Bachelor thesis in
Computer Science
May 2010

Performance evaluation of multithreading in a Diameter Credit Control Application

Pontus Rantzow
Gustav Åkesson

School of Computing
Blekinge Institute of Technology
Sweden

Performance evaluation of multithreading in a Diameter Credit Control Application

This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the Bachelor degree in Computer Science. The thesis is equivalent to 2 x 10 weeks of full time studies.

Contact Information:
Author(s):
Pontus Rantzow
Gustav Åkesson
E-mail: Gustav Åkesson <gustavakesson@hotmail.com> , Pontus Rantzow <p.rantzow@gmail.com>

University Advisor:
Professor Håkan Grahn
School of Computing
Phone: +46 455 38 58 04

School of Computing
Blekinge Institute of Technology
SE - 371 79 Karlskrona
Internet : http://www.bth.se/com
Phone : +46 455 38 50 00
Fax: +46 455 38 50 57

Acknowledgements

We would like to thank Professor Håkan Grahn for contributing with invaluable knowledge and at all times a positive attitude.

We would also like to thank Capgemini Karlskrona for taking the time to give continuous feedback on the Diameter Credit Control Application prototype. That made this thesis become even better.

-- Gustav Åkesson
-- Pontus Rantzow

ABSTRACT

Moore's law states that the amount of computational power available at a given cost doubles every 18 months and indeed, for the past 20 years there has been a tremendous development in microprocessors. However, for the last few years, Moore's law has been subject for debate, since to manage heat issues, processor manufacturers have begun favoring multicore processors, which means parallel computation has become necessary to fully utilize the hardware. This also means that software has to be written with multiprocessing in mind to take full advantage of the hardware, and writing parallel software introduces a whole new set of problems.

For the last couple of years, the demands on telecommunication systems have increased and to manage the increasing demands, multiprocessor servers have become a necessity. Applications must fully utilize the hardware and such an application is the Diameter Credit Control Application (DCCA). The DCCA uses the Diameter networking protocol and the DCCA's purpose is to provide a framework for real-time charging. This could, for instance, be to grant or deny a user's request of a specific network activity and to account for the eventual use of that network resource.

This thesis investigates whether it is possible to develop a Diameter Credit Control Application that achieves linear scaling and the eventual pitfalls that exist when developing a scalable DCCA server. The assumption is based on the observation that the DCCA server's connections have little to nothing in common (i.e. little or no synchronization), and introducing more processors should therefore give linear scaling.

To investigate whether a DCCA server's performance scales linearly, a prototype has been developed. Along with the development of the prototype, constant performance analysis was conducted to see what affected performance and server scalability in a multiprocessor DCCA environment. As the results show, quite a few factors besides synchronization and independent connections affected scalability of the DCCA prototype.

The results show that the DCCA prototype did not always achieve linear scaling. However, even if it was not linear, certain design decisions gave considerable performance increase when more processors were introduced.

KEYWORDS

Diameter, DCCA, Multiprocessing, Server, Performance, Cache coherence, Dynamic memory

TABLE OF CONTENTS

ABSTRACT....4
KEYWORDS....4
INTRODUCTION....6
1.1 BACKGROUND....6
1.2 RESEARCH OBJECTIVES....6
1.3 HYPOTHESIS....7
1.4 METHODOLOGY....7
1.5 SCOPE AND LIMITATIONS....7
1.6 MAIN CONTRIBUTION AND RESULTS....8
BACKGROUND....9
2.1 MULTIPROCESSOR SYSTEMS AND PARALLEL COMPUTING....9
2.2 DIAMETER AND DCCA....10
2.3 PERFORMANCE EVALUATION....11
EXPERIMENTAL ENVIRONMENT....13
3.1 THE DCCA PROTOTYPE....13
3.2 PLATFORM....16
EXPERIMENTAL RESULTS....17
4.1 OVERALL PERFORMANCE OF THE DCCA PROTOTYPE....17
4.2 DYNAMIC MEMORY AND ALLOCATORS....18
4.2.1 DCCA ANALYSIS....18
4.2.2 GENERAL CASE ANALYSIS....20
4.3 TRANSPORT PROTOCOLS....23
4.4 THE IMPACT OF LONG SERIALIZATION POINTS....24
4.5 CACHE COHERENCE....27
4.5.1 FALSE SHARING....27
4.5.2 PROCESSOR AFFINITY....30
4.6 MEMORY BLOCK COPYING....32
DISCUSSION....33
CONCLUSIONS AND FUTURE WORK....36
6.1 CONCLUSIONS....36
6.2 FUTURE WORK....37
REFERENCES....38
BIBLIOGRAPHY....38
WEB RESOURCES....38
GLOSSARY....39
APPENDIX A - SERIALIZATION POINTS....40
APPENDIX B - MEMORY BLOCK COPYING....42

Chapter

1

INTRODUCTION

1.1 BACKGROUND

Diameter is a networking protocol referred to as AAA - Authentication/Authorization/Accounting - and is used in the telecommunication industry for controlling access and the charging of services such as mobile Internet. Diameter is a successor to RADIUS, and in contrast to RADIUS, Diameter runs over reliable transport protocols (TCP and SCTP), enables network layer security (such as IPSec) and has better support for roaming. These are only a few features that differ the two protocols. Diameter base protocol standard is defined in the RFC3588 [RFC3588].

With the introduction of multiprocessor systems, a server today is able to have unique connections work on different processors simultaneously with the use of threads. On a computer with a single processor, only one thread can work at a time, but threads can still be useful and can for example be used as an abstraction to make programs easier to manage. However, even though employing a multiprocessor system in theory could mean huge performance gains, it also introduces another set of problems to consider. A multithreaded DCCA is no exception.

Capgemini is an international consulting company with several telecommunication clients. As the need for Diameter-based solutions has increased, Capgemini has become interested in a study of a practical implementation of a Diameter-based server - a Diameter Credit Control Application (DCCA). The purpose of a DCCA is to provide a framework for real-time charging, and as the performance demand on telecommunication systems increase, Capgemini was especially interested in the capacity and limitations that comes with a DCCA. They were particularly interested in measuring what affects the accounting throughput of a multithreaded DCCA, running on a multiprocessor system.

1.2 RESEARCH OBJECTIVES

This thesis' goal is to investigate and evaluate the performance of a Diameter Credit Control Application, in a multithreaded Symmetric Multiprocessing (SMP) environment. This will be done by identifying the performance bottlenecks and how to address those bottlenecks. The definition of performance is in this case the throughput - i.e. the number of credit control requests per second that the DCCA is capable of handling.

1.3 HYPOTHESIS

In a DCCA, the connected nodes have little in common. That is, the connections will not need any significant serialization points between them. Therefore, the hypothesis is that a DCCA will be able to scale near linearly with respect to the number of processors.

However, load imbalance could be a potential problem in a DCCA. One connected node may send significantly more requests than another node. This means that the connection itself must be parallelized to fully utilize the processors. What performance issues will arise in such a scenario, and how can those issues be resolved?

1.4 METHODOLOGY

To be able to draw conclusions from the performance evaluation of a DCCA, a prototype has to be developed. This prototype will be the basis of performance measurements in this thesis. Briefly described, the implemented prototype will be able to connect up to 50 simultaneous client nodes. These nodes can send Credit Control Requests (CCRs) to the DCCA and the server responds with Credit Control Answers (CCAs). The node's request is always granted (since this is a prototype), but before the DCCA answers the node, a simulation is performed of the work a DCCA does in the case of a CCR.

Along with the prototype server, a complete data structure for handling Diameter-messages (including all supported Attribute Value Pairs, AVPs, which are an essential part of a Diameter message) will be developed. This makes it easier to manage the byte-sequence that is received from the network. It also makes it easier to handle and send specific Diameter-messages over the network. Finally, a DCCA client will be developed as a helping tool in the performance evaluations and measurements. The client's only purpose is to benchmark the capacity of the server.

The work described in this thesis is done in the form of exploratory approach [Robson-02]. Once the DCCA prototype is complete, performance analysis tools are used to find bottlenecks and hotspot areas in a credit control scenario. Information will also be gathered to find out e.g. what C++ specific optimizations can be done for a DCCA, in respect to multithreading and scalability.

The evaluations will be conducted by benchmarking the server in one scenario - 50 client nodes send a total of three million CCR, and receives as many CCA. The idea is that if a value is extracted from one measurement, will there be any improvements if the server is modified and measured under the exact same scenario? There will also be general case analysis of certain scenarios. During the work of this thesis, the following saying will constantly be present:

"If you can't measure it, you can't control it. And if you can't control it, you can't improve it."

1.5 SCOPE AND LIMITATIONS

The ways to increase performance are many - there's hardware, programming languages, compilers, operating systems and many other aspects that all have different impact on an application's

performance. This thesis will, however, try to limit the performance optimizations and scalability evaluations to the software in a Diameter Credit Control Application server architecture. For instance, an Intel© processor using Intel© compiler could get a performance increase in comparison to the g++ compiler. Aspects such as different compilers will not be taken into consideration in this thesis.

This thesis will focus on evaluating performance and scalability in a Diameter Credit Control Application server. Any other Diameter-based server applications are not covered. That also means that performance evaluations of DCCA clients will not be covered.

To fully understand this thesis, knowledge about programming, memory, networking and a certain degree of parallel computing is required.

1.6 MAIN CONTRIBUTION AND RESULTS

This research thesis has evaluated a multithreaded Diameter Credit Control Application (DCCA), in respect to multiprocessing performance and scalability. To be able to properly evaluate DCCA, a prototype that mimics the real-world as close as possible was developed. The primary goal was to evaluate scalability, and what affects multiprocessing performance of the DCCA prototype. The results of this thesis should aid multithreaded server development within the telecommunication industry, where Diameter is used as the networking protocol. It should particularly aid the development of multithreaded DCCA servers, but could also be of interest for any development of multithreaded applications.

The main results show that the DCCA prototype did not always achieve linear scaling. Aspects such as memory management, synchronization and cache coherence have significant impact on server scalability. However, even if scalability was not linear, certain design decisions gave considerable performance increase when the DCCA prototype was employed on a multiprocessor system.

Chapter

2

BACKGROUND

2.1 MULTIPROCESSOR SYSTEMS AND PARALLEL COMPUTING

A multiprocessor system is a computer system that consist of more than one processor that can work simultaneously. An example of such a system is Symmetric (shared memory) MultiProcessing. SMP is a computer system that consists of multiple identical cache-processor subsystems that share the same physical memory and connect via a bus. This type of centralized shared-memory is by far the most popular organization [Hennessey-03].

The term multiprocessing refers to the possibility of running a computer system with more than one processor. The idea behind multiprocessing is that performance can be doubled by employing twice as many processors. But the reality is that this is rarely the case, even though multiprocessing under certain circumstances can result in increased performance. Along with multiprocessing comes a series of problems one must consider.

As multiprocessor systems more or less have become the norm of today's computer systems, it makes sense to use parallel computing techniques to increase performance. For multiprocessor systems that consist of a single computer (such as SMP), parallelism is achieved by using threads [Gerber-05]. A thread is a single stream of control in the flow of a program. A thread does not necessarily follow the sequential flow of an application, which means that on a SMP system, independent threads can execute simultaneously, i.e. in parallel.

Gaining performance increase by using threads can be accomplished either by a clever compiler that can determine where it's appropriate to *thread* (i.e. extract parallelism), but in most cases the programmer must explicitly express a parallel formulation of the application. That is, the programmer must write code where *threading* is appropriate.

To be able to write a multithreaded application the programmer must identify the tasks than can be executed concurrently. Once identified, the programmer can use an API such as POSIX threads (pthreads) to express the identified concurrency.

Pthreads defines an extensive API for creating and manipulating threads. Pthreads is a so-called low-level thread API, which means that the programmer has a greater responsibility of the threads than if an

API such as OpenMP (which is considered high-level) is used. The differences between low-level and high-level thread-API is similar to the differences between assembly language and C++ - a classic tradeoff in control and complexity.

In order to achieve a performance increase, there are common threading goals to consider [Gerber-05]:

- Focusing on hotspots - There is no point in optimizing the part of an application that consumes an insignificant part of the execution time.
- Minimize synchronization - Threads that require little synchronization will in most cases result in higher performance, in contrast to those threads that require more synchronization.
- Minimize memory sharing - Multiple processors do not like to share memory, since memory sharing results in extra bus transactions and usually additional synchronization.

However, there are not only threading goals that the programmer must consider. Along with parallel computing and threading, there are pitfalls and other issues:

- Short loops - Thread creation, termination and synchronization in a parallelized short loop could prove to be more expensive than sequentially executing the loop.
- False sharing – Results in unnecessary bus transactions by sending a cache-line back and forth between the processors.
- Processor affinity - Describes the specific processor(s) that a thread executes on. A potential performance issue is when threads are constantly moving between the different processors.
- Total memory bandwidth - The processors collectively share the fixed memory bandwidth. For memory intensive applications, the total bandwidth can be a problem even if more processors are introduced.
- Synchronization overhead - Synchronization may force threads to wait for each other. This means that threads wait without doing any work, with the consequence of reducing parallelism of the application.

As presented, multiprocessing introduces a whole new set of possibilities and problems. Threading goals and pitfalls described earlier will be discussed and evaluated in this thesis. In particular in regards to the Diameter Credit Control Application server.

2.2 DIAMETER AND DCCA

Diameter is a networking protocol that controls communication between connected nodes. For instance, this could be between an authenticator (such as a Diameter-based server) and a network node requesting authentication. Diameter can be used in the telecommunication industry.

Diameter is a successor to the RADIUS protocol (the name itself is a pun of RADIUS, Diameter is twice what RADIUS is) and offers improvements. Amongst those improvements are reliable transport (either TCP or SCTP), error handling, network layer security and a more extensive roaming support.

Diameter is thought to be the next generation AAA - Authentication/Authorization/Accounting -

protocol. Authentication refers to authenticate an entity's identity, authorization determines whether a specific entity is allowed to perform the requested activity, and accounting refers to pursuing the user's access to network resources. Since the emergence of new technologies such as mobile Internet, the requirements and demands of authentication and authorization have increased. This is where RADIUS is considered to be insufficient [RFC3588].

The Diameter protocol is not forced to be running on top of a specific application. Instead, Diameter focuses on general message exchanging features. In the case of this thesis, Diameter will be used to run on top of a Diameter Credit Control Application. A DCCA, as the name suggests, is a Diameter application used for credit control purposes. Briefly described, the DCCA provides a framework for real-time charging, primarily used within telecommunication systems. This could, for instance, be to grant or deny a user's request of a specific network activity. DCCA is of course also meant to account the eventual use of the network resource.

A Diameter message is the communication unit to send a request or deliver answers to connected Diameter nodes. A Diameter message can come in many forms. Depending on certain message values, a message could for instance be a watchdog (message used to confirm connections are still alive) or credit control requests that carries accounting-related information.

The Diameter message's information is used to identify and carry information about the intent of the message. However, the actual data resides in Attribute-Value-Pairs (AVPs). For instance, a credit control request message could contain the AVPs origin-host (which node sent the message), and accounting-record-type (which determines the action of the credit control request, e.g. if accounting should start or terminate). A Diameter message can contain any given number of AVP's, depending on the message's intent.

2.3 PERFORMANCE EVALUATION

Before the experimental results are presented, it is important to define certain aspects of performance evaluation. Evaluation and benchmark will be defined in the following sub chapter.

A benchmark is a program or process that aids in objectively evaluating the performance of an application. The benchmark is run before and after a performance optimization in order to detect if the optimization had the desired outcome. If a benchmark is not conducted, there is no way for the programmer to know if an optimization was successful or not. In fact, an optimization could turn out to be a decrease in performance. It is also fundamental that the application is run in the exact same manner before *and* after the benchmark. Otherwise, it is not possible to confirm that the optimization itself was the reason behind the new outcome.

There are certain key attributes for a benchmark.

- The benchmark must produce repeatable results. A benchmark that produces different results each time it is run is not useful.
- The benchmark should be representative. The best benchmark mimics the actual situation.

- The benchmark should be easy to run and produce results that are simple to interpret.

This definition of a benchmark and key attributes have been adopted when creating benchmark applications and conducting benchmarks on the DCCA prototype.

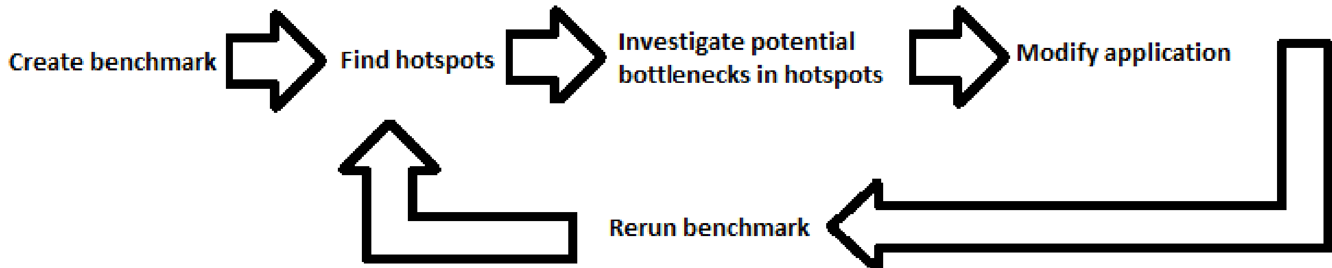

Figure 2.1 The software optimization process.

Finding hotspots in the DCCA prototype will mainly be done by using Intel© VTune™ Performance analyzer Linux tools. For instance, using Intel© VTune™ Call Graph analyzer aids in pinpointing hotspot areas during execution.

In this thesis, performance evaluation refers to evaluating different solutions in respect to performance. The evaluations will be based on benchmarks, where timing mechanism (stopwatch) and throughput are the main performance measuring tools.

Throughout this thesis, certain performance evaluation metrics will be used. These are:

- Speedup – refers to how much faster a parallel algorithm is in contrast to the sequential version. For instance, if a sequential algorithm takes 16 seconds to execute, but the parallel algorithm executes in 4 seconds, the speedup is 4 (16/4 = 4).
- Parallel efficiency – is a metric of the time that a processor is fully utilized. For instance, if a sequential algorithm takes 16 seconds to execute, whereas on 8 processors the parallel algorithm executes in 4 seconds, the parallel efficiency is 0.5 (16/4/8 = 0.5). That is, the parallel efficiency is 50%. The higher efficiency, the better utilization of the processors. If speedup is not superlinear (which it in some cases may be), maximum value is 100% utilization of the processors.
- Scalability – a parallel system is considered scalable if the speedup increases asymptotically with the number of processors.

Chapter

3

EXPERIMENTAL ENVIRONMENT

3.1 THE DCCA PROTOTYPE

The DCCA prototype has a fixed number of threads (50 for this thesis), referred to as connection threads. The amount of connection threads (or at least the maximum number) can be determined prior to the server execution. Due to the nature of a DCCA, the different nodes that may connect are usually known. As mentioned, 50 connection threads will be used during measurements, which enables 50 clients to connect simultaneously.

These connection threads listen for incoming connections. Once a client wants to connect, a thread picks up the request and handles only that single connection. After that, the client sends a given number of message requests to the server, whereas the server answers each message request. Once the connection is terminated, the connection thread will return to a listening state.

When a connection is established, a fixed number of worker threads are created (20 for this thesis). These threads will work with the specific connection and it's Diameter-messages. That is, the worker threads will read from and write to the network connection, but also simulate DCCA work and save each received message to a queue. This queue will later be saved to disc.

For each established connection, a single thread is created that handles saving data (Diameter-messages) to disc. This thread sleeps for a given amount of time before waking up and starting to work on the message queue. The work is, in this case, to empty the message queue and convert the messages to one big XML-chunk, and save that chunk.

However, the part of the actual disc access has been excluded. It would have been too difficult to draw any conclusions regarding other performance aspects since accessing the disc would be the single biggest bottleneck. The preparation of disc access (see Appendix A, Figure A.2) should therefore be considered as simulated work of a real DCCA server.

A brief use case scenario of the prototype is:

1. Main application thread creates a given number of connection threads.
2. Each connection thread waits until a client node connects. There can be as many simultaneously connected nodes as the amount of created connection threads.
3. When a client has connected, the connection thread creates a given number of worker threads. A thread for file handling is also created.
4. The worker threads deal with the connected node – receive message (request), handle the request, save request to a connection-shared queue and answer the node's request.
5. A file thread created by the connection thread wakes up after a given time and handles the connection's message queue. Each message is dequeued and merged into a single XML-chunk. After the queue has been emptied, the XML-chunk is meant to be saved to disc.
6. Step 4-5 is repeated until the connection is closed. Once that happens, the worker threads and file thread terminate, and the connection thread returns to a listening state (step 2).

Figure 3.1 is a general illustration of the DCCA prototype.

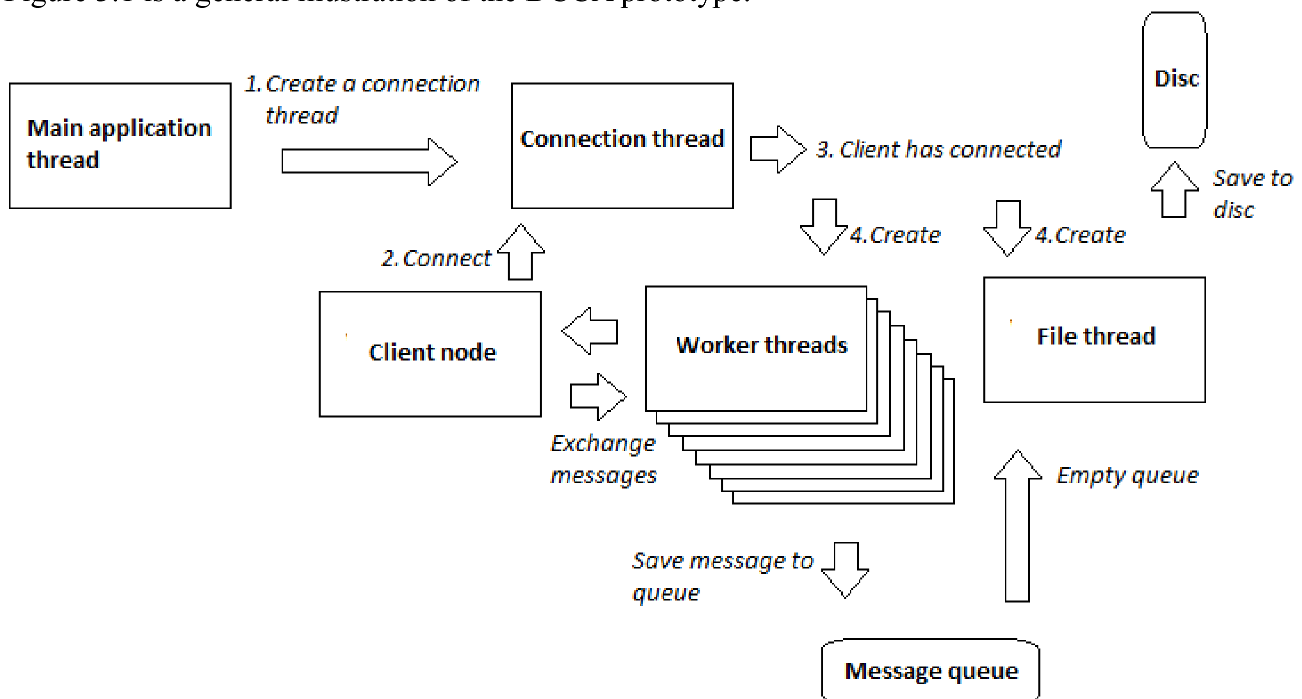

Figure 3.1 A general description of the DCCA prototype. Note that for simplicity, this only illustrates one connection. However, the illustration can easily be translated to any given number of connections.

The data structure for Diameter messages is represented by a number of classes (Figure 3.2). Object-oriented inheritance is used to represent AVPs, since an AVP can be one of 8 different AVP datatypes (for instance, 32 bit integer or an octet string) [RFC3588]. Different AVPs are identical in every aspect, except that the datatype the AVP is carrying may differ. Due to that fact, inheritance is an appropriate approach as it also makes it easier to add new non-standard datatypes.

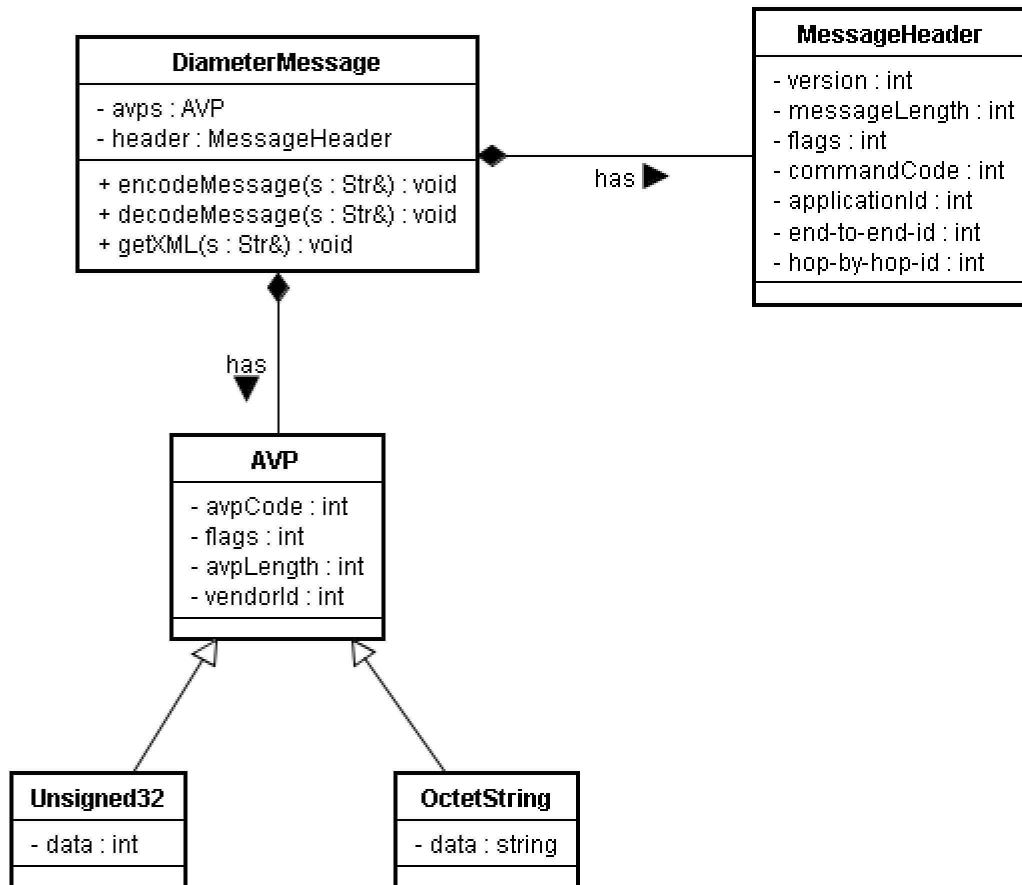

Figure 3.2 The Diameter message data structure. OctetString and Unsigned32 (which are 2 out of 8 existing datatypes) inherit from AVP. Note that subclasses data differs, in respect to datatypes.

A Diameter message-structure may contain any given number of AVP-objects. A Diameter message only has one MessageHeader. The Diameter message design does not only make it easy to manage Diameter messages, but the data structure has several appropriate member functions for handling different DCCA needs. For instance, it is possible to turn a properly concatenated byte array (i.e. that has been received from a connected client) into a easily manageable Diameter message. It is also possible to turn the data structure object itself into a byte array that can be sent to the connected client.

Besides the Diameter message-structure, several other user-defined classes are commonly used in the DCCA prototype. For instance, a class handling character arrays (string) have been used. It is similar to STL string, but the string class used for the DCCA prototype has features that STL string could not provide. It is also easier to integrate the user-defined string class into other user-defined classes. A class for network handling has also been used, which makes network management easier. Figure 3.3 shows an example of the user-defined String, Network and Log classes.

```
bool exit = false;
String x;//Used-defined string class
int i;
Log l;//Class for logging.
Network net;//Network class

net.bind(9000);//Bind port 9000

while(exit == false)
{
    //Wait for incoming connections (wait for max 5 seconds)
    i = net.accept(5.0);

    if(i == -3)//Fatal error while waiting
        exit = true;
    else if(i == -2)// Accept timeout - accept again
        continue;
    else //Connection was accepted
    {
            while(exit == false && net.isConnected() == true)
            {
                    // Wait for data (max 2 seconds). Read into x object
                    if(net.read(x) <= 0)
                            exit = true;
                    //Write recieved data to log file
                    l.log("Received [%s]", x.c_str());
                    //Transmit 3 characters, transmit timeout = 1.0
                    net.transmit("OK\n", 3, 1.0);
            }
    }
}
//Disconnect cleanup
net.disconnect();
```

Figure 3.3 An example of three commonly used classes – String, Log and Network.
The example is a simple server that a client can connect to.

3.2 PLATFORM

The evaluations will only be conducted on the following platform:

- Intel(R) Xeon(R) CPU E5335, consisting of totally 8 processors (2 physical processor chips, with 4 cores each). Each processor core is running at 2.00GHz.
- Total memory capacity is 16 GB.
- Network is Gigabit Ethernet.
- Operating system is x64 GNU/Linux 2.6.24-23-server.
- Programming language is C++, and applications are compiled using g++ 4.2.4 and glibc 2.7. During compilation, optimization level -O3 is used. No other optimization flags are set.

The thread API used will be POSIX threads (pthreads).

Chapter

4

EXPERIMENTAL RESULTS

4.1 OVERALL PERFORMANCE OF THE DCCA PROTOTYPE

The final result of this thesis is shown in Figure 4.1 as the total throughput the DCAA prototype can handle. The throughput has been measured using 1, 2, 4 and 8 processors. Figure 4.1 is the result of the final conclusions and evaluations of this thesis. In the following sub chapters, the analysis that lead to these results are outlined.

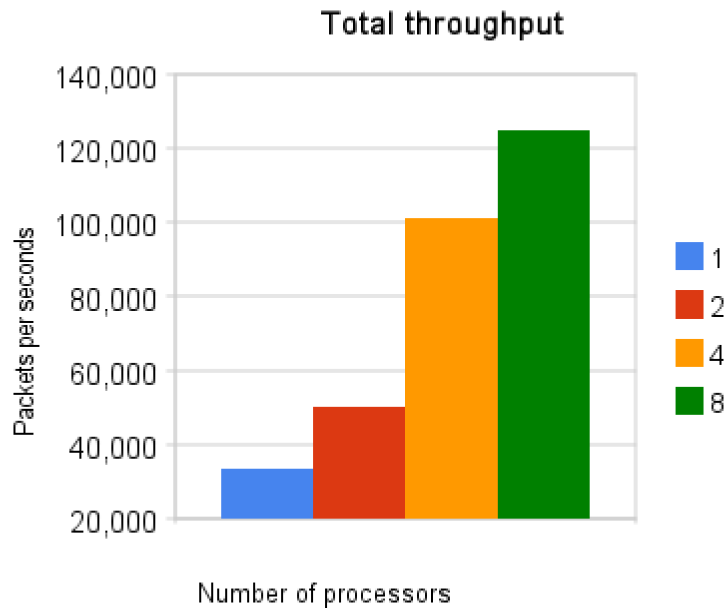

Figure 4.1 Final result of DCCA throughput.

4.2 DYNAMIC MEMORY AND ALLOCATORS

Dynamic memory enables an application to allocate memory when needed, and free it when it's no longer needed. For system design and flexibility, this could often be a desirable approach due to maintainability and design reasons. However, along with the flexibility of dynamic memory comes a cost in performance [Smith-01]. This sub chapter focuses on dynamic memory, and the goal is to find out how dynamic memory affects the DCCA prototype and find possible solutions on how to address eventual performance hits.

4.2.1 DCCA ANALYSIS

According to the Intel© VTune™ Call Graph, about 10% of the function calls in the DCCA developed for this thesis are to either C malloc() or free(). Frequent dynamic memory management can play a significant role in an application's performance, and the performance penalty often boils down to the default memory manager [Berger-00]. The performance penalty becomes even more significant in a multithreaded environment due to serialization locks used by the runtime library to access the shared memory heap [Intel].

Obviously, a performance optimization would be to reduce the number of dynamic memory calls, but a certain degree of dynamic memory is still needed. Due to that reason, measurements were done comparing glibc builtin memory manager and Google TCMalloc (Thread Caching Malloc).

The default memory manager is by nature general purpose and cannot make any simplifying assumptions about the application. An application may use dynamic memory in a very specific way and could therefore pay penalty for functionality it does not need. The default memory manager could also have features that penalizes performance, such as frequent system calls for every new() and delete() and serialization points (such as contention for the memory heap) that reduce scalability [Bulka-03]. Using other allocators than the default could address these issues by having memory pools and thread caches. Not only does this aim to reduce system calls and serialization points, but also aims to reduce cache coherence problems [Berger-00]. A specially designed memory manager for a multithreaded environment (such as TCMalloc) could therefore be beneficial for the performance in the DCCA prototype.

Figure 4.2 and Figure 4.3 shows a benchmark of the DCCA using glibc malloc and TCMalloc. Figure 4.2 shows throughput, whereas Figure 4.3 shows parallel efficiency based on throughput.

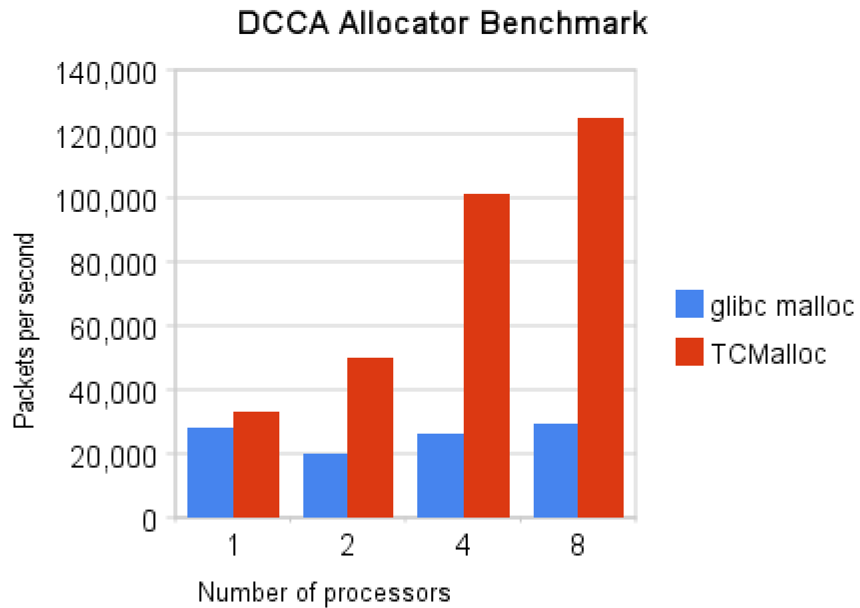

Figure 4.2 Result of DCCA throughput, using different allocators.

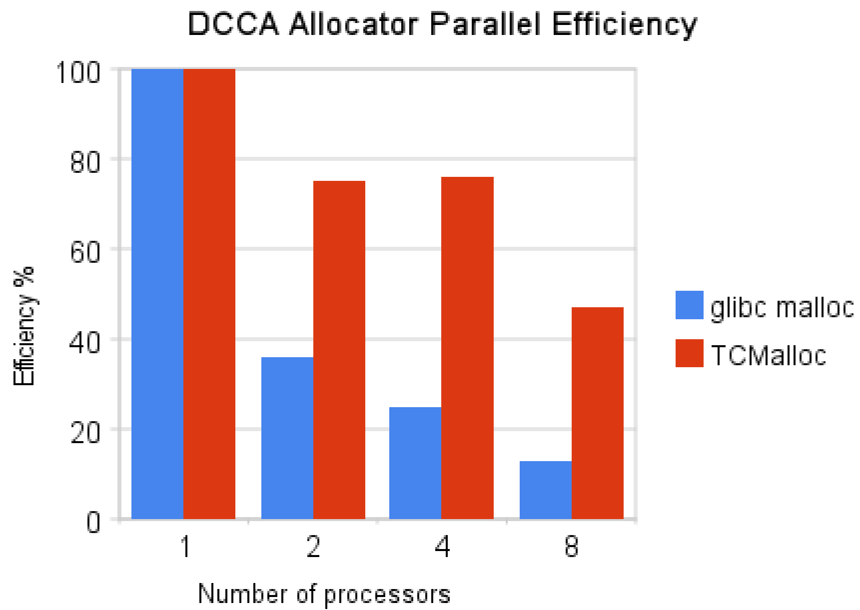

Figure 4.3 DCCA Parallel Efficiency, using different allocators.

As shown in the Figure 4.2 and Figure 4.3 , TCMalloc completely outperforms glibc malloc. In fact, no matter the amount of processors used, the performance increase with glibc malloc is negligible.

TCMalloc works by assigning thread-local cache from the central memory heap to each thread. In this cache, smaller allocations are done without any serialization. TCMalloc uses garbage collection for thread-local storage that is considered redundant. This redundant memory migrates from the local cache to the central heap. Thread caching could mean a higher memory use, and TCMalloc may be more memory hungry than other malloc implementations [Ghemawat].

A number of measurements were conducted both on the DCCA prototype, but also in more isolated situations. Besides DCCA benchmarking, integer allocation as well as Diameter message handling were measured.

4.2.2 GENERAL CASE ANALYSIS

Hoard is another allocator developed for multithreaded environments. Hoard uses mmap exclusively, but handles memory in 64 kilobyte chunks referred to as *superblocks*. Hoard's heap is logically divided into a common, global heap but also a given number of per-processor heaps. As with TCMalloc, Hoard also employs thread-local cache that holds a limited number of superblocks. The developers of Hoard claims the allocator achieves almost linear scalability with the number of threads [Berger-00].

The Diameter message benchmark was conducted by letting a given number of threads (8, 16, 32) work on a pre-created Diameter message-object. Inside the message structure the message deals with dynamically allocated AVP-structures. During each iteration, 11 of such AVP-structures were added to the message. The message is then encoded to a byte-sequence and lastly it's decoded from a byte-sequence to the message structure with all the added AVPs. This benchmark case represents the hotspot part of the DCCA where the dynamic memory management is primarily based. By not including other performance variables (such as network overhead), the measurement results of different allocators in a Diameter context are more thoroughly evaluated. 10 million iterations are performed in the scenario described earlier and the results are shown in Figure 4.4.

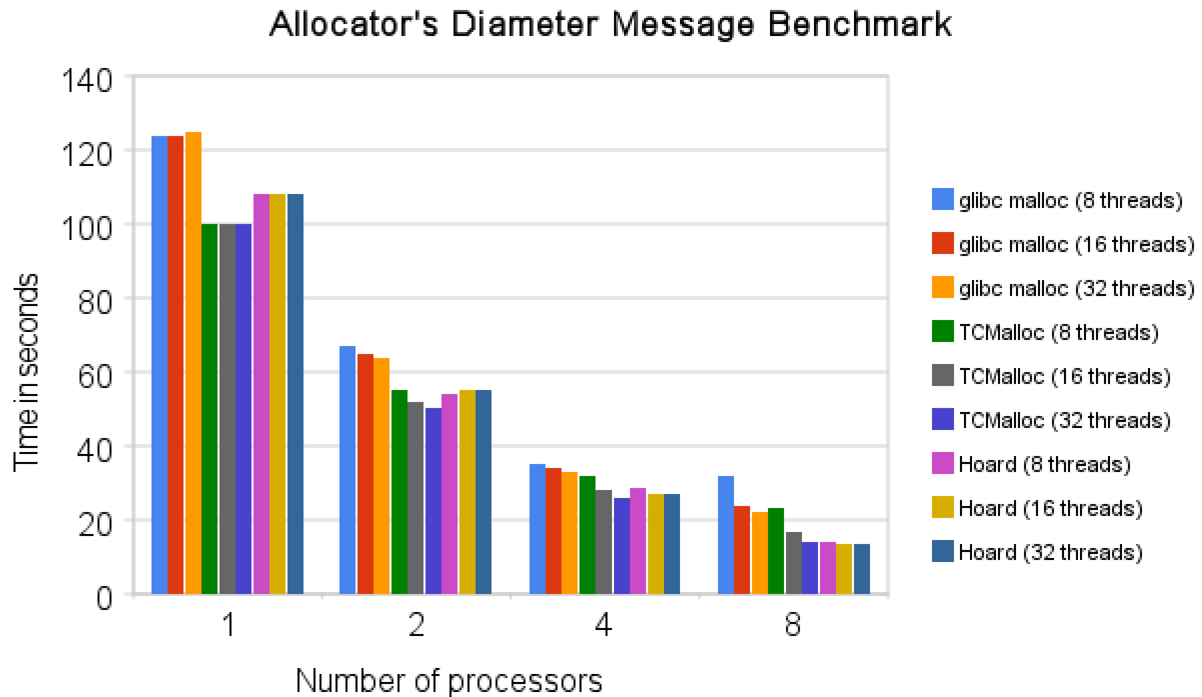

Figure 4.4 Diameter message handling benchmark, using different allocators.

In Figure 4.4, Hoard performs well at 8 cores, with little respect to the number of threads used. On 8 threads Hoard achieves a 2.3 speedup compared to glibc malloc and Hoard performs slightly better than TCMalloc. TCMalloc performs similar at 32 threads, but TCMalloc does not perform as well as Hoard with 8 threads.

The integer benchmark Figure 4.5 was conducted by letting a given number of threads (8, 16 or 32) share the work of allocating and deallocating a 400-byte array of integers. The total work was 1 billion allocation/deallocation, and the tests were measured on 1,2,4 and 8 processors.

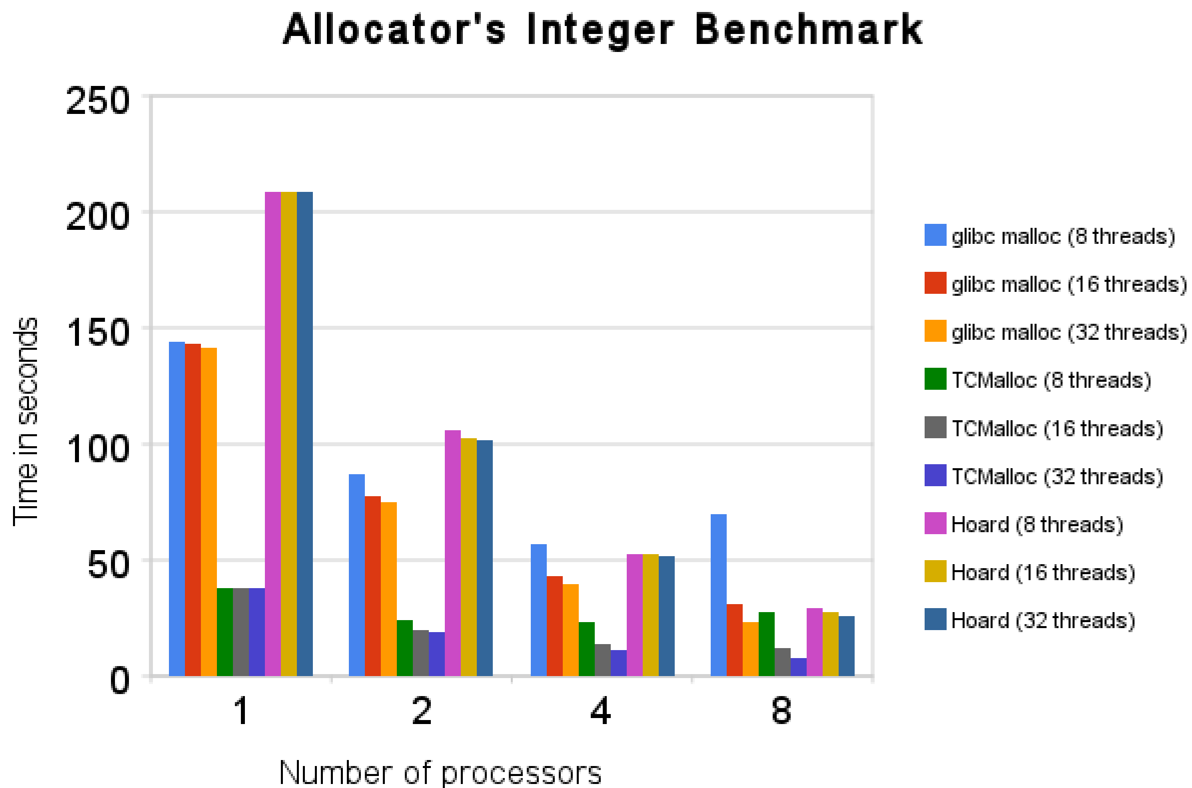

Figure 4.5 Allocation/Deallocation benchmark, using different allocators.

In Figure 4.5, Hoard performed surprisingly bad, as the allocator on one processor is ~30% worse than glibc malloc. However, as the number of cores increase as do Hoard's performance. At 8 processors, Hoard outperforms glibc malloc. Especially when using only 8 threads, as Hoard has a speedup of 2.3 compared to glibc malloc. TCMalloc performs well right from the start, and scales well on 32 threads - on 8 cores, 28 seconds elapsed with 8 threads, but only 8 seconds when using 32 threads. Increasing the number of threads beyond 32 did not give any further performance increase.

It was not possible to conduct any measurements with Hoard on the DCCA prototype. Even though it was confirmed the shared library was properly linked, the server had network issues when using Hoard. However, Hoard could still be of interest on another platform as it supposedly has a big theoretical performance increase in multithreaded environments [Lin-00]. Failing to use Hoard is also a reminder that using allocators other than the built-in is not only about performance increase - the implementation itself must be reliable, something Hoard could not provide for the DCCA prototype.

It is important to note that free() and malloc() are primitive system calls and in a reasonable measurement should not be directly compared to TCMalloc or Hoard. These allocators reduce calls to free() and malloc(), structures memory in an efficient way by logically separating the memory heap and are designed to give performance gains in multithreaded environments. However, this is not primarily a thesis on comparing allocators, but as the measurements show, there is a significant performance difference for a practical implementation. When developing a DCCA, or for that matter any

multithreaded server that in hotspot areas allocate dynamic memory, using different allocators can have a significant performance impact.

4.3 TRANSPORT PROTOCOLS

Unlike it's predecessor RADIUS, Diameter operates over reliable transport protocols TCP or SCTP. This comes with the expense of speed, but on the other hand it enables flow control and retransmissions of lost packets. Having a reliable transport protocol is immensely important for a charging telecommunication system.

TCP is well known whereas Stream Control Transmission Protocol (SCTP) first became a standard in 2000 [RFC2960]. SCTP is similar to TCP, such as both protocols support full duplex, is connection-oriented, provides reliable transmission and employs a flow control windowing mechanism. However, SCTP provides capabilities that TCP does not.

SCTP enables and uses the notion of multihoming. A multihomed host is a host with more than one network interface. At connection initialization, the SCTP peers exchange information about their interface(s) and if a SCTP message requires retransmission, that message can be sent to an alternate interface. This increases the reliability of the SCTP session. Even though this feature primarily is for reliability and not for load-sharing, load sharing could still be a potential side effect of being multihomed.

TCP is byte-oriented, meaning that the upper layer protocols have to count and accumulate bytes of each Diameter message. SCTP on the other hand is message-oriented, meaning that SCTP maintains message boundaries and delivers complete messages (called chunks) to the upper layer protocols. Dealing with TCP's byte-oriented nature could introduce certain network overhead, and potentially more system calls due to read and write operations of byte sequences.

Perhaps the most important difference is that SCTP provides multiple data streams between two peers, whereas TCP only allows one byte stream. This means that independent data exchanges may be delivered over different streams - messages lost in one stream does not affect other streams. In TCP, lost segments would delay delivery of all subsequent messages. This is referred to head-of-line blocking, which is a phenomenon limiting overall throughput - e.g. the total performance and scalability of the DCCA.

Measurements in Figure 4.6 were done comparing TCP and SCTP on the prototype developed for this thesis. The usage of SCTP is rather primitive, i.e. parallel message streams and multihoming were not included.

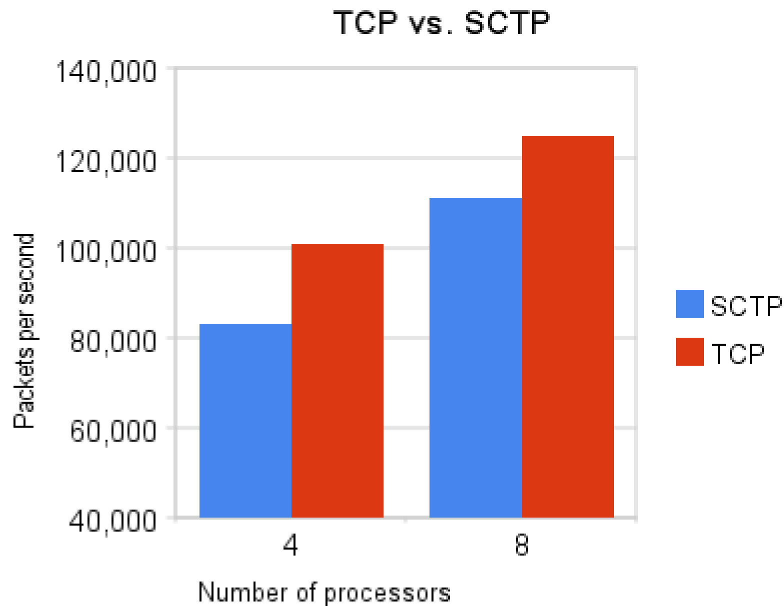

Figure 4.6 Result of DCCA throughput using TCP and SCTP,

Other research measurements have been done on how SCTP impacts server scalability [Ono-08]. Due to a large amount of association-related data (SCTP connection data), a server's scalability dropped by 20% when using SCTP instead of TCP. In those measurements, the drop was later reduced to 10% by adjusting capacity for accepting out-of-order data delivery (which SCTP supports), but the scalability decrease was still significant. These results confirm the measurements on the DCCA prototype in this thesis.

As noted earlier, SCTP became a standard in 2000 and is not nearly as improved as TCP, which has been around for almost 30 years. Most probably, there is room for improvements that will increase SCTP performance and as a consequence also server scalability for applications employing SCTP. If network throughput is a big concern for the DCCA and the network is a performance bottleneck, running Diameter on SCTP instead of TCP could be worth investigating. Measuring the impact of parallel message streams and multihoming is also important in order to take full advantage of all of SCTP's features. It is, however, important to verify performance measurements as there is no guarantee SCTP will result in a performance increase.

4.4 THE IMPACT OF LONG SERIALIZATION POINTS

If several threads try to read/write the same data simultaneously, race condition problems can occur. That is, threads *race* for executing a given expression(s). Depending on which thread that "wins" the race, the end result can differ. To avoid the occurrence of race condition problems, serialization points, also known as mutual exclusions, are used to protect the critical regions.

If a thread claims mutual exclusion, another thread will stop and wait (either sleep or busy wait) until it is safe to enter the critical region. The use of mutual exclusions can cause serialization problems since

no matter how many processors, only one at a time can access the critical region. It can be crucial to performance to try and avoid long mutual exclusions in hotspot areas.

The DCCA has to save each message in a container, to later be written to disc by another thread. A major bottleneck was found when a STL vector was used as a container, because every time a thread wanted to access the vector, mutual exclusion had to be claimed. This would not be a problem if only the threads that inserted messages had to deal with the mutual exclusion. The bottleneck occurred when the thread dedicated to emptying this vector had to keep claiming mutual exclusion until every message had been handled and removed (see Appendix Figure A.3). This meant no other threads could insert any messages until the emptying operation was complete. The result of this solution is shown in Figure 4.7.

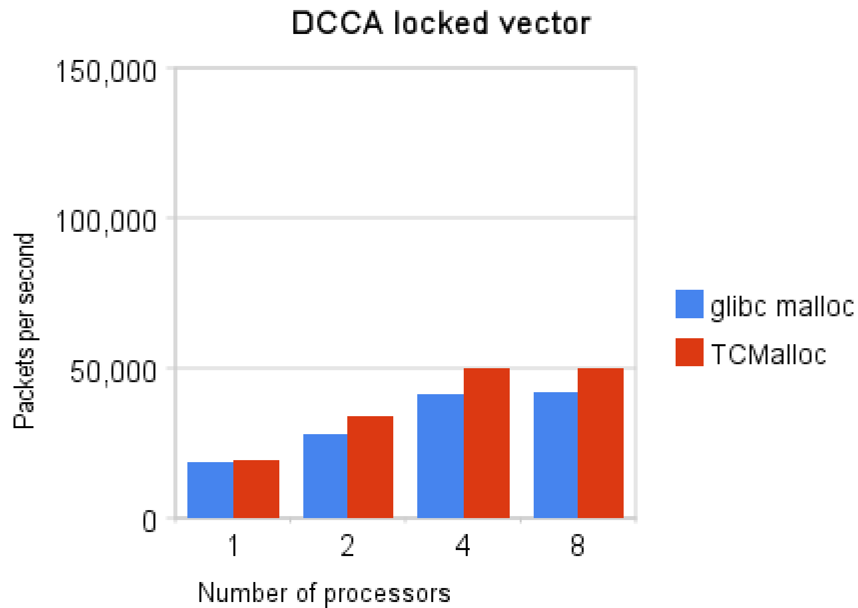

Figure 4.7 Result of DCCA throughput, using a locked STL vector.

The results in Figure 4.7 show no increase at all between 4 and 8 processors. To counter this problem, a linked list was used instead of a STL vector. The graph in Figure 4.8 demonstrates throughput, if all access of the list was done during mutual exclusion.

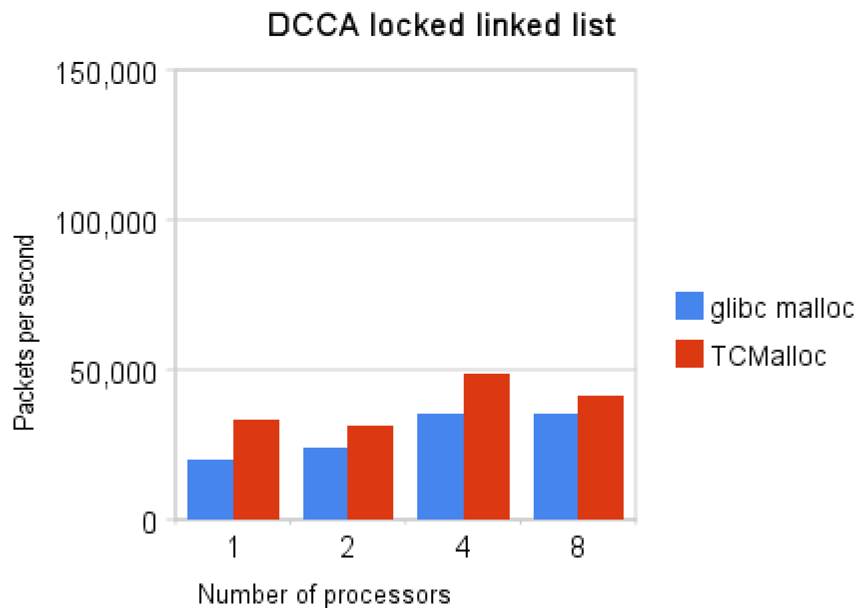

Figure 4.8 Result of DCCA throughput, using a locked linked list.

The results are worse almost across the board. These measurements rule out that the choice of data structure (vector or list) was the cause of the problem. However, by using a specially designed linked list data structure gives the possibility of inserting and removing messages simultaneously (even though those respective operations have to be done during mutual exclusion). The serialization point of emptying the list can therefore be alleviated, making it possible for the application to still insert messages even if the container is currently being emptied. See Appendix Figure A.2 for the complete source code of the emptying operations.

The list is accessed in two ways - one access is to the front of the list, while the other is accessed in the rear. If the application inserts messages at the rear of the list while the thread that empties the list takes messages from the front, both can work on the list simultaneously. Accessing the container by alleviating a major part of the synchronization in the common case fulfills an important threading goal [Gerber05]. Minimizing synchronization is a key to gain performance from an SMP environment. The impact of minimizing synchronization in this hotspot area is shown in Figure 4.9.

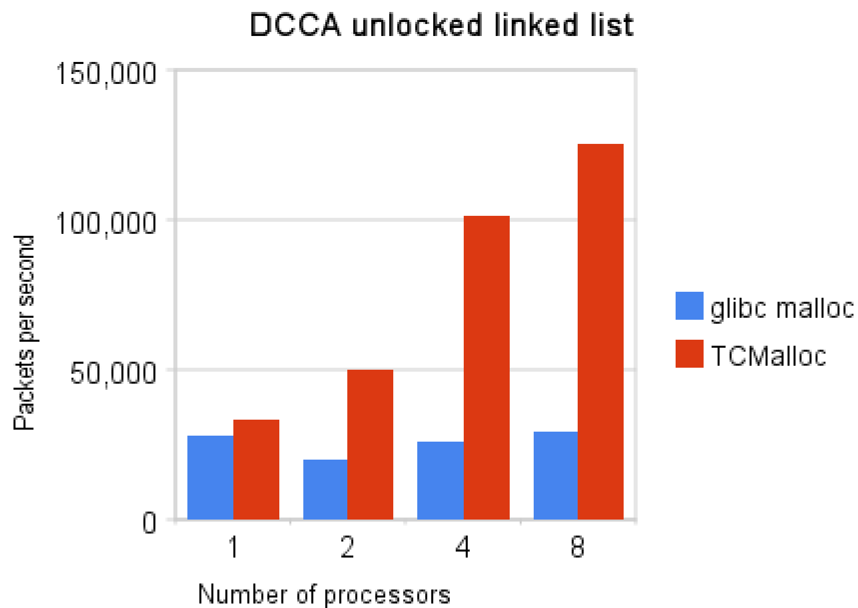

Figure 4.9 Result of DCCA throughput, using an unlocked linked list.

Comparing with Figure 4.8, the results in Figure 4.9 show up to 150% increase in throughput performance with TCMalloc on two or more processors. However, despite the minimized synchronization, glibc malloc shows a performance decrease on two or more processors.

4.5 CACHE COHERENCE

A common problem in a SMP environment is cache coherence, since each processor has it's own exclusive cache. If two processors work on the same memory block, a protocol to keep the data in the caches consistent has to be employed. For instance, if two threads on two different processors work on shared data, the data must travel between the different processor caches to stay coherent. The need for cache coherence protocols gives more transactions to the shared memory bus.

4.5.1 FALSE SHARING

For the DCCA prototype there are three mutual exclusions that each connection share, one for reading from the network connection, one for writing to the network connection and one for saving the read Diameter-message in a container. The mutual exclusions are contained inside the shared data structure for each connection (see Figure 4.14). These locks are constantly accessed and changed. In certain scenarios, this memory setup can cause a performance hit in the form of false-sharing.

False sharing occurs when several processors write to different data in the same cache-line, which means that the cache block will ping-pong between the different caches [Grama-03], [Gerber-05]. In the DCCA prototype, this could for instance be the mutex locks. To demonstrate this problem, a small benchmark application was developed that creates a given number of threads (8, 16 or 32). They all share the same data structure which has two different locks declared after each other (see Figure 4.10).

Half of the amount of threads locks and unlocks one of the them, while the other half does the same on the other mutex lock. The results are shown in Figure 4.11.

```
struct SharedResource
{
    pthread_mutex_t lockOne;
    pthread_mutex_t lockTwo;
};
```

Figure 4.10 Data structure without cache padding.

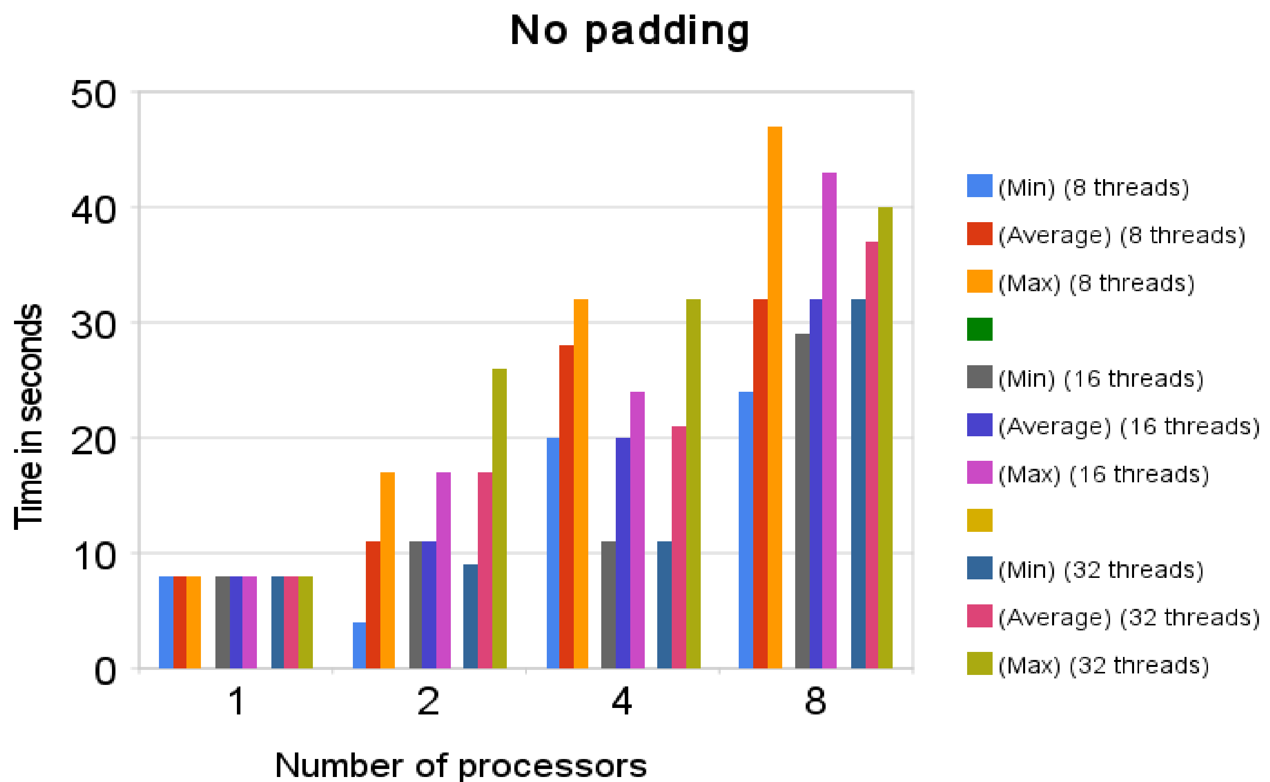

Figure 4.11 False sharing without using cache padding. Min refers to the minimum execution time during the measurements, Max to the maximum and Average to the average execution time.

As shown in Figure 4.11, there is no difference on a single processor, since the cache-line does not have to ping-pong anywhere. However, as the number of cores increase, the effects of cache-line ping-pong becomes more visible. It takes longer to do the same amount of iterations the more processors that are introduced. Also note that the results can vary by up to 300% on two or more processors, which happens since there is a random amount of how many cache-line ping-pongs occur. To confirm that there is a false-sharing problem a test with cache-padding is performed. In this case, the definition of cache-padding in this thesis is that enough data is added between the locks to push the locks into different cache-lines (see Figure 4.12). "Enough data" depends on the platform's cache-line size.

```
struct SharedResource
{
    pthread_mutex_t lockOne;
    int cachePadding[50];
    pthread_mutex_t lockTwo;
};
```

Figure 4.12 Data structure with cache padding.

The result if adding cache-padding between the locks in the data structure is shown in Figure 4.13.

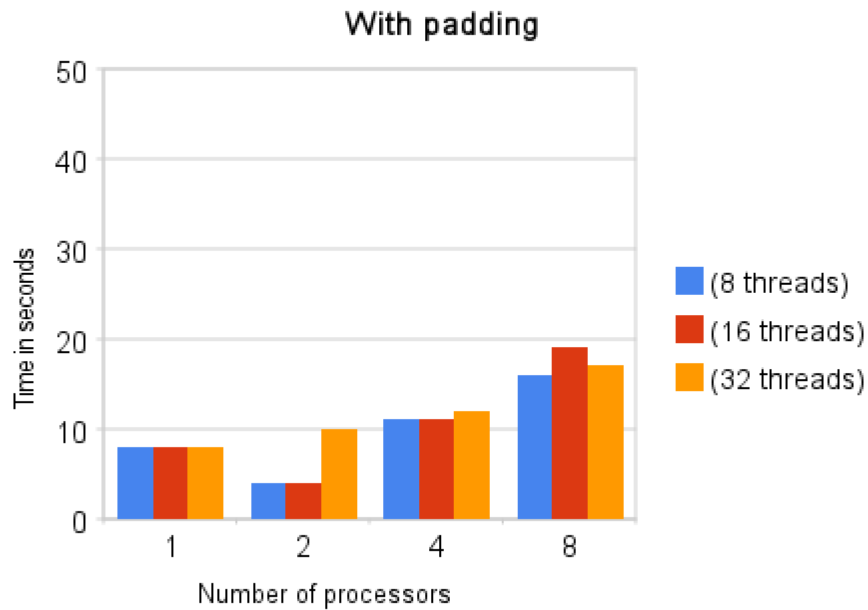

Figure 4.13 False sharing using cache padding.

As the results in Figure 4.13 show, on average there is an almost double speedup across the board on two or more processors. Also to note is the results are not fluctuating as much anymore, at maximum 50%. Compared to the non-padding in Figure 4.11, it is clear that there was a false-sharing problem. The cache-line ping-pong caused by false-sharing is gone, but ping-pong still occurs since the locks still have to move between the different processors cache, as the results show an increase in computation time the more processors are used.

The DCCA has a shared data structure for each connection with a size of 392 byte over 11 different variables. In this case there might not be a need for padding, but instead the locks can be positioned in the data structure so they are not in the same cache-line (see Figure 4.14). Results showed a 1% increase in performance after the new positioning of the shared mutex locks was performed.

```
struct sLOCAL_THREAD_RESOURCE
{
    EmuTel ET;
    int threadId;
    unsigned dwrCounter;
    bool isMaster;
    bool connectionShutDown;
    pthread_mutex_t readMutex;
    pthread_mutex_t writeMutex;
    pthread_mutex_t queueMutex;
    pthread_t* workers;
    sNODE* localNode;
    Queue<Message> messageQ;
};
```

```
struct sLOCAL_THREAD_RESOURCE
{
    pthread_mutex_t readMutex;
    EmuTel ET;
    int threadId;
    unsigned dwrCounter;
    bool isMaster;
    bool connectionShutDown;
    pthread_mutex_t writeMutex;
    pthread_t* workers;
    sNODE* localNode;
    pthread_mutex_t queueMutex;
    Queue<Message> messageQ;
};
```

Figure 4.14 Example of restructuring of shared resource. Structure to the right is restructured.

4.5.2 PROCESSOR AFFINITY

In Linux there is two types of processor affinity. One called soft affinity, which means that the scheduler will try to keep threads on the same processor as long as possible. However, the scheduler could only *try* to achieve this because at some point a long-lived thread will most certainly migrate to another processor. In more recent versions of the Linux kernel, there is support for a second type - hard processor affinity. Hard processor affinity is a kind of scheduler property that forces a thread to a given processor set. The operating system's scheduler will then honor this rule by running the thread on the specified processor(s).

The most obvious benefit of processor affinity in a multithreaded server application is optimizing the cache performance. Multiprocessor computers go through a lot of trouble keeping the processors' caches valid [Gerber-05]. If a thread adds a line of data to it's processor's cache, coherence action must be taken to keep the other processors' caches coherent. And if threads constantly bounce between different processors there is another twist - is the data the thread needs available in the specific processor? Perhaps not, as the data in the cache is data that belongs to another thread, and not the running one. If this occurs in a common case of the application, it could result in a significant cache miss ratio.

To further investigate the impact of processor affinity and cache performance on the DCCA, each connection and it's associated threads were forced onto one processor. Results are shown in Figure 4.15.

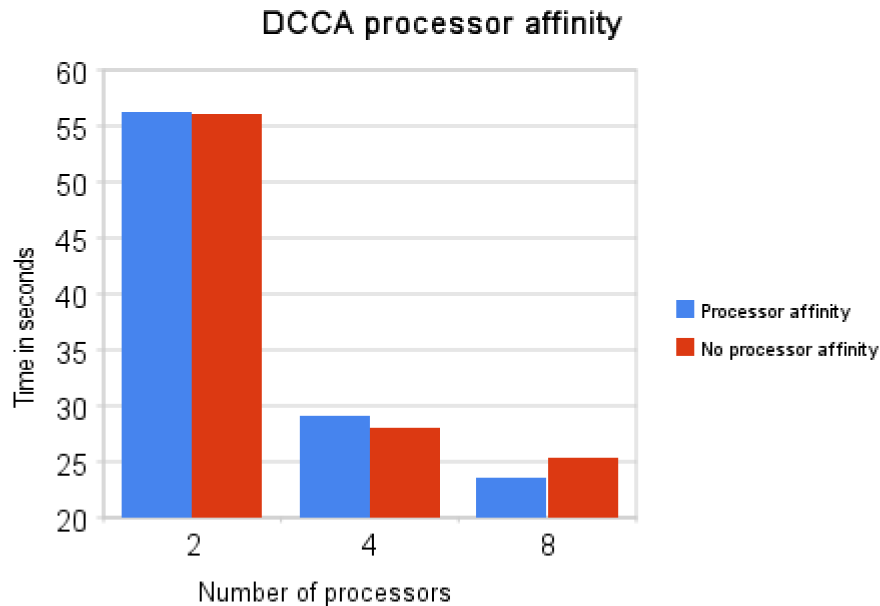

Figure 4.15 Throughput of DCCA, using processor affinity

As the results in Figure 4.15 show, using processor affinity on 2 and 4 processors had a slight performance decrease. However, when using 8 processors there was instead a performance gain when using processor affinity.

A third measurement was conducted, but not included in Figure 4.15 - allowing the connection's associated threads to use a set of processors (instead of just one processor). However, this experiment had no performance impact.

Forcing a specific connection, and all it's associated threads, to one processor has two major disadvantages. If there is load imbalance, i.e. if one connected node sends more credit control requests than the other nodes, that specific hotspot connection will not be able to fully utilize the SMP.

With processor affinity there is also the potential risk that two hotspot connections end up wanting to use the same processor. Such a scenario could mean an even greater performance decrease. Imagine a DCCA scenario where there is only two active connections. Those connections happen to be under heavy load, and both end up being dedicated to the same processor. This turns a parallelized application into a completely sequential one!

A potential solution to the issue with hotspot connections wanting the same processor is to design a forced thread migration. If a processor has a significant amount of work to do, whereas another processor does not, one connection could migrate to that available processor. This alleviates the problem with hotspot connections wanting the same processor, but would still optimize cache performance due to the reason that the connection's associated threads will not run on any other processor. It could, however, introduce new complex issues such as an efficient implementation of such forced thread migration.

If processor affinity is used, careful consideration must be taken in order to properly map connections to processors. As the measurements showed on the DCCA, affinity could have both an increase and a decrease in performance depending on the number of processors. The potential risks of this solution must also be considered.

4.6 MEMORY BLOCK COPYING

Another performance optimization was to use the C function memcpy(), instead of manually copying bytes by iterating through an array. Intel© VTune™ Call Graph pointed out that the DCCA prototype was using this kind of byte concatenation a significant part of the execution time. Each time a Diameter message was received, handled or sent, character byte sequences were put together by concatenating arrays.

The memcpy(void* destination, const void* source, size_t num) function copies the values of *num* bytes from *source* directly to the memory pointed by *destination*. However, memcpy() does not only copy characters. In fact, memcpy() does not care what kind of data is copied, since the result is a binary copy of the data [Cplusplus]. memcpy() can result in a performance increase, since it is already optimized and may use special assembly instructions to copy blocks of data [Bulka-03]. The code blocks in Figure 4.16 are two simple examples of how to concatenate two character arrays. The first example uses memcpy(), and the other uses a user-defined iteration to copy character by character.

```
....
const int SIZE = 26;
char data[SIZE+SIZE+1] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
char newData[SIZE+1] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

// memcpy(destination, source, length)
memcpy(&data[SIZE], newData, SIZE);
....
```

```
....
const int SIZE = 26;
int x = 0;
char data[SIZE+SIZE+1] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
char newData[SIZE+1] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

for(int i=SIZE; i<SIZE+SIZE; ++i)
    data[i] = newData[x++];
....
```

Figure 4.16 - Example of character concatenation using either memcpy() or an iteration.

By using memcpy() the overall performance of the DCCA was increased by 10%, but only up to 4 processors. No improvement was observed when using 8 processors. How memcpy() is primarily used in the DCCA prototype can be seen in Appendix B, Figure B.1.

Chapter

5

DISCUSSION

The main focus of this thesis has been to evaluate the scalability of a Diameter Credit Control Application. The hypothesis was based on the observation that the connections work independent of each other, and therefore the DCCA prototype would scale near linear in respect to the number of processors.

As the results show, the DCCA prototype was not scalable in all cases and especially not considering all the aspects (which were presented in Chapter 4) that ended up affecting scalability. Synchronization and independent connections were only parts of what would affect the performance throughput. If sloppy design decisions were taken, there was an insignificant performance increase when adding more processors.

The parallel efficiency of 2 processors are 75%, which indicates that the performance does not scale linearly. However, the DCCA is considered linear scalable when comparing 2 and 4 processors; both have a parallel efficiency of 75%.

One interesting note is that glibc malloc performed better with *more* synchronization than with less (see differences between Figure 4.7 and Figure 4.8). The exact cause for this is not clear. However, it could potentially be due to glibc malloc's handling of the shared memory heap. If the solution has a higher degree of synchronization, the thread that empties has less competition on who gets access to the memory heap and could therefore complete it's work faster than if more threads on different processors are trying to get the memory synchronization lock.

A lot of other measurements were conducted, but not included in Chapter 4 because the main focus was on multithreading and scalability. However, along with the work of this thesis, other performance evaluations were done.

The impact of inheritance and dynamic binding (this design exist in the Diameter message data structure) were measured. Instead of inheritance, composition and conventional function calls were used. Inheritance and especially dynamic binding have been considered a performance killer due to the compiler not being able to inline the virtual functions [Bulka-03], but the the measurements conducted (general case and DCCA prototype) did not show any significant performance increase or decrease. However, it is important to do further investigation on this matter to fully confirm the potential impact

of inheritance and dynamic binding.

Other C++ specific optimizations were also evaluated and embraced. Most notably the use of initialization lists and call-by-reference (instead of call-by-value) proved to be good optimization techniques. In order to make the common case fast, it was important to reduce the creation and destruction of temporary objects.

The DCCA prototype was only measured and tested in a simulated environment. Besides the benchmarking client, the Diameter traffic generator tool Seagull has been used as a client to generate credit control scenarios. A simulated environment is not an ideal methodology, as in real-world other performance bottlenecks and reliability considerations will most certainly arise. Unfortunately, due to different reasons, Capgemini could not provide a real-world testing environment. However, even if it would have been preferred to have tested the prototype in a real-world environment, the conclusions drawn from this thesis should not be affected by whether the environment is simulated or not.

One could also argue that the DCCA prototype is simply just a prototype, and that it is not representative of a real-world DCCA server. However, even if it is a prototype, the development of the application has been done in close contact with developers at Capgemini Karlskrona. They have given continuous feedback in order to aid the development of an application that is as representative of the real-world as possible. The evaluations and conclusions from this thesis will be an integral part of of their future use of scalable Diameter-based server applications.

If possible, the evaluations and conclusions would probably have benefited from including other platforms than the one used in this thesis (see Chapter 3), since e.g. allocators differ. However, it was not possible with the time given.

For performance critical applications, C++ may be considered a poor choice in contrast to e.g. C. In C, each statement corresponds to a relatively small number of assembler instructions. A good C generalization is that each statement is five to eight instructions. C++ is not that uniform. The cost of C++ statements can fluctuate widely – one statement can generate three instructions, whereas another can generate 300 [Bulka-03]. One must navigate through the performance minefield, trying to avoid the routes that contain the 300-instruction mines.

So why have C++ been the programming language for the DCCA prototype, one may ask. Mainly, it boils down to flexibility and maintenance reasons. C++ has object oriented features that C cannot provide. If used cautiously, these features can give a system design that is stable and that alleviates maintenance efforts. Programming carefully with C++, avoiding heavy C++ construct overheads, could therefore provide the best of the C and C++ worlds – performance, flexibility, stability and a good system design.

The prototype itself is meant to mimic an actual real-world application, and C++ is commonly used within the telecommunication industry, due to the mentioned flexibility and maintenance reasons. It would not be wise to develop a prototype that is hard or even impossible to maintain, as that would not mimic the real-world. C++ is also the programming language that Capgemini requested.

A part of this thesis' work and measurements involved different memory allocators. One of those was TCMalloc, which is an allocator that assigns each thread a cache of it's own and if an allocation needs to be done that is small enough (TCMalloc treats objects with a size of <=32KB as small), then no locks will need to be acquired. That can save approximately 100 nanoseconds for each allocation/deallocation [Ghemawat].

TCMalloc therefore scales well with small object allocations, especially up to 1KB [Ghemawat]. This can be observed on the DCCA prototype, since most allocations are less than 1KB. This results in a major performance increase with TCMalloc compared to glibc malloc (as can be seen in Chapter 4.2). TCMalloc's performance increase compared to glibc malloc decreases the larger the allocations are, which could mean the performance increase observed on the DCCA may not occur with another application. However the size of an allocation in a DCCA conversation rarely exceeds 600 byte and most allocations are much smaller.

Chapter

6

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSIONS

This thesis investigates whether it is possible to develop a Diameter Credit Control Application that achieves linear scaling and the eventual pitfalls that exist when developing a scalable DCCA server. For this reason, a DCCA prototype was developed and evaluated based on throughput. The assumption of achieving linear scalability on the prototype is based on the observation that the different connections have little communication between each other and could therefore utilize every processor to their maximum potential. However, during the work of this thesis it soon became clear that it was not that easy to get a linear speedup by simply adding more processors. There are many factors beside raw processor power that have a major impact on performance. Introducing multiprocessing also introduces a whole set of new problems.

Memory plays a major role and the choice of allocators can play a big part in the performance of a multiprocessor system, as can be seen in Chapter 4.2 where the difference between TCMalloc and glibc malloc was up to a staggering 300% better performance with TCMalloc. General memory issues is a problem. Even small changes like moving variables around in a shared data structure could give noticeable performance increases, (see Chapter 4.5.1) or employing static processor affinity (see Chapter 4.5.2). Due to the impact memory can have on a multithreaded application, it is fundamental to pay attention on how the application handles the shared memory. This is indeed an important lesson to be learned when developing any high performance application.

Careless use of mutual exclusions can also make a huge impact if used improperly, as can be seen in Chapter 4.4 where a different approach increased the performance by up to 150%. Synchronization is a problem, and it is important to be attentive of which parts of the application that have to be executed with mutual exclusion, and which parts that do not.

Unfortunately, even with optimizations in place, the DCCA prototype was not capable of linear scalability, since it had several issues that affected parallel performance (see Chapter 2). The evaluations also point out that careless system design could lead to an insignificant performance increase. That is why awareness of bottlenecks in a parallel server system (such as the DCCA) is fundamental.

It is not wise to take scalability for granted, but instead the parallel performance should constantly be measured and evaluated to see where further optimizations can be achieved. Otherwise, the server could prove to be a potential problem for a telecommunication system, as the demands of applications and services such as the DCCA will most likely increase. In particular since more and more fully-featured cellular phones that have need of AAA hit the market.

6.2 FUTURE WORK

During the work of this thesis, new questions arose that can be of interest. One of those questions is a further investigation of the SCTP protocol. This thesis did not focus on all of SCTP's features, but there are features that could gain a performance increase. Even if multihoming is not thought of as a load-balance feature, it could still be a desirable side effect. An evaluation of the multiple message streams would also be of interest. Perhaps such investigation would also lead to better network reliability.

The DCCA prototype was developed with the standard socket API that comes with Linux. However, there are other socket APIs that could be more performance friendly. APIs that alleviates some of the issues associated with I/O operations, and is designed for a multiprocessor environment. Investigation and evaluation of such APIs could give another lift to the server's scalability.

One last interesting question that came up is the potential use of several physical servers that cooperate as if there is only one. A replicated server approach to gain performance by letting different servers handle different connections. This does not necessarily increase the scalability of a specific server, but it could give better throughput in a DCCA system.

REFERENCES

BIBLIOGRAPHY

[Gerber-05] Gerber R., Bik A. J. C., Smith K., Tian X., *The Software Optimization Cookbook Second Edition. High Performance Recipes for IA 32 Platforms*, Intel Press, 2005

[Bulka-03] Bulka .D, Mayhew D., *Efficient C++*, Addison-Wesley, 2003

[Lin-00] Lin C.., *Principles of Parallel programming*, Pearson Education, 2008

[Grama-03] Grama A., Gupta A., Karypis G., Kumar V., *Introduction to Parallel computing Second edition*, Pearson Addison-Wesley, 2003

[Hennessey-03] Hennessey J., *Computer architecture: A quantitative approach*, Morgan Kauffmann Publishers, 2003

[Robson-02] Robson C., *Real World Research Second Edition*, Blackwell Publishing, 2002,

[Smith-01] Smith C.U., Williams L.G., *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley, 2001

[Berger-00] Berger E.D., McKinley K.S., Blumofe R.D., Wilson P.R., *Hoard: a scalable memory allocator for multithreaded applications*, ACM, 2000

[Ono-08] Ono K., Schulzrinne H., *The Impact of SCTP on Server Scalability and Performance*, Columbia University Technical Report, 2008

WEB RESOURCES

[RFC3588] Diameter Base Protocol
RFC
http://tools.ietf.org/html/rfc3588

[RFC2960] Stream Control Transmission Protocol
RFC
http://www.ietf.org/rfc/rfc2960.txt

[Intel] Avoiding Heap Contention Among Threads
Intel Software Network
http://software.intel.com/en-us/articles/avoiding-heap-contention-among-threads/

[Ghemawat] TCMalloc: Thread-Caching Malloc
Sanjay Ghemawat, Paul Menage
http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[Cplusplus] memcpy
The C++ Resources Network, 2008
http://www.cplusplus.com/reference/clibrary/cstring/memcpy/

GLOSSARY

AAA	Authentication/Authorization/Accounting, refers to the tracking of consumption of network resources by users.
AVP	Attribute Value Pair, is a part of the Diameter message. The actual data of a Diamater message resides in AVPs.
CCA	Credit Control Answer, a common Diameter message in a DCCA.
CCR	Credit Control Request, a common Diameter message in a DCCA.
DCCA	Diameter Credit Control Application, is a Diameter application used for credit control purposes.
Diameter	Is a networking protocol that controls communication between connected nodes.
Diameter message	The communication unit to send a request or deliver answers to connected Diameter nodes.
I/O-operation	Input/Output-operation, e.g. communication via network interface.
IPSec	Protocol suite for securing Internet Protocol (IP).
Memory heap	Is an area of the memory used for dynamic memory allocation.
RADIUS	Networking protocol. Diameter is supposed to be the successor of this protocol.
SCTP	Stream Control Transmission Protocol, a reliable transport protocol.
SMP	Symmetric (shared memory) MultiProcessing. Is a computer system that consists of multiple identical cache-processor subsystems that share the same physical memory and connect via a bus.
STL	Standard Template Library, a C++ standard library which provides e.g. containers such as vector, and algorithms such as sorting.
TCP	Transmission Control Protocol, a reliable transport protocol.
Thread	A single stream of control in the flow of a program. Used to extract parallelism.
XML	eXtensive Markup Language, a markup language for encoding documents.

APPENDIX A - SERIALIZATION POINTS

Serialization point measurements from the DCCA prototype. Figure A.1 represents how a message from a connection is saved in a queue. Figure A.2 represents a solution where the queue synchronization has been alleviated. Figure A.3 represents a solution with a higher degree of synchronization (all of the connections' threads must wait until fileOutputThread is done).

The main difference between Figure A.2 and Figure A.3 is the degree of synchronization.

```
//Push message to file queue.
pthread_mutex_lock(&lr->queueMutex);
lr->messageQ.push_back(m);
pthread_mutex_unlock(&lr->queueMutex);
```

Figure A.1

```
//Unlocked list
void* fileOutputThread(void* ptr)
{
    struct sLOCAL_THREAD_RESOURCE* lr = (struct sLOCAL_THREAD_RESOURCE*)ptr;
    Str toSave;
    Str savePath = lr->localNode->recordPath;
    unsigned sleepTime = lr->localNode->saveInterval;
    unsigned queueSize;

    while(sr->shutDown == false && lr->connectionShutDown == false)
    {
        sleep(sleepTime);

        pthread_mutex_lock(&lr->queueMutex);
        queueSize = lr->messageQ.getSize();

        if(queueSize > 0)
            lr->messageQ.setSize(1);

        pthread_mutex_unlock(&lr->queueMutex);
        queueSize--;

        for(unsigned i=0; i<queueSize; ++i)
            lr->messageQ.dequeue().getXML(toSave);

        toSave.AppendFile(savePath.c_str());
        toSave.Truncate();
    }

    return (void*)0;
}
```

Figure A.2

```
//Locked list
void* fileOutputThread(void* ptr)
{
    struct sLOCAL_THREAD_RESOURCE* lr = (struct sLOCAL_THREAD_RESOURCE*)ptr;
    Str toSave;
    Str savePath = lr->localNode->recordPath;
    unsigned sleepTime = lr->localNode->saveInterval;
    unsigned queueSize;

    while(sr->shutDown == false && lr->connectionShutDown == false)
    {
        sleep(sleepTime);

        pthread_mutex_lock(&lr->queueMutex);//Lock here
        queueSize = lr->messageQ.getSize();

        for(unsigned i=0; i<queueSize; ++i)
            lr->messageQ.dequeue().getXML(toSave);

        lr->messageQ.setSize(0);
        pthread_mutex_unlock(&lr->queueMutex); //Unlock here

        toSave.AppendFile(savePath.c_str());
        toSave.Truncate();
    }

    return (void*)0;
}
```

Figure A.3

APPENDIX B - MEMORY BLOCK COPYING

Using memcpy() function may have a significant performance impact, as discussed in Chapter 4.6. Figure B.1 shows how memcpy is primarily used in the DCCA prototype. String is a user-defined class for handling a char array, i.e. a string. The Add member function adds *len* characters from source *s*, to destination *m_data*.

Using an iteration instead of memcpy is included within C++ comments.

```
// Append len characters from source s, to destination m_data
void String::add(const char* s, long len)
{
    // Verify input
    if(len == 0 || s == NULL)
        return;

    // Find length if not given (default = -1)
    if(len == -1)
        len = strlen(s);

    // Check if array is full
    if(m_len + len >= m_maxLen)
    {
        m_maxLen = m_len + len + 1 + STR_ADDLEN;
        m_data = (char*)realloc(m_data, m_maxLen);
        assert(m_data != NULL);
    }

    // Copy
    memcpy(&m_data[m_len], s, len);
    m_len += len;

    //Copying manually, without memcpy
    /*for(long i=0; i<len; ++i)
     {
         *(m_data + m_len) = *(s + i);
         m_len++;
     }*/

    // Null terminate string
    m_data[m_len] = '\0';
}
```

Figure B.1