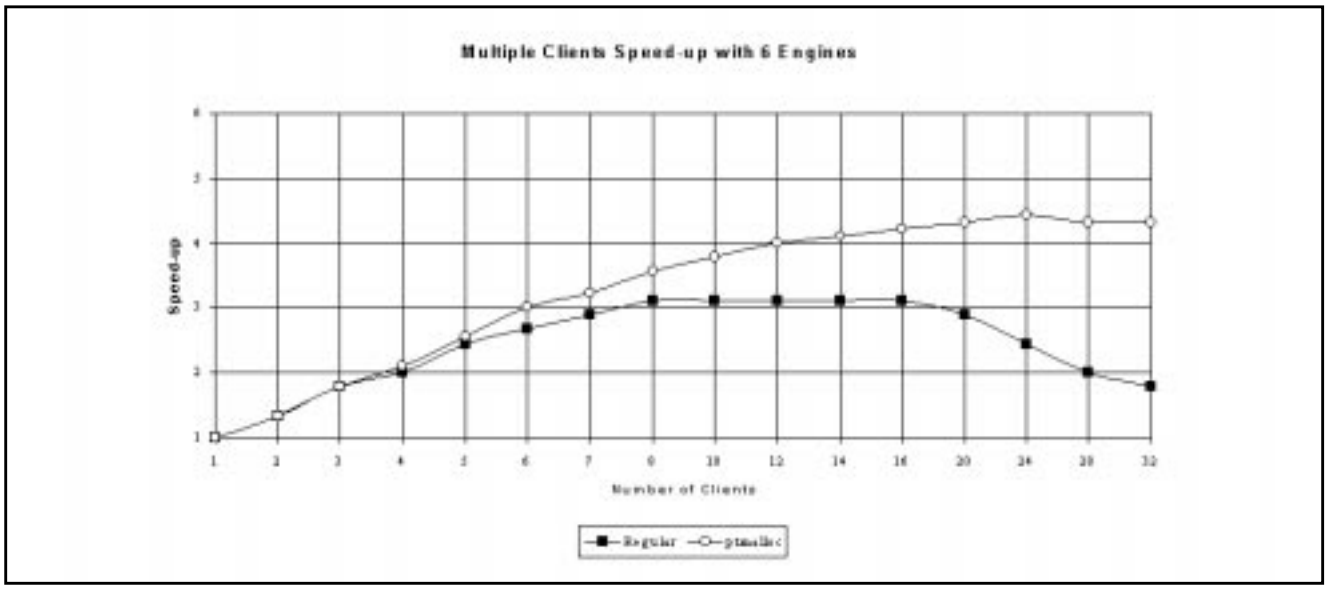

Figure 9. The thread Speed-up of FCC.

4.2. Thread speed-up for 6 engines (Fig. 9)

Two version of FCC were tested. The original version and a ptmalloc version. The original version had a significantly lower speed-up and had its maximum capacity already with 8 threads (3.1). The ptmalloc version had a maximum speed-up on 4.4. The value was measured using 24 threads, i.e. 4 threads per engine.

4.3. The FCC speed-up on an 8-way SMP (Fig. 10)

The speed-up diagram shows speed-up curves for five different FCC versions.

-The original version with cashed transaction log.
-The version with two Unix processes.
-The version with three Unix processes.
-The ptmalloc version.
-The original version without cashed transaction log.

An un-cashed transaction log made it hard for FCC to speed-up. Also when the transaction log was cached by the Solaris file system, FCC did not speed-up sufficiently. The reason for this is lock contention within the dynamic memory handling. Ptmalloc has proven to successfully solve this problem [2]. The measurement made on the ptmalloc version of FCC also verified this. In FCC's case it was also ef-

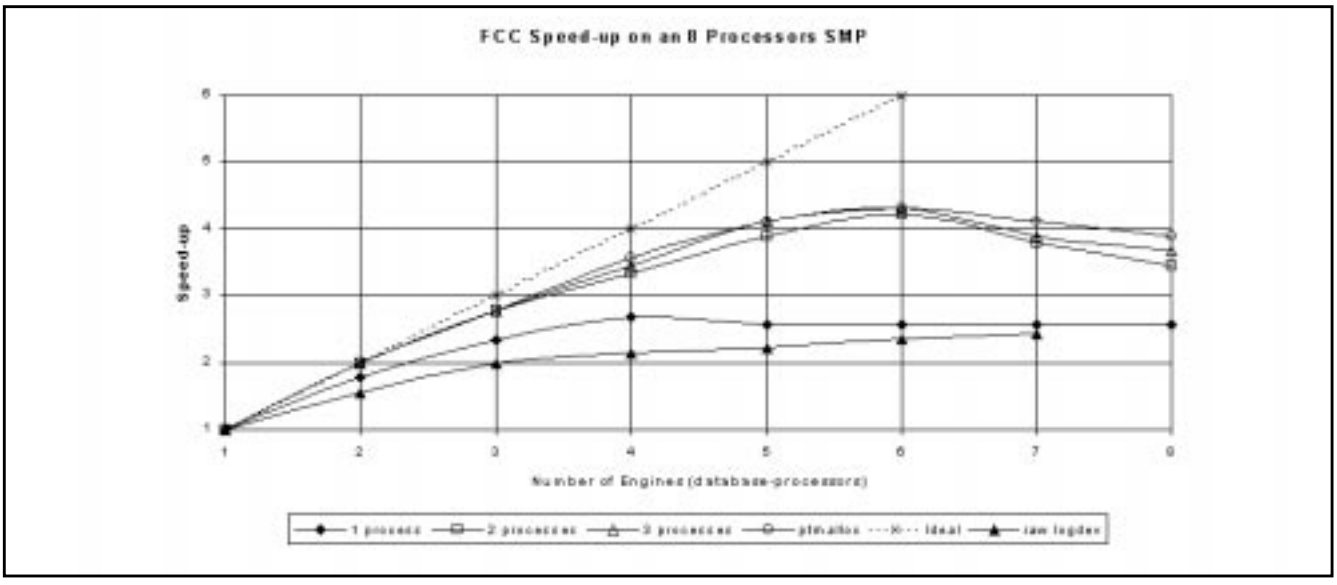

Figure 10. The FFC speed-up on an 8 processor SMP.

ficient to divided the FCC Main module into 2 or 3 processes, i.e. 2 and 3 memory images.

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 been used. The maximum performance was therefore achieved when 6 processors was used for the database server and 2 by the multithreaded FCC Main module (see sections 2.6).

5. Experiences from the FCC project

A long and close relationship between the Ericsson company and our research group has made it possible for us to follow projects such as FCC in detail. This section shortly describes some performance related issues, which occurred during the FCC project.

5.1. Product Prototype

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 processors in order to handle the expected workload. An additional result of the prototype was that at least five parallel database connections 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 speed-up problems (see raw log in Figure 10).

5.2. Implementation

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, this lead to a separation of the Main and the TMOS modules. The interaction with TMOS was normally made via an API. Access via a 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 program makes a complete copy of the parent process, including all existing threads [4]. Further on, a new set of “thread safe” libraries had to be ordered from Sybase, since the one usually used was of a none safe type.

Since most software developers were unfamiliar with concurrent programing, 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 which consumes work hours and decreases the quality of the product.

5.3. Performance problems

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 an build in congestion protection. When a transaction holds more than a certain number of page locks, 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. A deletion which could take up to 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.

5.4. Multithreading verses multiple Unix processes

The prototype had shown that a number of parallel database connection had to be used. The parallel database connections had in the FCC prototype been accomplished with multiple Unix processes. However, similar problems had been solved with multithreaded programing in other Ericsson products. This fact, together with a willingness from the software developers to work with new technology, was probably the main reason for choosing multithreading. A decision which was maintained even though the project was advised not to use multithreading. In a design based on multiple Unix processes the TMOS module could have been accessed directly, no new software had to be obtained from Sybase and “old traditional” programming, including documentation and testing methods could have been used.

However, there are a number of similar products where it would be more difficult or even impossible to divide the database client into multiple processes.

6. Conclusion

One important conclusion from the industrial Ericsson FCC project is that the present and future performance re-

quirements 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 also 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 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. Our experiment shows that a database with 6 engines (database server processes) 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 issuing 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, there is only one database client in many telecommunication applications.

In order to obtain high performance in database applications with only one client, the designers will often use multithreading. Performance evaluations using a SMP with 8 processors showed that there should be 3-4 clients threads for each database server engine.

Experience from other multithreaded object-oriented applications shows that dynamic memory management tend to be a performance bottleneck [2] 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 4.3. The speedup is 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.

We used two different approaches for optimizing the dynamic memory handling in FCC. One approach was to replace the standard memory handler with a parallel memory handler called 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 advantage of the ptmalloc approach is that we do not need to modify the application code in any way. The advantage of splitting the client into multiple processes is that we can use standard memory handler. In FCC the effort of splitting the client into a number of processes was very limited. However, for other applications this approach could be costly or even impossible.

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.

References

[1] 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.

[2] 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.

[3] C. Larman, “Applying UML and Patterns”, Prentice Hall, 1998.

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

[5] 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.

[6] Catharina Lundin, Binh Nguyen and Ben Ewart, “Fraud management and prevention in Ericsson’s AMPS/D-AMPS system”, Ericsson Review No. 4, 1996.

[7] J. Panttaja, M. Panttaja and J. Bowman, “The Sybase SQL Server -Survival Gudie”, John Wiley & Sons, 1996.

[8] S. Roy and M. Sugiyama, “Sybase Performance Tuning”, Prentice Hall, 1996.

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

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

[11] http://www.rougewave.com/products/tools/tools.html

[12] http://www.sybase.com/products/databaseservers/