Evaluate the benefits of SMP support for IO-intensive Erlang applications

Erisa Dervishi

Master of Science Thesis
Stockholm, Sweden 2012

TRITA-ICT-EX-2012:164

KTH Information and
Communication Technology

KTH Royal Institute of Technology

School of Information and Communication Technology

Degree project in Distributed Computing

Evaluate the benefits of SMP support for IO-intensive Erlang applications

Author: Erisa Dervishi
Supervisor: Kristoffer Andersson
Examiner: Prof. Johan Montelius, KTH, Sweden

TRITA: TRITA-ICT-EX-2012:164

Abstract

In the recent years, parallel hardware has become the mainstream standard worldwide. We are living in the era of multi-core processors which have improved dramatically the computer's processing power.The biggest problem is that the speed of software evolution went much slower, resulting in microprocessors with features that software developers could not exploit. However, languages that support concurrent programming seem to be the solution for developing effective software on such systems. Erlang is a very successful language of this category, and its SMP (Symmetric Multi Processing) feature for multi-core support increases software performance in a multi-core environment. The aim of this thesis is to evaluate the benefits of the SMP support in such an environment for different versions of the Erlang runtime system, and for a very specific target of Erlang applications, the Input/Output-intensive ones. The applications chosen for this evaluation (Mnesia,and Erlang MySql Driver), though being all IO bound, differ from the way they handle the read/write operations from/to the disk. To achieve the aforementioned goal, Tsung, an Erlang-written tool for stressing databases and web servers, is adapted for generating the required load for the tests. A valuable contribution of this thesis is expanding Tsung's functionalities with a new plugin for testing remote Erlang nodes and sharing it with the users' community. Results show that SMP helps in handling more load. However, SMP's benefits are closely related to the application's behavior and SMP has to be tuned according to the specific needs.

Acknowledgment

I first want to thank my program coordinator and at the same time my examiner Johan Montelius for giving me the possibility to be part of this program and guiding me through the whole process. I would also like to thank my industrial supervisor Kristoffer Andersson and all the other guys from the development department at Synapse. They helped me to steer this work in the right direction. We have many times discussed on how to proceed, and they have always been predisposed to answer my questions. Finally, I would also like to thank my fiancé, my family, and my closest friends for all the support (though most of the time online) I have received from them.

Stockholm, 15 July 2012

Erisa Dervishi

Contents

List of Figures

List of Tables

Chapter 1

1 Introduction

A brief introduction of the subject is presented in this chapter. The motivation for the project is given in 1.1, a description of the problem is presented in 1.2, the contribution of the thesis, and some general outcomes are described in 1.3, a short description about the company where the Master s thesis was conducted is given in 1.4, and finally the thesis outline is given in 1.5.

1.1 Motivation

As we are living in the era of the multi-core processors [1], parallel hardware is becoming a standard. The number of processing units that can be integrated into a single package is increasing faithfully to Moore's law [2]. We can say that the hardware is keeping up quite well with the increasing need for applications that require high performance computing and energy efficiency. But what can we say about the programming technologies for developing the softwares? Unfortunately, the answer is that we have a lot of software that is not taking full advantage of the available hardware power. The great challenge nowadays pertains to programmers; they have to parallelize their softwares to run on different cores simultaneously, having a balanced workload on each of them. Obviously, making software development on multi-core platforms productive and easy at the same time, requires not only skills, but also good programming models and languages.

Erlang [3][4][5][6] is a language developed for programming concurrent, distributed, and fault-tolerant software systems. With native support of concurrent programming, Erlang provides an efficient way of software development on many-core systems. In Erlang, programmers write pieces of code that can be executed simultaneously by spawning light-weight processes. These processes are handled by the schedulers of the runtime system and their workload is distributed to different cores automatically. Erlang processes communicate and synchronize with each other only through asynchronous message-passing.

The best part of Erlang is that programmers do not have to think about any synchronization primitive, since there is no shared memory. All the error-prone and tricky synchronization mechanisms that deal with locking and shared memory are handled by the run-time system. Since 2006 the Erlang VM (Virtual Machine) comes with Symmetric Multicore Processing (SMP) capabilities, which help the applications benefit from the multi-core environment without the need of the programmer to write special code for SMP scalability. Erlang SMP features have improved considerably since the first release. There are a lot of benchmarks that show linear scalability of the speed-up with increasing the number of cores. However, these results seem to affect only a limited category of applications, the CPU-bound ones. What about the IO-intensive ones that do heavy read and write operations in the disk? The scope of work for this thesis is to evaluate the Erlang SMP built-in multiprocessor support for IO-intensive Erlang applications in a multi-core environment. The perspectives of interest of this study include the evaluation of the application's performance for different SMP scheduling parameters, in different Erlang/OTP versions and for different characteristics (single or multiple threaded) of database access drivers.

The study results could give some interesting insights about the ability of the Erlang VM to support database-access applications on multi-core platforms.

1.2 Problem Statement

A few years back, the situation was not in favor of Erlang programmers who were developing applications with intensive IO communication with the disk. A lot of factors such as slower disks, single threaded Erlang database-access drivers, older CPU versions, and no SMP support were the cause of IO being the bottleneck. However, nowadays this bottleneck is becoming less disturbing because of the super fast Solid State Disks (SSD-s) and multi-core cpu-s. Erlang itself has improved in this direction by offering toolkits [7] for easily developing Erlang Multi-threaded Drivers, and by including SMP in its recent distributions. The Erlang Developers' community has also contributed with modern Erlang Drivers which can handle pools with multiple connections while communicating with the database. This study aims to evaluate how the IO bottleneck is affected by combining all these innovative hardware and software technologies. Two types of Erlang applications are chosen for the evaluations: Erlang Mnesia Database, and Erlang MySql Driver (Emysql). Both of them perform IO intensive operations, but differ in the way they are implemented.

Mnesia [8] is a distributed DataBase Management System (DBMS), which is written in Erlang and comes with the Erlang/OTP distribution. Emysql is a recent Erlang driver for accessing MySql database from Erlang modules. It offers the

ability to have multiple connections with the database, and handle simultaneous operations.

The major aspect to be evaluated in this thesis is the SMP effects on the performance of these IO-driven applications, measuring meanwhile even the CPU activity. To check if the SMP improvements affect this category of applications, the benchmarks will run in two different Erlang/OTP versions (Erlang/OTP R12 and Erlang/OTP R15). Furthermore, in order to determine whether the application's behavior is general, or bounded to a specific hardware platform, the tests will be running in two different hardware boxes.

In the end, this study should be a good indicator in answering the following questions: Do we get any speedup using SMP for Erlang IO intensive applications? Is this speedup higher in R15 than in R12 (do we get any improvement regarding the IO bottleneck if we upgrade?) What is the benefit of using multiple connection drivers? Compare the results between two different HW platforms (Oracle and HP). Do we lose in performance if we switch to the cheapest one?

1.3 Contribution

This study intends to help companies in a better decision-making. By answering the questions listed at the end of section 1.2, they can decide quite easily if an upgrade to a latter Erlang/OTP version or to a new Erlang DB driver would diminish the IO bottleneck. Tsung [9], is the load generating tool used for the benchmarks' evaluation. It is written in Erlang and is used for stressing databases and web servers. Another valuable contribution of this thesis is expanding Tsung's functionalities with a new plugin for testing remote Erlang nodes and sharing it with the users' community. Results show that SMP helps in handling more load. However, SMP's benefits are closely related to the application's behavior and SMP has to be tuned according to the specific needs.

1.4 Context - Synapse Mobile Networks

The work for this thesis is done at Synapse Mobile Networks, which is a company that develops solutions for Telecoms' operators. Their most successful product is the Automatic Device Management System (ADMS). This system helps to automatically configure the devices of an operator's subscribers [10].The configurations are e.g. enabling MMS, WAP etc., and they are pushed to subscribers. The new devices are capable of data services and remote configuration Over-The-Air, making the operators reduce their costumer care costs, and earn more money at the same time by the increased number of subscribers using their services. They use Erlang OTP R12B as their programming environment (with additional patches from later

OTP versions). The data are stored in the Oracle's Berkeley Database. They are using their proprietary single threaded Erlang Driver for the Berkeley DB access. The database stores information about the subscriber of the Telecoms operator.

1.5 Thesis outline

The report has the following structure:

- *Chapter 2*: This chapter will give an introduction to Erlang, MySql database, Mnesia database, and a short description of different Erlang database drivers. Some related work is discussed and explained by the end of this chapter
- *Chapter 3*: This part starts with a description of the Erlang VM; the evolution of SMP support in Erlang is explained in the second section; finally, some internals on SMP's scheduling algorithms are given in the last section.
- *Chapter 4*: This chapter explains the environment and the experiments setup. The different test-cases are listed in here. It also gives a theoretical explanation on how Tsung and the newly created rpc plugin generate and monitor the load.
- *Chapter 5*: In this chapter test results are shown and evaluated. This chapter reveals some interesting findings regarding SMP's behavior.
- *Chapter 6*: Finally, conclusions together with a short discussion are presented in the last chapter.

Chapter 2
2 Background

2.1 The Erlang System

2.1.1 Introduction

Erlang is a functional programming language that was developed by Ericsson in 1980s. It was intended for developing large distributed, and fault-tolerant Telecom applications [4]. Today, there are many other applications [11] (servers, distributed systems, financial systems) that need to be distributed and fault-tolerant; that is why Erlang, as a language tailored to build such category of applications, has gained a lot of popularity. Erlang differs in many ways from the normal imperative programming languages like Java or C. It is a high level declarative language. Programs written in Erlang are usually more concise, and tend to be shorter in terms of coding lines compared to the standard programming languages. Thus, the time to make the product ready for the market is shortened. On the developers' side, the benefits reside in a more readable and maintainable code.

Furthermore, Erlang, by using the message passing paradigm, makes it possible for its light-weight concurrent processes to communicate with each other without needing to share memory. This paradigm makes Erlang a very good candidate language for software development on a multi-core environment, and offers a higher level of abstraction for the synchronization mechanism compared to the locking one used in other programming languages. Erlang applications can be ported to multi-core systems without change, if they have a sufficient level of parallelism since the beginning.

While Erlang is productive, it is not not suitable for some application domains, such as number-crunching applications and graphics-intensive systems. Erlang applications are compiled to bytecode and then interpreted or executed by the virtual machine (VM). The bytecode is translated into the instructions that can be run on the real machine by the VM. Because of this extra translation step, applications running on a VM are usually slower than other ones that are directly compiled into machine code. If more speed is required, Erlang applications can be compiled into native

machine code with HiPE (High Performance Erlang System) compiler [12]. The readers must keep in mind that Erlang is not always a good choice, especially for some scientific applications which can be both time critical and compute-intensive [13]. A fast low-level language is the best solution in such cases. In other words, Erlang is very powerful when used in the right place, but it is not the solution to every problem.

2.1.2 Erlang Features

Erlang has the following core features [1]:

- Concurrency

  A process in Erlang can encapsulate a chunk of work. Erlang processes are fast to create, suspend or terminate. They are much more light-weight than OS ones. An Erlang system may have hundreds of thousands of or even millions of concurrent processes. Each process has its own memory area, and processes do not share memory. The memory each process allocates changes dynamically during the execution according to its needs. Processes communicate only by asynchronous message passing. Message sending is non-blocking, and a process continues execution after it sends a message. On the other side, a process waiting for a message is suspended until there comes a matching message in its mailbox, or message queue.

- Distribution

  Erlang is designed to be run in a distributed environment. An Erlang virtual machine is called an Erlang node. A distributed Erlang system is a network of Erlang nodes. An Erlang node can create parallel processes running on other nodes in other machines which can use other operating systems. Processes residing on different nodes communicate in exactly the same was as processes residing on the same node.

- Robustness

  Erlang supports a catch/throw-style exception detection and recovery mechanism. It also offers the supervision feature. A process can register to be the supervisor of another process, and receive a notification message if the process under supervision terminates. The supervised node can even reside in a different machine. The supervisor can restart a crashed process.

- Hot code replacement

  Erlang was tailored for telecom systems, which need high availability and cannot be halted when upgraded. Thus, Erlang provides a way of replacing running

[1] `http://ftp.sunet.se/pub/lang/erlang/white_paper.html`

code without stopping the system. The runtime system maintains a global table containing the addresses for all the loaded modules. These addresses are updated whenever new modules replace old ones. Future calls invoke functions in the new modules. It is also possible for two versions of a module to run simultaneously in a system. With this feature any bug fix or software update is done while the system is online.

- Soft real-time

  Erlang supports developing soft real-time applications [2] with response time demands in the order of milliseconds.

- Memory management

  Memory is managed by the virtual machine automatically without the need of the programmer to allocate and deallocate it explicitly. The memory occupied by every process is garbage collected separately. When a process terminates, its memory is simply reclaimed. This results in a short garbage collection time and less disturbance to the whole system [14].

Apart from the above mentioned features, Erlang is a dynamically typed language. There is no need to declare variables before they are used. A variable is single assigned, which means that it is bound to its first assigned value and cannot be changed later. Erlang can share data only by using ETS (Erlang Term Storage) tables [15] and the Mnesia database [16]. Erlang's basic data types are number, atom, function type, binary, reference, process identifier, and port identifier.

Atoms are constant literals, and resemble to the enumerations used in other programming languages. Erlang is a functional programming language, and functions have many roles. They can also be considered as a data type, can be passed as an argument to other functions, or can be a returning result of a function. Binaries are a reference to a chunk of raw memory. In other words a binary is a stream of ones or zeros, and is an efficient way for storing and transferring large amounts of data. References are unique values generated on a node that are used to identify messages.

Process and port identifiers represent different processes and ports. Erlang ports are used to pass binary messages between Erlang nodes and external programs. This programs may be written in other programming languages (C, Java, etc.). A port in Erlang behaves like a process. There is an Erlang process for each port. This port-process is responsible for coordinating all the messages passing through that port.

Beside its basic data types, Erlang provides some more complex data structures, such as tuples, lists and records. Tuples and lists are used to store a collection of items. An item can be any valid Erlang data-type. From a tuple we can only extract

[2] applications that can tolerate some operations to miss their deadlines

a particular element, but a list can be split and combined. Records in Erlang are similar to the structure data-type in C. So, they have a fixed number of named fields.

Modules are the building blocks of an Erlang program. Every module has a number of functions which can be called from other modules if they are exported by the programmer. Functions can consist of several clauses. A clause is chosen to be executed at runtime by pattern matching the argument that was passed to the function. Loops are not implemented in Erlang; recursive function-calls are used instead. To reduce stack consumption, tail call optimization is implemented. Whenever the last statement of a function is a call to itself, the same stack frame is used without needing to allocate new memory.

Finally, Erlang has a large set of built-in functions (BIFs). The Erlang OTP middleware provides a library of standard solutions for building telecommunication applications, such as a real-time database, servers, state machines, and communication protocols.

2.2 Mnesia database

Mnesia [17] is a distributed DataBase Management System (DBMS), tailored for telecommunication or other Erlang applications which require a continuous operation and have soft real-time properties. Thus, beside having all the features of a traditional DBMS, Mnesia additionally fulfills the following requirements:

- Fast realtime key value lookup.
- Complicated non real-time queries mainly for operation and maintenance.
- Distributed data due to distributed applications.
- High fault tolerance.
- Dynamic re-configuration.
- Complex objects.

In other words, Mnesia is designed to offer very fast real-time operations, fault-tolerance, and the ability to reconfigure the system without taking it offline. Mnesia [18, Chapter 17] is implemented in 20,000 lines of Erlang code. The fact that it is so closely integrated with the language makes it powerful in terms of performance and easiness in the development of Erlang applications (it can store any type of Erlang data structure). A common example of Mnesia's benefits in a telecommunications application could be a software for managing the mobile calls for prepaid cards. In such systems calls go on by charging the money for the next few seconds from the user's account. There can be hundreds of thousands of concurrent users' sessions debiting the money from the respective accounts while the calls are progressing. If

the transaction of charging the money cannot occur for some reason, the call has to be canceled for the telecoms company not to lose money. But the customers will not be happy with this solution, and will switch to another operator. For this reason, in such cases of failure, companies allow their subscribers to call for free rather than dismiss the call. This solution satisfies the customer, but is a loss in money for the operator.

Mnesia is perfect for such problems with real-time requirements. It offers fault-tolerance by replicating the tables to different machines which can even be geographically spread. Hot standby servers can take action within a second when an active node goes down. Furthermore, tables in Mnesia are location-transparent; you refer to them only by their name, without needing to know in which node they reside. Database tables are highly configurable, and can be stored in RAM (for speeding up the performance) or on disk (for persistence). It is up to the application's requirements whether to store data in RAM (ram-copies), or on disk (disc-only-copies), or both (disc-copies), or to replicate data on several machines. For the previously mentioned example, in order to handle high transactions per second, you can configure Mnesia to keep a RAM-table.

However, Mnesia has some limitations. Since it is primarily intended to be a memory-resident database, there are some design trade-offs. Both ram-copies and disc-copies tables rely on storing a full copy of the whole table and data in main memory. This will limit the size of the table to the size of the available RAM memory of the machine. On the other hand, disc-only-copies tables that do not suffer from this limitation, are slow (from disk), and the data is stored in DETS [3] tables which may take a long time to repair if they are not closed properly during a system crash. DETS tables can be up to 4Gb, which delimiters the largest possible Mnesia table (for now). So, really large tables must be stored in a fragmented manner.

2.3 MySql database

MySQL is a very successful DBMS for Web, E-commerce and Online Transaction Processing (OLTP) applications.It is an ACID compliant database with full commit, rollback (transaction-safe), crash recovery, and row level locking capabilities. Its good performance, scalability, and ease of use makes it one of the world's most popular open source databases. Some famous websites like Facebook, Google, eBay partially rely on MySQL for their applications.

MySQL handles simultaneous client connections by implementing multi-threading. It has some connection manager threads which handle client connection requests on the network interfaces that the server listens to. On all platforms, one manager thread handles TCP/IP connection requests. On Unix, this manager thread also handles

[3] an efficient storage of Erlang terms on disk only

Unix socket file connection requests. By default, connection manager threads associate a dedicated thread to each client connection. This thread handles authentication and request processing for that connection. Whenever there is a new request, manager threads first check the cache for an existing thread that can be used for the connection, and when necessary create a new one. When a connection ends, its thread is returned to the thread cache if the cache is not full. In the default connection thread model, there are as many threads as there are clients currently connected. This may have some disadvantages when there are large numbers of connections. For example, thread creation and disposal may become expensive, the server may consume large amounts of memory (each thread requires server and kernel resources), and the scheduling overhead may increase significantly .

Since MySQL 5.0, there are 10 storage engines in MySQL. Before MySQL 5.5 was released, MyISAM was the default storage engine. So, when a new table was created without specifying the storage engine, MyISAM would be the default chosen one. The default engine is now InnoDB. A great feature of MySQL is the freedom to use different storage engines for different tables or database schemas depending on the application's logic.

MyISAM is the oldest storage engine in MySQL and most commonly used. It is easy to setup and very good in read related operations (supports full text indexing). However, it has many drawbacks, such as no data integrity check (no strict table relations), no transaction support, and it supports locking only in the table level. The full table lock slows the performance of update or instert queries. MyISAM is not very reliable in case of hardware failure. A process shutdown or some other failure may cause data corruption depending on the last operation that was being executed when the disruption occurred.

On the other hand, InnoDB [19] (the new default engine of MySql) provides transactions, concurrency control and crash recovery features. It uses mutli-version concurrency control with row-level locking in order to maximize performance. With several innovative techniques such as automatic hash indexes, and insert buffering InnoDB contributes in a more efficient use of memory, cpu and disk i/o. InnoDB is the optimal choice when data integrity is an important issue. Due to its design (tables with foreign key constrains), InnoDB is is more complex than MyISAM, and requires more memory. Furthermore, the user needs to spend some time optimizing the engine. Depending on the level of optimization and hardware used, InnoDB can be set to run much more faster than the default setup.

While MySQL is the upper level in the database server which handles most of the portability code for different platforms, communicates with the clients, parses and optimizes the SQL statements, InnoDB is a low-level module used by the upper one to do transaction management, to manage a main memory buffer pool, to perform crash-recovery actions, and to maintain the storage of InnoDB tables and indexes.

The flexible pluggable storage engine architecture of MySQL is one of its major advantages. Both the previously mentioned storage engines (InnoDB and MyISAM) are examples of pluggable storage engines that can be selected for each individual table in a transparent way from the user and the application. Recent performance benchmarks [20] [21] show that in a multi-core environment, MyISAM demonstrates almost zero scalability from 6 to 36 cores, with performance significantly lower than InnoDB. This is the reason InnoDB is the chosen storage engine for this study.

2.4 Erlang Database Drivers

Erlang offers some interfacing techniques [18, Chapter 12] for accessing programs written in other languages. The first and the safest one is by using an Erlang port. The process that creates the port is called the connected process, and is responsible for the communication with the external program. The external program runs in another OS process, outside the Erlang VM. It communicates with the port through a byte-oriented communication channel. The port behaves as a normal Erlang process, so a programmer can register it, and can send messages to it. A crash of the external program is detected by a message sent to the Erlang port, but it doesn't crash the Erlang system. On the other side, if the connected process dies, the external program will be killed.

Another way of interfacing Erlang with programs written in other programming languages is by dynamically linking them to the Erlang runtime machine. This technique is called linked-in drivers. In the programmer's perspective, a linked-in driver is a shared library which obeys the same protocol as a port driver. Linked-in drivers are the most efficient way of interfacing Erlang with programs written in other languages. However, using a linked-in driver can be fatal for the Erlang system if the driver crashes. It crashes the Erlang VM, and affects all the processes running in the system.

Mnesia, as a primarily designed in-memory database, has many limitations (mentioned in section 2.2). For that reason, many other drivers that make it possible for Erlang applications to interact with the most famous databases have been created. There exist some evaluations [22] for DBMS-s for Erlang which include MySQL, PostgreSQL, Berkeley DB, and Ingres as the most suitable databases for Erlang (of course when using Mnesia is not enough). These databases are good candidates since they are very robust, and at the same time they are open source. There have been developed many drivers for connecting Erlang applications to these databases.

Erlang team has come up with Erlang ODBC application[4]. This application provides an Erlang interface to communicate with relational SQL-databases. It is built on

[4] `http://www.erlang.org/doc/man/odbc.html`

top of Microsofts ODBC interface[5], and therefore requires an ODBC driver to the database that you want to connect to. The Erlang ODBC application consists of both Erlang and C code, and can be used to interface any database that can support an ODBC driver. The Erlang ODBC application should work for any SQL-like database (MySql, Postregs, and Ingres) that has an ODBC driver.

Another category of Erlang database drivers is created by using some C libraries that Erlang team provides for interfacing Erlang programs to other systems. These drivers fall in the category of the linked-in drivers; therefore they are loaded and executed in the context of the emulator (sharing the same memory and the same thread). The main C libraries for implementing such drivers are erl_driver[6], and driver_entry[7]. Many versions of Berkeley DB Drivers of this type exist. Synapse has created a single-threaded driver of this category of Berkeley DB Drivers.

Finally, a very successful category of Erlang DB Drivers, is the Erlang native ones. These drivers are implemented in Erlang (no additional non-erlang software is needed), and the communication with the database is done by using Erlang socket programming [18, Chapter 14] in order to implement the wire protocol of the specific database . The most well known native Erlang db drivers are Emysql[8] for accessing MySql Server, and Epgsql[9] for the PgSql Database. These drivers are heavily used recently because they inherit from Erlang the fault-tolerance, and a much simpler concurrency. Moreover, they give a better performance than the other drivers. For example, Epgsql and Emysql drivers implement connection pools in order to keep multiple opened connections to the respective databases, and handle more concurrent users. Compared to Erlang ODBC drivers which require the appropriate ODBC driver code installed on your platform, the native ones do not require additional software. And in contrary to Linked-drivers, they do not crash the VM when something goes wrong. Emysql is one of the drivers chosen for this study, and will be explained in more details in section 4.3.2.

2.5 Related work

The first stable version of SMP was first released in 2006 with Erlang OTP R11B. This makes SMP quite a new feature of Erlang and the only evaluations come from the Erlang team itself, and some master's thesis reports. "Big Bang" is one of the benchmarks used for comparing the previous version of SMP(one run-queue) with the current one (multiple run-queues). The benchmark spawns 1000 processes. Each

[5] ODBC (Open Database Connectivity) is a standard C programming language interface for accessing DBMS-s. An application can use ODBC to query data from a DBMS, regardless of the operating system or DBMS it uses.

[6] `http://www.erlang.org/doc/man/erl_driver.html`

[7] `http://www.erlang.org/doc/man/driver_entry.html`

[8] `https://github.com/Eonblast/Emysql`

[9] `https://github.com/wg/epgsql`

process sends a 'ping' message to all other processes and answers with a 'pong' message for all 'ping' it receives. Results [23] show that the multiple run-queue version of SMP improves performance significantly. A detailed explanation on how SMP works internally, and some other benchmark evaluations of SMP on many-core processors can be found in Jianrong Zhang master's thesis [24]. He uses four benchmarks for his evaluation: Big Bang, Mandelbrot set calculation, Erlang Hackbench, and Random. All these benchmarks fall either in the category of CPU-intensive applications, or in the category of memory-intensive ones. The results show that the CPU-intensive ones scale very well in a multi-core environment with SMP enabled. In contrary, for the applications that require a big memory footprint, there is more lock contention for the memory allocation. Therefore, they scale poorly with the increase of the number of SMP schedulers. However, these evaluations do not consider IO-bound applications. The focus of this thesis is to limit the experiments only for this category of Erlang applications, and analyze their behavior in a multi-core environment for different SMP implementations and parameters.

Chapter 3
3 SMP

3.1 Inside the Erlang VM

BEAM[1] is the standard virtual machine for Erlang. The first experimental implementation of the SMP VM occurred in 1998 as a result of a master degree project [25]. Since 2006, the SMP VM is part of the official releases. The SMP Erlang VM is a multithreaded program. POSIX thread (Pthread) libraries are used for the SMP implementation in Linux. Threads in OS processes share the memory space. Ports and processes inside an Erlang VM are scheduled and executed by an Erlang scheduler which is a thread. The scheduler has both the role of a scheduler and a worker. Processes and ports are scheduled and executed in an interleaving fashion.

An Erlang process contains a control block (PCB), a stack and a private heap. A PCB is a data structure that contains process management information, such as process ID (IDentifier), position of stack and heap, argument registers and program counter. There can also be some other small heap fragments which are merged into the main heap after each memory garbage collection. These heap fragments are used when the Erlang process requires more memory, but there is not enough free memory in the heap, and the garbage collection cannot be performed to free memory. Binaries larger than 64 bytes and ETS tables are stored in the common heap which is shared by all processes. Figure 3.1 shows these main memory areas. The stack and heap of an Erlang process are located in the same continuous memory area. This area is allocated and managed for both the stack and heap. In terms of an OS process, this common memory area belongs to the OS heap, so the heap and stack of an Erlang process belongs to the heap of the Erlang VM. In this memory allocated area, the heap starts at the lowest address and grows upwards, while the stack starts at the highest address and grows downwards. Thus a heap overflow can be easily detected by examining both the heap's and stack's top. The heap is used to store compound data structures such as tuples, lists or big integers. The stack is used to store simple

[1] Bogdans/Bjorn's ERLANG Abstract Machine

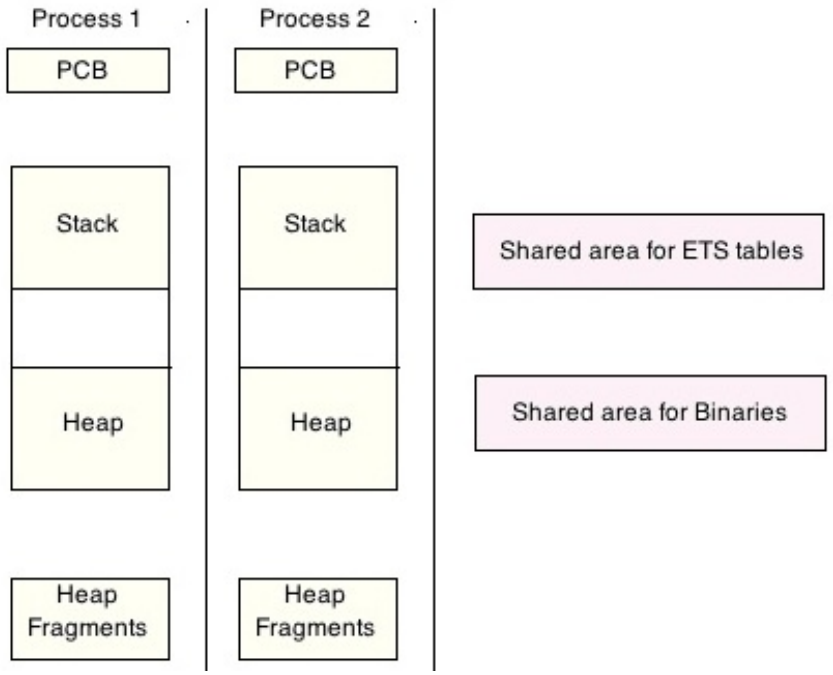

Figure 3.1: Memory structure for Erlang processes

data and pointers to compound data in the heap. There are no pointers from the heap to the stack.

In order to support a large number of processes, an Erlang process starts with a small stack and heap. Erlang processes are expected to have a short life and require a small amount of data. However, when there is no free memory in the process heap, it is garbage collected. If the freed memory is still less that the required one, the process size grows. Garbage collection is done independently for each process.

Since each process has its private heap, messages are copied from the sender's heap to the receiver's one. This architecture causes a high message-passing overhead, but on the other hand, garbage collection disturbs less since it is done in each process independently. When a process terminates, its memory is reclaimed.

3.2 Evolution of SMP support in Erlang

The first stable release of Erlang SMP was included in Erlang OTP R11B in May 2006. In March 2007, it began to run in products with with a 1.7 scaling on a dual-

core processor. The first commercial product using SMP Erlang was the Ericsson Telephony Gateway Controller.

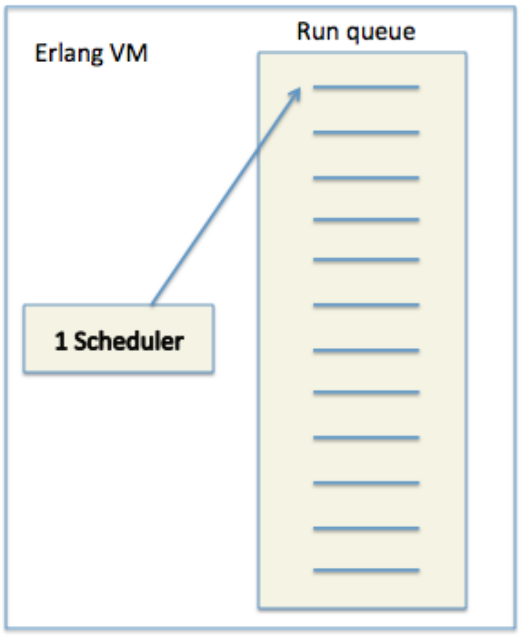

Figure 3.2: Erlang non SMP VM

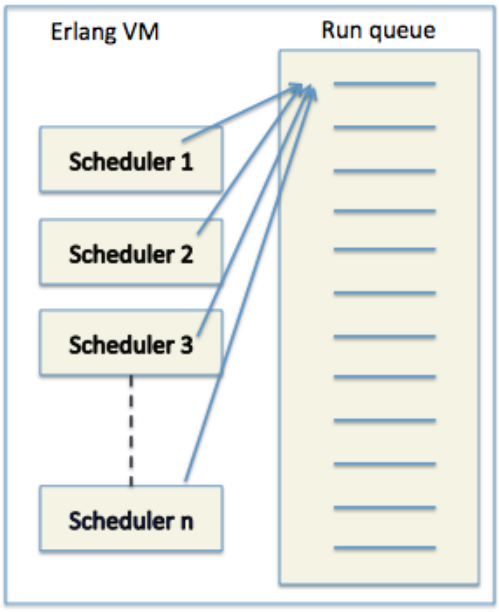

Figure 3.3: Erlang SMP VM (before R13)

As illustrated in figure 3.2, Erlang VM with no SMP support had one Scheduler and one run-queue. The jobs were pushed on the queue and fetched by the scheduler. Since there was only one scheduler picking up the processes from the queue, there was no need to lock the data structures.

As can be seen in figure 3.3, the first version of SMP support included in R11B and R12B contained multiple schedulers and one run-queue. The number of schedulers varied from one to 1024, and every scheduler was run in a separate thread. The drawback of this first SMP implementation is that the scheduler picks runnable Erlang processes and IO-jobs from the only one common run-queue. In the SMP VM all shared data structures, including the run-queue, are protected with locks. This makes the run-queue a dominant bottleneck when the number of CPUs increases. The bottleneck will be visible from four cores and upwards. Furthermore, ets tables involve locking. Before R12B-4 there were 2 locks involved in every access to an ets-table, but in R12B-4 the locking was optimized to reduce the conflicts significantly. Performance dropped significantly when many Erlang processes were accessing the same ets-table causing a lot of lock conflicts. The locking was on table-level, but in later versions a more fine granular locking was introduced. Since Mnesia uses heavily ets-tables, the locking strategy impacts Mnesia's performance directly.

The next performance improvement related to SMP support in the Erlang runtime

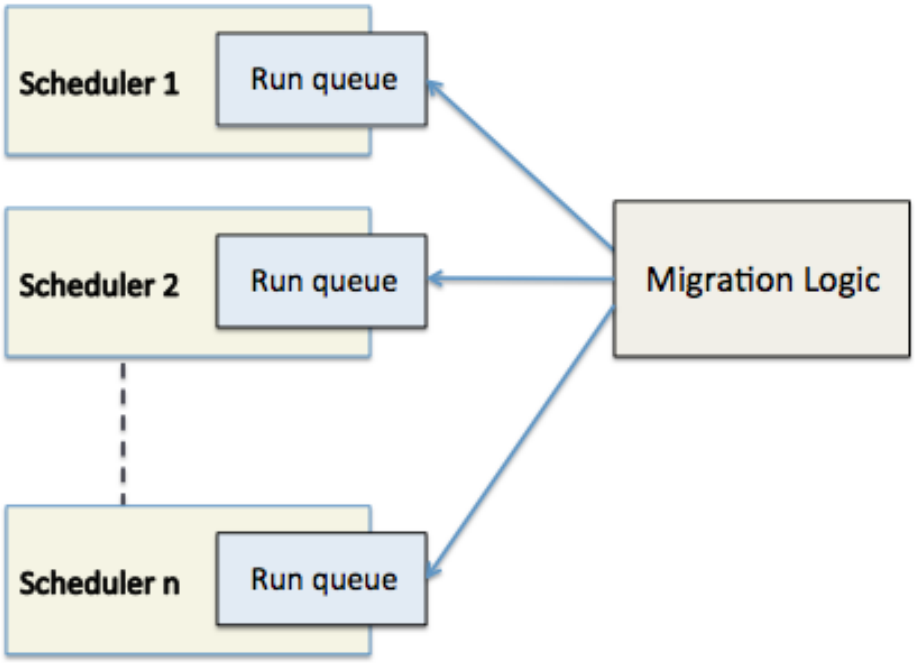

Figure 3.4: Memory structure for Erlang processes

system was the change from having one common run-queue to having a separate run-queue for each scheduler. This new implementation is shown in figure 3.4, and it was introduced in R13. This change decreased the number of lock conflicts for systems with many cores or processors.However, with separate run-queues per scheduler the problem was moved from the locking conflicts when accessing the common run-queue to the migration logic of balancing the processes in different run-queues. This process has to be both efficient and reasonably fair. On multi-core processors, it is a good practice to configure Erlang VM with one scheduler per core or one scheduler per hardware thread if hardware multi-threading is supported.

Processes in Erlang communicate with each-other through message passing. Message passing is done by copying the message residing on the heap of the sending process to the heap of the receiving one. When sending a message in Erlang SMP, if the receiving process is executing on another scheduler, or another message is being copied to it by another process, it cannot accommodate this message. In such a case, the sending process allocates a temporary fragment of memory from the heap on behalf of the receiving process, and the message is copied there. This heap fragment is merged into the private heap of the receiving process during garbage collection. After the sent message is copied, a management data structure that contains a pointer to the actual message is put at the end of the receiving process message queue. If the receiving process is suspended, it is woken up and appended

to a run-queue. In the VM with SMP implemented, the message queue of a process consists of two queues, the private and the public one. The public queue is used for other processes to send their messages, and is protected by mutual exclusion locks. The private queue is used by the process in order to reduce the lock acquisition overhead. First, a process looks for a matching message in its private queue. If it cannot find the matching message there, the messages are removed from the public queue, and are appended in the private one. In the non-SMP VM there is only the private queue.

3.3 Erlang VM Scheduling

There are four work categories that are scheduled in the Erlang VM, process, ports, linked-in drivers, and system-level activities. The system-level work includes checking I/O tasks such as user input in the Erlang terminal. As stated in section 2.4 linked-in drivers are a mechanism for integrating external programs written in other languages into Erlang. While with normal port the external program is executed in a separate OS process, as a linked-in driver it is executed as a thread in the same OS process as the Erlang node. The following description about the scheduling mechanism is focused on scheduling processes.

The method the Erlang schedulers use for measuring the execution time is based on reduction counting. Reductions are roughly similar to function calls. Depending on the time it takes to make a function call, the period can vary between different reductions. A process that is scheduled to run has a predefined number of reductions that it is allowed to execute. The process continues executing until this reduction limit is reached, or pauses to wait for a message. A process in a waiting state is rescheduled when a new message arrives or when a timer expires. New or rescheduled processes are always put at the end of the respective run-queues.Suspended processes are not stored in the run queues.

Processes have four priorities: maximum, high, normal and low. Every scheduler has one queue for the maximum priority and another one for the high priority. Processes with the normal and low priority reside in the same queue. So, the run queue of a scheduler, has three queues for processes. There is also a queue for ports. In total, a scheduler's run queue has four queues that store all the processes and ports that are runnable. The number of processes and ports in all these queues is denoted as the run-queue length. Processes of the same priority queue are executed with the round-robin algorithm. Thus, equal period of time (here a number of reductions) is assigned to each process in a circular order.A scheduler chooses processes from the queue with the maximum priority to execute until this priority queue is empty. Then it does the same for the queue with the high priority. The next processes to be executed are the normal priority ones. Since the normal and low priority processes

reside in the same queue, the priority order is maintained by skipping a low priority process a number of times before its execution.

Workload balancing on multiple processors or cores is another important task of the schedulers. The two mechanisms that are implemented are the work sharing and stealing. The workload is checked periodically and shared equally. Work stealing is applied inside a period in order to further balance the workload. In every check period only one of the schedulers will analyze the load of each scheduler. This load is checked from an arbitrary scheduler when its responsible counter reaches zero. The counter in each scheduler is decreased whenever a number of reductions are executed by processes on that scheduler. In every balance check this counter is reset to its initial value, and the time it takes for this counter to reach value zero, is the time between two work balance checks. If a scheduler has executed the number of reductions required to do a balance check (its counter is equal to zero), but finds out that another scheduler is doing the check, than it will skip the check and its counter will be reseted. In this way we are sure that only one scheduler is checking the work load in all the other ones. The number of schedulers is configured when starting the Erlang VM. Its default value is equal to the number of logical processors in the system. There are different settings for binding the schedulers' threads to the cores or hardware threads. Moreover, users can also determine the number of the online schedulers when starting a node. This number can be changed at runtime because some schedulers may be put into an inactive state when there is no workload. The number of active schedulers in the next period is determined during the balance checking. It can increase if some inactive schedulers are woken up because of high workload, or decrease if some schedulers are out of work and in the waiting state.

Another important task of the scheduler checking the load is also to compute the migration limit. The migration limit, is setting the number of processes or ports, for each priority queue of a scheduler based on the system load and availability of the queue of the previous periods.Migration paths are established by indicating which priority queues should push work to other queues and which priority queues should pull work from other queues. After these relationships are settled, priority queues with less work will pull processes or port from their counterparts, while priority queues with more work will push tasks to other queues. Scheduling activities are interleaved with time slots for executing processes, ports or other tasks. When some schedulers are inactive because of low load, the work is mainly pushed by them to the active schedulers. Inactive schedulers will become standby when all their work is pushed out. However, when the system is fully loaded and all available schedulers are active, the work is generally pulled by the schedulers with less workload.

Figure 3.5 is a simple example of the migration limit calculation. We assume there are only processes with the normal priority like in the normal case. Then the calculation of migration limit is just the averaging operation of the length of the run-queues. In this example the migration limit is equal to 14.

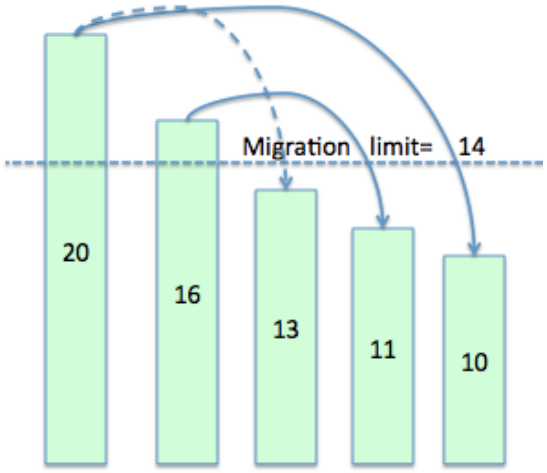

Figure 3.5: Migration path

After the migration limit calculation, the next step is to determine the migration paths. A migration path shows which priority queue needs to transfer tasks to another queue. This is done by subtracting the maximum queue length of each scheduler by the migration limit. If the result is positive, then the queue will have to push its work; otherwise, when the result is negative, the queue has less work than the limit and is a candidate for pulling work from another queue. The queues of the same priority are sorted by the subtraction results, and the migration path is set between the queue with the least negative difference and the one with the largest positive difference. Following the same logic, another migration path is set between the queues with the second largest and smallest difference, and so on. There are two types of flags that are set on the queues for defining their migration strategy. The emigration or push flag is set on queues with a positive subtraction difference, and the immigration or the pull flag is set on queues with a negative difference. It is also set the target or the source for the emigration or immigration respectively. There is only one target or source for each queue, and a queue is either pushing or pulling, but not both. As illustrated in the example in figure 3.5, the number of queues with positive differences can be different from the number of queues with negative differences. In such cases, if there are more emigrating queues, the emigration flag is set in all of them, and their target queues are set starting from the queue with the least negative difference. There may be more than one queue pushing work to a queue, but the pulling queue has only one source for immigration. In the case of more immigrating queues, the immigration flags are set on the additional immigrating queues, and the immigration sources are set starting from the queue with the largest positive difference. For this reason, it can be more than one queue pulling work from a queue, but the corresponding pushing queue has a single emigration target.

In figure 3.5 there are more pulling queues. Both queues with the maximum length

13 and 10 pull work from the queue with the maximum length 14, but only the queue with the length 10 is set as the target for the emigrating queue. Maximum queue length is a value that belongs only to a period. It does not mean that the run queue has that number of processes and ports when the balance check is done. After the establishment of the migration paths, in every scheduling slot a scheduler checks the migration flag of its priority queues. If immigration flags are set, the scheduler pulls processes or ports for its priority queues with an immigration flag set from the head of the source queues. If the emigration flags are set, the responsible scheduler does not push tasks repeatedly. The emigration flag is checked only when a process or a port is being added to a priority queue. If this queue has an emigration flag set, the process or the port is added to the end of the migration's target queue instead of the current queue.

If an active scheduler has no work left, but it cannot pull work from another scheduler any more, it tries to steal work from other schedulers. If stealing does not succeed, and there are no system-level activities, the scheduler thread changes its state into waiting. It is in the state of waiting for either system-level activities or normal work. In normal waiting state it spins on a variable for some time waiting for another scheduler to wake it up. If this time expires with no scheduler waking it up, the spinning scheduler thread is blocked on a conditional variable. A blocked scheduler thread takes longer time to wake up. Normally, a scheduler with high workload will wake up another waiting scheduler either in a spinning or blocked state.

Chapter 4

4 Proposed architecture for the evaluations

4.1 Software and hardware environment

The experiments of this study were run in two different environments, a HP server and an Oracle one. Since the applications chosen for the tests demonstrated almost the same behavior in both the hardware platforms, in this report we are only presenting the results of the Oracle server.

Target server details:

- Operating System: Oracle Solaris 10 X86
- Mysql Database Version: 5.5.23
- Erlang installed versions: R12B and R15B
- Hardware details are listed in table 4.1

Hardware parameters	**Values**
Model	Sun-Fire X4170
CPU	2 X Intel Xeon L5520 2.27 GHZ (16 cores)
Memory	6 X 4 GB
Disk	4 X 136 GB

Table 4.1: Hardware Platform

4.2 Tsung

One of the purposes of this thesis was to automate the evaluation process by using, adapting or building an evaluation tool. After doing a study on existing Erlang tools, according to a survey on Erlang testing tools [26] Tsung was amongst the most

used tools for load and stress testing. Thus, Tsung was the chosen candidate for the purpose of this study.

Tsung is an open-source multi-protocol distributed load testing tool. It can be used to stress HTTP, WebDAV, SOAP, PostgreSQL, MySQL, LDAP and Jabber/XMPP servers. It is a free software released under the GPLv2 license. Its purpose is to simulate users in order to test the scalability and performance of client/server applications. Many protocols have been implemented and tested, and it can be easily extended. Tsung is developed in Erlang; thus it offers the ability to be distributed on several client machines and is able to simulate hundreds of thousands of virtual users concurrently The following subsections list the most important functionalities of Tsung together with the new Tsung plugin implemented excursively for achieving the goals of our evaluations.

4.2.1 Tsung functionalities

Tsung's main features include:

- High Performance

  Tsung can simulate a huge number of simultaneous users per physical computer. Furthermore it offers the ability for a simulated user to not always be active; it can also be idle in order to simulate a real user's think-time.

- Distributed

  The load can be distributed on a cluster of client machines

- Multi-Protocols

  It uses a plug-in system, and its functionalities can be further expanded by adding new plugins. The currently supported plugins are the HTTP (both standard web traffic and SOAP), WebDAV, Jabber/XMPP and PostgreSQL. LDAP and MySQL plugins were first included in the 1.3.0 release.

- SSL support

- OS monitoring

  CPU, memory and network traffic are monitored using Erlang agents on remote servers or SNMP

- XML configuration system

  An XML configuration file is used to write complex user scenarios. Furthermore, scenarios can be written even by using the Tsung recorder (HTTP and PostgreSQL only) which records one or more sessions and helps in automating the scenario-writing process.

- Dynamic scenarios

  The user can get dynamic data from the server under load (without writing any code) and re-inject these data in the following requests. A session can be manipulated (stopped, restarted or looped) whenever a regular expression set by the user matches a server response.

- Mixed behaviors

  Multiple sessions can be used to simulate different users' behavior during the same benchmark. The duration and other characteristics of each session are set in benchmark scenario.

- Stochastic processes

  Tsung simulates think-times and randomized arrival rates by using a probability distribution function (currently exponential). Thus, it can generate a realistic traffic.

The measurements and statistics produced by Tsung are all represented as graphics. Tsung produces performance statistics such as response time, connection time, requests per seconds. Furthermore, it collects statistics on page return code in order to trace erros. Most importantly, it reveals important information about the target-server behavior. Erlang agents are started on target servers in order to gather the required information. SNMP and munin is also supported to monitor remote servers. However, for our evaluations we monitor the target servers with Erlang agents. In the collection of graphs generated by Tsung are also included data about CPU, memory consumption and network traffic. It is possible to generate graphs during the benchmark as statistics are gathered in real-time.

4.2.2 Rpc Tsung plugin

Tsung, being implemented in Erlang, is an injection tool that offers high performance and distributed benchmark. However, being protocol based, it uses different plugins for testing different types of servers. In our case, in order to evaluate the Erlang SMP support, our aim was to load and test the Erlang modules responsible for doing the I/O to the database server, not the servers themselves. Tsung could not offer such a functionality, so we came up with the idea of extending Tsung with a new plugin for testing Erlang nodes specifically. For instance, for testing a MySql Server, Tsung uses the ts_mysql plugin. This plugin offers four types of requests such as connect, authenticate, sql, and close. The generated Tsung requests are sent to the server through TCP socket connection. Then, Tsung collects statistics about the responses. The load Tsung generates can be distributed to several clients, but there is only one Erlang node that is started in the machine where Tsung starts which is responsible for the whole process. This node is called the Tsung controller. Tsung controller starts Erlang slave-nodes in other clients for generating heavy load, and in

the target server (if Erlang monitoring is chosen) for collecting performance results. Thus, we decided to use the Erlang remote communication, in order to send the requests from the client Erlang nodes to the Erlang node running the application in the server. Erlang uses the rpc module [1] for sending the load requests to the server node.

The newly implemented Erlang plugin is called ts_rpc. As mentioned in section 4.2.1 Tsung operates based on an XML configuration file containing the whole benchmark scenario. The xml attributes of this configuration file are defined in a dtd [2] file called tsung-1.0.dtd. We extended this file with new attributes for the new plugin. First, the "type" attribute of the "sessions" element was expanded with the name of the new plugin (ts_rpc). We also added the rpc tag to the request element:

```
<!ELEMENT request ( match*, dyn_variable*,
( http | jabber | raw | pgsql | ldap | mysql |fs | shell | job | rpc) )>
```

Additionally, we created a new element called rpc, with the following attributes:

```
<!ELEMENT rpc (#PCDATA) >
 <!ATTLIST rpc
     module       CDATA   #IMPLIED
     function     CDATA   #IMPLIED
     arguments    CDATA   #IMPLIED
     node         CDATA   #IMPLIED
     type         (cast | call) #REQUIRED >
```

We use these attributes in the XML configuration file in order to pass to Tsung the request's parameters. These parameters include the remote server node name where the load is sent, the erlang module that is called, the function to be executed and its arguments. Basically, the rpc xml-element contains all the parameters needed for performing an Erlang rpc request. Two more Erlang modules are added to Tsung's source code. The first one (ts_config_rpc.erl) is responsible for parsing the rpc part of the XML file. The second file (ts_rpc.erl), builds the actual data to be transmitted to the server.

With the new rpc plugin, we can use Tsung to generate load to any Erlang node, and test any type of database server that this Erlang node is connected to. The user only has to create the xml config file, and start the Erlang remote node that he wants to evaluate prior to starting Tsung. We are confident this new plugin can be a valuable

[1] This module contains services which are similar to remote procedure calls. A remote procedure call is a method to call a function on a remote node and collect the answer

[2] A Document Type Definition (DTD) file is a set of markup declarations that define a document type for markup languages such as XML, HTML, etc.

feature for Tsung's users community. It is important to mention that Tsung uses SSH to spawn Erlang remote nodes, so the Tsung controller node and the remote node have to be able to communicate with each-other both the ways.

4.3 MySQL experiments setup

This section describes the setup of the first set of experiments which include Emysql driver and MySql Server.

4.3.1 Mysql Database

For the purpose of the experiments we have created one database with only one table. This table is named subscriber and its fields store the same information Synapse stores for their subscribers. MySql server was tuned in order to give the best performance. Many InnoDB-related configuration parameters were changed in the my.cnf [3] file to maximize the benefits gained from the current hardware parameters. Generally, the most important parameters that affect the server's performance are related to buffer pool size, thread concurrency, logs' flush-method, logs' buffer size, etc. Detailed information about tuning Mysql's performance can be found in the MySQL 5.5 Reference Manual[4]

4.3.2 Emysql driver

Emysql driver was created from scratch in 2009 for achieving a better stability and throughput. It is an Erlang written application for communicating with MySql server. This driver is easy to use, and supports the execution of prepared statements and stored procedures. It is optimized for a central node architecture and OLTP[5]. A valuable feature of Emysql is the connection pool [6] Emysql Driver can handle multiple pools of multiple connections each to multiple database servers. The common usage of the driver is through a single Erlang node that serves as the MySQL communication point. It allows a number of connected nodes to connect through it to MySQL.

[3] MySQL's configuration file which stores default startup options for both the server and for clients. It is very important to have a correct configuration of this file in order to optimize MySQL

[4] `http://dev.mysql.com/doc/refman/5.5/en/innodb-configuration.html`

[5] Online transaction processing (OLTP) includes those systems that facilitate and manage transaction-oriented applications which perform data entry and retrieval transaction processing

[6] A connection pool is a cache of database connections that can be reused when future requests to the database are required. Connection pools are used to enhance the performance of executing commands on a database

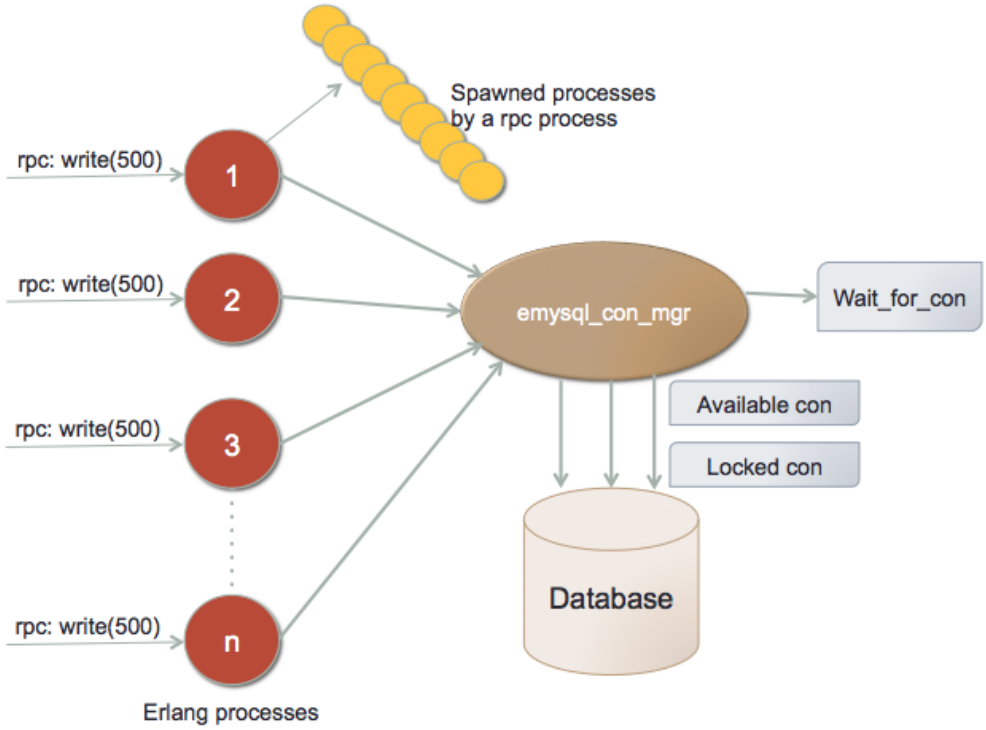

Figure 4.1: Emysql internal structure

For our experiments we created an Erlang benchmark module with the following main functions:

- run: Starts the Emysql application, and creates a connection pool of 50 connections to Mysql Database with the database-user credentials and the database name.
- write(N): This function writes N records to the subscriber's table
- read(N): This function reads N random records from the subscriber's table.

As illustrated in figure 4.1, per each Tsung rpc request, an Erlang process is created in the target server where both the Emysql driver and the Mysql database are running. The main process of Emysql driver is the Emysql connection manager (emysql_con_mgr). This process is responsible for maintaining the connection pool. It manages two lists, the available connections list and the locked ones. The available connections list stores the free connections to the database, and the locked connections list stores the connections that are already in use by other processes. The example in figure 4.1 shows rpc requests with write(500) (insert 500 new records to the database) as a parameter. Thus, the process responsible for the request in the target server, first contacts the emysql_con_mgr process for an available connection. If there are

any free connections in the available connections list, it takes one connection and spawns a new worker process for doing the insert to the database. Otherwise, the request process is added to the wait_for_con (wait for connection) queue until a locked connection is freed. This procedure is repeated for the 500 inserts of the request.

Since one of the goals of this thesis was to evaluate the SMP enhancements from R12 to R15, the emysql driver was compiled in both these Erlang versions. Emysql is quite a recent driver, and there were some compatibility issues when compiling it back in R12, but we succeeded in doing that with some minor changes. However, we are confident these changes do not make unfair the comparison between the two versions of the driver.

4.4 Mnesia experiments setup

Mnesia database is created with the disc-copies option. This means that the tables are persistently stored in disk, but loaded in RAM whenever Mnesia is started. The same logic as for Mysql setup is followed even in here. There is only one table named subscriber. For our experiments we created an Erlang benchmark module with the following main functions:

- start: Starts Mnesia and loads the database in memory.
- write(N): This function writes N records to the subscriber's table
- read(N): This function reads N random records from the subscriber's table.

Mnesia benchmark modules are compiled in both R12 and R15. Since Mnesia DB is all loaded in memory, whenever the load increases, a warning appears notifying that Mnesia is overloaded. To minimize the frequency this warning is displayed, and increase the load Mnesia can handle, we change two Mnesia related parameters whenever we start the remote server node. The first one is dump_log_write_threshold. This variable defines the maximum number of writes to the transaction log before a new dump is performed. The default value is 100, so a new transaction log dump is performed for every 100 writes. If Mnesia is handling hundreds or thousands of writes in a short time, it cannot keep up; thus, increasing this value when there is enough RAM helps in handling more load. The second variable we change is named dc_dump_limit. It controls how often disc_copies tables are dumped from memory.The default value is four, so if the size of the log is greater than the size of table / 4, then a dump occurs. Increasing this value makes tables dump from memory more often. Furthermore, in order not to run out of memory, we make sure that the number of inserted records is not more than seven millions.

4.5 Test cases

This section explains the experiments setup, and the test-cases included in our performance evaluations. As listed in table 4.2, experiments include two different databases such as MySql and Mnesia.

Test-cases	
Database	MySql Mnesia
DB Operations	Writes Reads
OTP Versions	R12B01 R15B
SMP	Auto 1 scheduler Disabled

Table 4.2: Parameters included in the experiments setup

The operations that are used for the performance evaluation are database read and write requests. The test-cases do not include a mix of read and write operations; there are only read-only test cases, and write-only ones. Tests are run on two different Erlang OTP versions, R12B01 and R15B. In the remaining of this report, we will shortly refer to these versions as R12 and R15 respectively. The Erlang VM in both the versions is started with 3 different options for SMP. The first one is SMP auto which means that the number of schedulers is equal to the number of cores. The second option is only one scheduler. At last, the disabled option means that there will not be used SMP at all Basically, a test case is a combination of a database type, a database operation, an OTP version and a SMP parameter. The Erlang VM that is started in the remote servers has to consider these parameters when it is first initialized. A configuration script is built for automating this initialization. The script takes as parameters the Erlang node name to be started, the OTP version, the SMP features, and the paths to the related Erlang modules needed for the benchmark. In addition, the script tells to the newly created Erlang node to initialize the modules that do the access to the database.

Figure 4.2 shows the underlying architecture of the experiments. On server's side, the node to be tested is started by passing the test-case's parameters to the configuration script. After the node is started, it automatically initializes the corresponding modules responsible for the DB access. Tsung is run locally on the client's side. When it is started, it takes as a parameter the xml configuration file. An xml file represents a single test-case. It gives Tsung information about which plugin to use (the rpc plugin in our case). The duration of the test-case is also specified in there. Our test-cases have a duration of 12 minutes. Another setting stored in the xml file

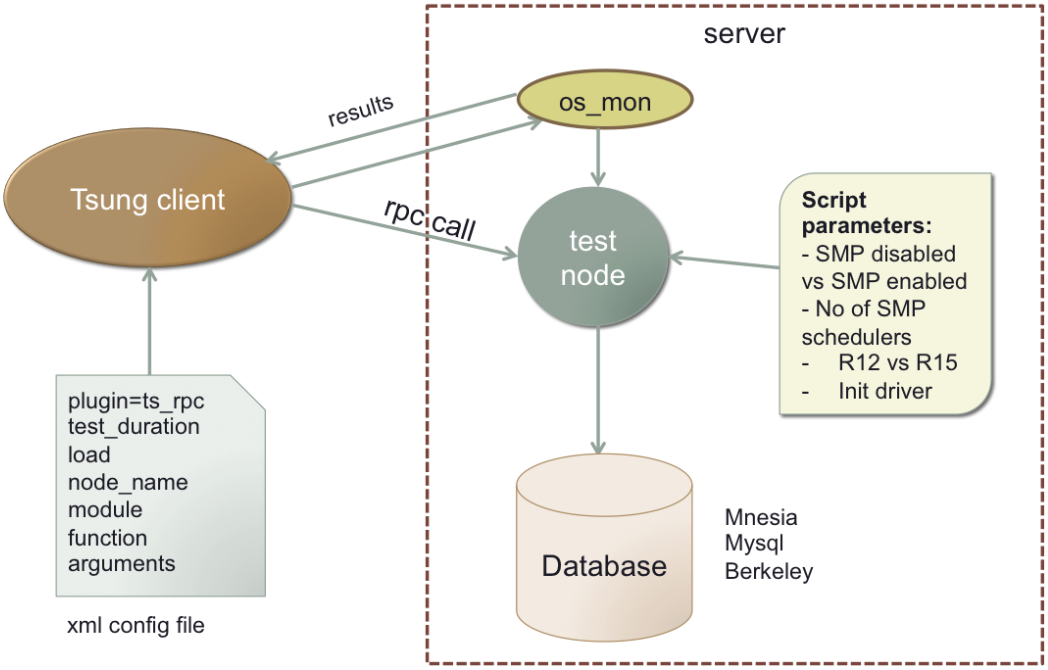

Figure 4.2: Experiments Setup

is the load that has to be simulated by Tsung. It is specified as time between two consecutive requests. We start with a low load, and start increasing it gradually until the server saturates. Additional parameters that are configured in the xml file are related to the rpc request generated by Tsung. We specify the server node name, which must be the same as the node name started by the initial script on server's side. Moreover, we give information about the module, function and arguments of the rpc request. An rpc request is generated as often as the user inter-arrival rate specified in the load section. For example, for a write test-case in MySql, this part of the xml file would look like:

```
<request>
  <rpc type="call" module="mysql_benchmark" function="write"
       node="erisa@dcr1b" arguments="500" >
  </rpc>
</request>
```

This code snippet tells to Tsung plugin to send a rpc call to the remote node "erisa@drc1b", which executes the function write(500) from module "mysql_benchmark". In order not to load Tsung, we decided not to generate an rpc request per operation; thus an rpc request does 500 read or write operations depending on the test-case.

As illustrated in figure 4.2, Tsung starts a client erlang node in the local machine. This is the master node, and it is responsible for the whole process, starting from generating the rpc requests and ending with collecting all the statistics. In order to get OS related statistics, tsung's master node start a remote slave node on the server. This remote node is named os_mon (OS monitoring) and collects information regarding CPU usage, free memory, etc.

Mean session duration is the basic metric used for the evaluations. Of course, we analyze even the variance from the mean and make sure that the mean value is meaningful. As mentioned earlier, a session consists of 500 operations. The load is presented as users/second or sessions/second since a session represents one user doing 500 operations. In the results section, we also use writes/second or reads/second as a load metric. However, the number of sections(users)/second when multiplied by 500 gives the number of operations per second.

Chapter 5
5 Results and analysis

5.1 Mysql benchmark results

As explained in section 4.5 the test-cases are a combination of different OTP versions, different SMP parameters and different load. The main metric we analyze is the mean response time per user's session under different load(number of concurrent users). A simulated user during his session performs 500 operations. In other words, 500 inserts to the database in case of a write benchmark, or 500 random reads in case of a read benchmark. Since for all the tests the results for SMP with only 1 scheduler, were slightly slower compared to the ones with SMP disabled, we omitted this case from the final graphs of MySql benchmark included in this report. This behavior is very reasonable, and is due to the fact that SMP with one scheduler performs the same way as the old no SMP version of Erlang VM. Nevertheless, it introduces some additional overhead because of the scheduling and controlling algorithms of the SMP logic.

5.1.1 Mysql write-only performance

The bar chart illustrated in figure 5.1 shows the performance of Emysql driver on the Oracle Server. The horizontal axis shows the load Tsung sends to emysql driver. The load is increased gradually until the server cannot handle the requests, and the response time per second increases continuously during the whole test. The vertical axis gives the mean time of a user's session duration in seconds. For a better visibility, the results for all the write test-cases are shown altogether using clustered bars. Tests that are run under the same load belong to the same cluster. Each cluster includes results for four different Erlang VMs: R15B SMP auto, R15B smp disabled, R12B01 SMP auto, R12B01 SMP disabled. SMP auto is the default option for both the R12 and R15 versions. It means that the Erlang VM starts with SMP enabled, and the number of schedulers is equal to the number of CPU cores. In our case we run the experiments in a 16 cores machine and with SMP enabled 16 schedulers are started. As can be easily noticed, only for the first two loads (1 usr/sec and 2 usr/sec) there

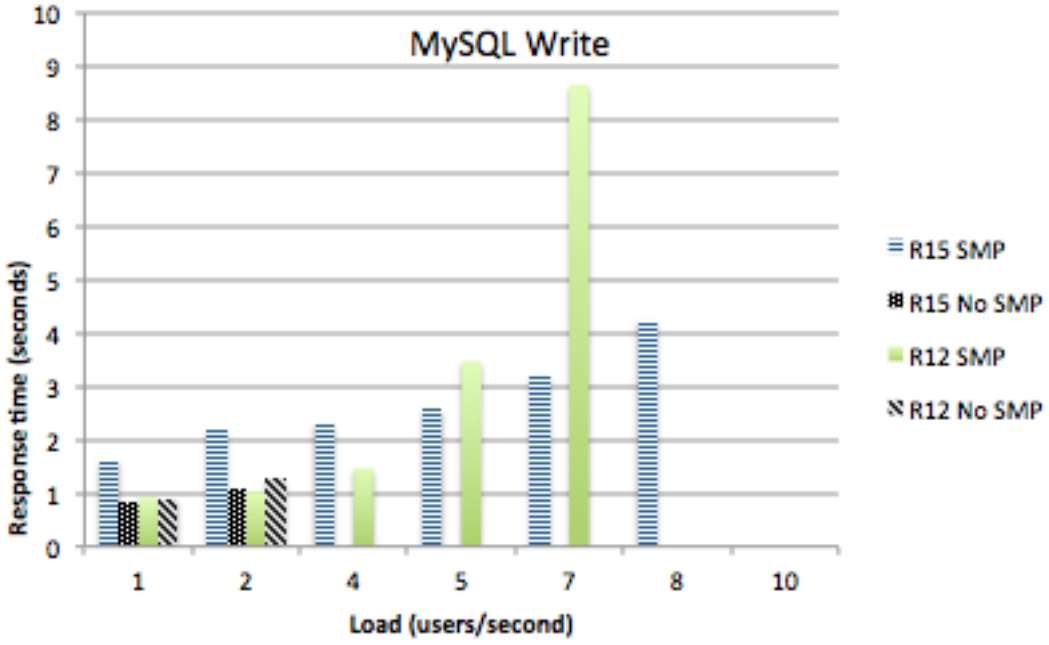

Figure 5.1: Mean response times for MySql write-benchmark (a user's session handles 500 write operations to Mysql)

are four bars. For higher loads the missing bars show that the corresponding VMs cannot handle that load at all.

As you can see from figure 5.1, SMP's performance is closely related to the load. For a low load of 1 usr/sec (which means 500 writes/sec) for both OTP versions SMP disabled gives e better performance. However, for R12 the difference between SMP enabled and disabled is not so visible. But for R15, the mean response time per user is twice longer with SMP enabled than disabled. This really bad performance of R15 SMP enabled is due to the fact that the load is very low, and the overhead of having multiple run-queues and schedulers which perform migration logic algorithms during the whole execution affects strongly the result. For little load we lose in performance by enabling SMP, especially in R15 where the scheduling algorithm is more complex and time-consuming. The situation is quite the same even when the load is increased to 2 usr/sec. R15 SMP enabled still gives the worst performance being twice slower than than the SMP disabled version. However, now there is enough load for R12 SMP enabled to perform better than the no-SMP version. As can be seen in figure 5.1, R12 with SMP enabled gives the best timing.

The situation changes in favor of SMP enabled when the load is further increased. For the third set of experiments with a load of 4 usr/sec (2000 writes/sec) there are no bars at all for R15 and R12 with SMP disabled. This means that by disabling SMP,

at 2000 writes/sec we hit a bottleneck in Emysql driver. The driver cannot handle the load, and the session duration is not stable, but keeps increasing gradually during the whole test duration. Thus, it does not make sense to calculate the mean time at all. Nevertheless, R15 SMP enabled still is slower than R12 SMP enabled. The benefit of having multiple run-queues in R15 is smaller than the overhead introduced by managing these queues.

Finally, for a load of 5 usr/sec (2500 writes/sec) and SMP enabled, R15 surpasses R12. The average response time per user in R15 is 2.57 sec., and in R12 is 3.47 sec. In the end, we managed to stress the driver enough, in order to benefit from the complex SMP algorithm of R15. The difference increases furthermore for 7 usr/sec. We can see that R12 starts performing really badly compared to R15. Moreover, the response time per user in R12 is not stable at all during the whole test duration. The variance of the mean response time per user is 8 times bigger in R12 than in R15. R12 SMP enabled gives up at the load of 8 usr/sec or 4000 writes/sec, while R15 SMP enabled still can handle the load with a mean session duration of 4.25 sec. This mean value, though high for doing 500 writes to the database, has a low variance. By increasing the load to 10 usr/sec we reach the threshold even for R15 SMP enabled.

After this analysis, we can conclude that Emysql driver can benefit from SMP in R15 by handling a load of 4000 writes/sec which is not bad compared to MySQL's Innodb performance evaluations published in Oracle's white paper [20]. However, the user's session duration of 4.25 sec for executing 500 inserts to the database is quite high. This long duration is not a driver's fault, but of how the Erlang VM handles the IO tasks. How schedulers perform IO operations in the Erlang VM will be explained by the end of this section.

Load (usr/sec) / VM versions	1	2	4	5	7	8
R15 SMP	100	100	100	100	100	100
R15 No SMP	7.98	13.82	-	-	-	-
R12 SMP	12.63	19.79	55.38	53.2	54.7	-
R12 No SMP	9.21	12.61	-	-	-	-

Table 5.1: CPU Usage for Mysql write-benchmark (%)

Another metric we checked during the evaluations was the CPU usage. Table 5.1 includes the CPU usage for all the set of experiments aforementioned. The table cells represented by a dash, show that the corresponding VM cannot handle that load. Thus, measuring CPU usage does not make sense in that case. The constant CPU usage of 100% in R15 with SMP enabled for all the loads is very noticeable. The drop of the CPU usage to around 10% in R15 with SMP disabled shows that Emysql driver is not CPU bound itself. CPU is heavily loaded only when SMP is enabled. Thus, the SMP mechanism in R15 (specifically R15B), is consuming all the

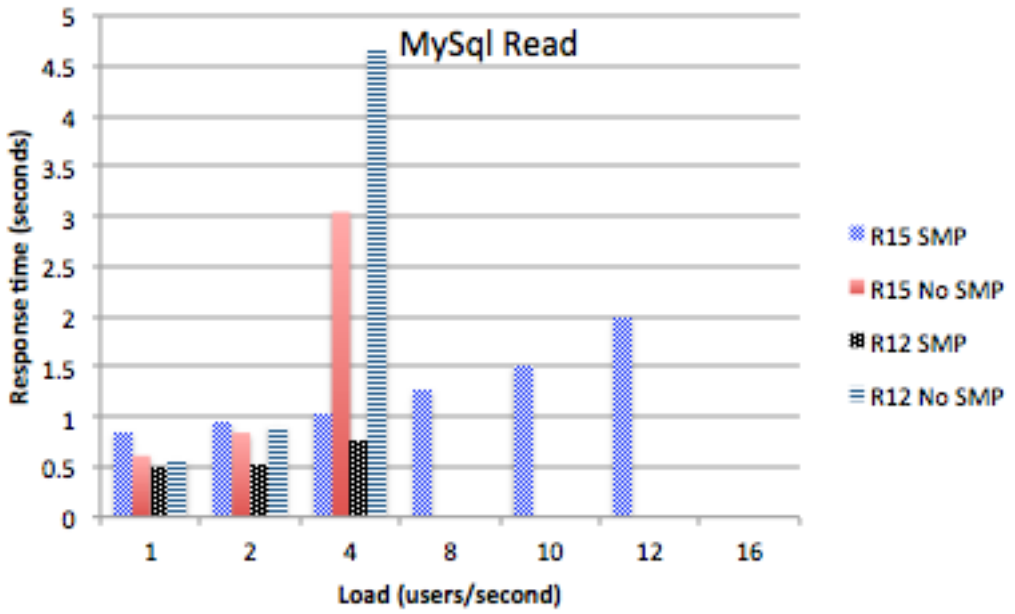

Figure 5.2: Mean response times for MySql read-benchmark (a user's session handles 500 read operations to Mysql)

CPU power. We do not see these extreme difference in R12 though. There is a slight difference in CPU usage between R12 SMP enabled and disabled with the first one being reasonably greater. For a high load such as 7 usr/sec or 3500 writes/sec with SMP enabled, CPU usage is around 55% in R12, and almost twice bigger in R15. This is another indicator of the different SMP implementation strategies included in the two Erlang OTP versions.

5.1.2 Mysql read-only performance

In the case of the read tests, a user's session consists of 500 random reads to MySql database. Figure 5.2 shows the mean session duration under different load and different Erlang VM parameters. As can be observed from the bar chart, SMP's behavior is almost the same as in the write test-cases. For low load such as 1 usr/sec or 2 usr/sec the complex SMP-logic implemented in R15 penalizes it by introducing a bigger overhead than the work done. Thus R15 with SMP enabled gives the worst performance under such load. At a load of 2000 reads/sec (4 usr/sec) there is enough load for SMP enabled VM to outperform the disabled version in both R12 and R15. Furthermore, at this load, the SMP disabled versions demonstrate a high variance of the session duration which means that they are really close to the bottleneck. Doubling the load from 4 usr/sec to 8 usr/sec shows the advantage of SMP in R15.

All the other VM versions (including SMP enabled in R12) cannot handle this load, thus they are not shown at all in the bar chart. R15 with SMP enabled continues to handle even a higher load of 10 and 12 usr/sec. Finally, at 16 usr/sec the linear increase of the session duration during all the execution shows that Emysql driver cannot handle the load.

We can come up to the same conclusion as in the write tests in regards to SMP and Emysql driver. Under a high load, SMP in R15 performs better than the other Erlang VM versions. Emysql driver in R15 with SMP enabled can handle at least a load of 12 usr/sec or 6000 reads/sec, but not more than 9000 reads/sec, which is reasonably good compared to Innodb read performance results published in the same Oracle's white paper [20].

Load (usr/sec) / VM versions	1	2	4	8	10	12
R15 SMP	60	85	96	100	100	100
R15 No SMP	3.5	6	9.5	-	-	-
R12 SMP	3.8	7.2	17.5	-	-	-
R12 No SMP	3.5	6.1	10.4	-	-	-

Table 5.2: CPU Usage for Mysql read-benchmark (%)

Table 5.2 shows the CPU usage for all the read-only performance tests shown in figure 5.2. As for the write tests, the most visible value is the very high CPU usage for R15 with SMP enabled. The drop of the CPU usage to less then 10% in R15 with SMP disabled again shows that this high CPU consumption power is caused by the SMP mechanism in R15. In R12, CPU usage is slightly greater with SMP enabled but still very low compared to R15.

5.1.3 Analysis of Emysql-driver performance

From both read and write performance tests we can conclude that SMP's behavior is closely related to the load. For low load results suggest that it is better to disable SMP for both Erlang versions (R12 and R15). Furthermore, R12 SMP enabled performs better than R15 SMP enabled for low load. However, results clearly reveal that R15 with SMP enabled can handle more load than the other versions, so it is better to be used under high load.

As for the extremely high CPU usage in R15, it is due to the aggressive scheduling mechanism implemented in the latest R14 releases and in R15. Erlang OTP team has introduced "spinning" (explained in section 3.3) when a scheduler is waiting for the next task to execute. Spinning results in a higher CPU load than before, especially when the whole Erlang node is running under low to moderate workload. However, spinning is a good finding for decreasing latency since it is more costly

to put the schedulers on sleep and wake them up after a while. If a new task appears while the spinning, it can be processed much faster than if the scheduler was sleeping. On the other hand, if nothing new appears while spinning , spinning was a waste of CPU instructions. For this reason, it is good practice not to use CPU usage as a metric while measuring the capacity or efficiency of an Erlang SMP application.

However, what we first thought was to stress Emysql driver enough for the schedulers not to spin; thus we could normalize CPU. But, we reached the performance bottleneck and schedulers still weren't loaded enough. Figure 4.1 shown in subsection 4.3.2 is an illustration of Emysql driver. There is a pool of opened connections to MySql Database. Each connection sends and receives data through socket communication. Next step was to increase the connection pool-size and check the impact on the load and SMP's behavior. We still couldn't handle more load, and nothing changed in the performance. We also checked the message queue length of Emysql's main process (Emysql connection manager) which handles all the requests for free connection to the database, and its length was reasonably low; so it could not be the bottleneck.

Finally, we analyzed how IO operations are handled by the Erlang VM, and that was the bottleneck. The problem relies on the fact that there is only one global poll-set where IO tasks are kept. Hence, only one scheduler at a time can call erts_check_io (the responsible function for performing IO tasks) to obtain pending tasks from the poll-set. So, a scheduler can finish its job, but it has to wait idly for the other schedulers to complete their IO pending tasks before it starts it own ones. In more details, for N scheduler, only one can call erts_check_io regardless the load; the other N-1 schedulers will start spinning until they gain access to erts_check_io() and finish executing their IO tasks. For a bigger number of schedulers, more schedulers will spin, and more CPU time will be waisted on spinning. This behavior was noticed even during our evaluations when running the tests in a 8 cores machine, apart from the 16 cores one. There are two conditions that determine whether a scheduler can access erts_check_io, one is for a mutex variable named "doing_sys_schedule" to be unlocked, and the other one is to make sure the variable "erts_port_task_outstanding_io_tasks" reaches a value of 0 meaning that there is no IO task to process in the whole system. If one of the conditions breaks and there is no other task to process, the scheduler starts spinning. Emysql driver generates a lot of processes to be executed by the schedulers since a new process is spawned per each single insert or read request. By using Erlang etop tool, we can check the lifetime of all the processes created in the Erlang VM. Their lifetime is extremely short, and they do nothing else (CPU-related) apart from the IO requests. The requests are serialized at this point because whenever a scheduler starts doing the system scheduling, it locks the mutex variable "doing_sys_schedule", and then calls erts_check_io() function for finding pending IO tasks. These tasks are spread to the other schedulers and are processed only when the two aforementioned conditions are fulfilled.

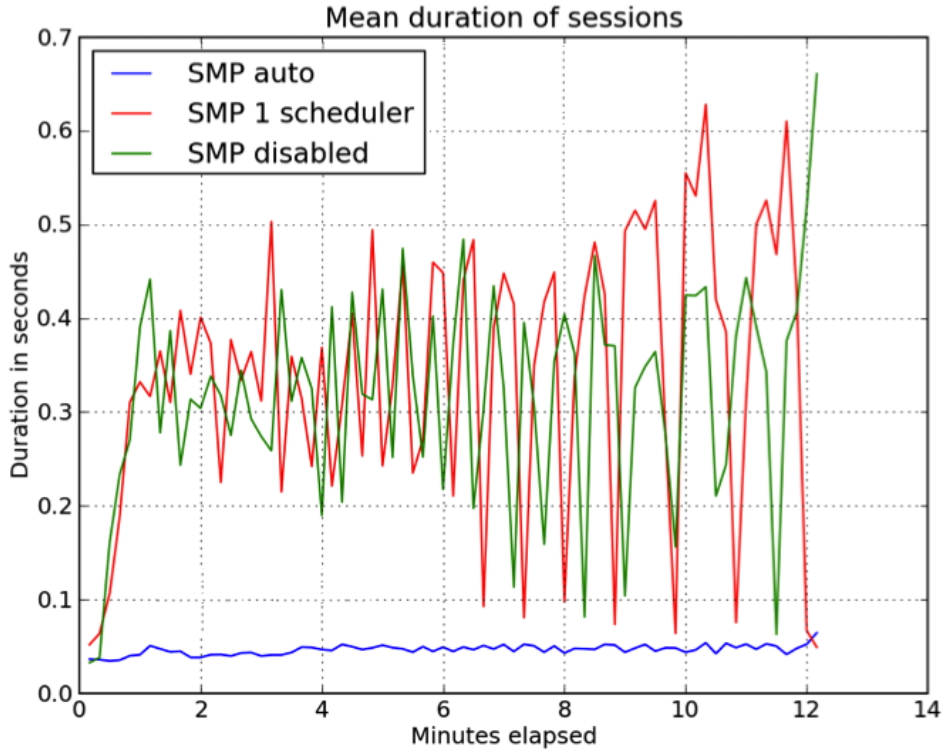

Figure 5.3: R12: Response times for Mnesia write-benchmark (a user's session handles 500 write operations)

5.2 Mnesia benchmark results

Mnesia is a distributed database that performs very fast because it is loaded in memory. However, for our experiments we do not make use of its distributed nature; we have only one Mnesia node that stores all the data. We also make sure not to load the database more that it can handle by limiting the duration of the test, the load, and the total number of inserted records. This is not fare compared to MySql database benchmark, but we wanted just to get a very general view of SMP in Mnesia. Since we store the whole table in main memory, our Mnesia benchmark is memory-bound. Hence, the results include a moderate workload. We tested SMP's behavior only for this load, because of timing limits and because of Mnesia's limited usage in real heavily loaded systems.

Figure 5.3 shows the results for a write-benchmark in R12, while figure 5.4 shows the results for the same benchmark but in R15. The load in both the cases is 20 usr/sec (10000 writes /sec). The horizontal axis represents time in minutes. As can be seen, the whole test lasts 12 minutes. The vertical one gives the mean session

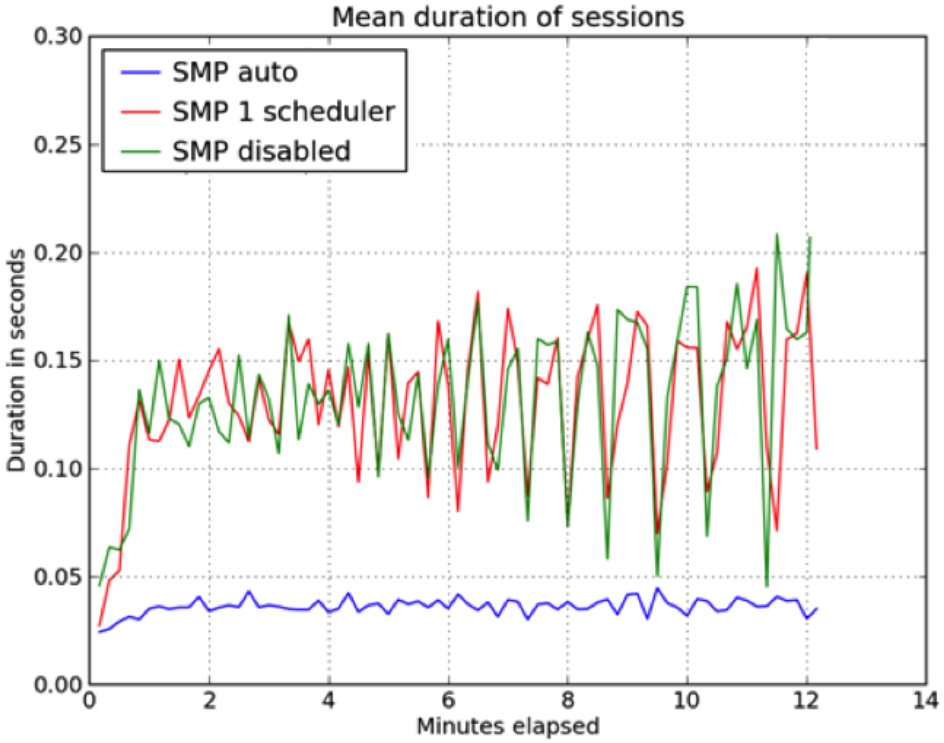

Figure 5.4: R15: Response times for Mnesia write-benchmark (a user's session handles 500 write operations)

duration in seconds for the elapsed time. In both the OTP version SMP enabled gives the best performance results. As explained earlier, SMP enabled with only one scheduler performs worse than SMP disabled in both R12 and R15. It is obvious from both the graphs that for SMP with 1 scheduler and SMP disabled the session response time is very variable and this variance increases with the passing of time. Since data are all loaded in memory, writes are very fast. Mean session duration for the whole test in R12 is 0.047 sec, while in R15 it is slightly faster, 0.036 sec. Depending on the application, it may be a good idea to use the dirty functions for certain operations. Read tests show the same behavior regarding SMP for the same load of 20 usr/sec. Since read tests are separated from the write ones, we can execute the read operations without protecting them with transactions by using the async_dirty option. We noticed that there is a considerably big difference in performance between using transactions and dirty reads. In our case it was 20 times more efficient to read records dirty than within a transaction. Hence, depending on the application, it may be a good idea to use Mnesia dirty functions for some types of operations.

Chapter 6
6 Conclusions

Tsung was a very handful tool for the purpose of our evaluations. The newly created rpc-plugin can generate load to every remote Erlang node. As for SMP's behavior for IO-bound applications, the main finding is that it is closely related to the load. If the benchmark is not doing work, what is being measured is the locking overhead from Elang VM and the kernel. Hence, for low load our MySql performance results suggest to disable SMP at all in order to achieve a better performance. However, with the increasing of the load schedulers in SMP have enough work to overpass the locking overhead and speedup performance. Our findings show that disabling SMP reduces the maximal load a system can handle.

In R15, letting the schedulers busy-poll for work can help the performance, but of course spinning raises total CPU consumption. This is the reason why it is not suggested to use CPU usage as a metric for defining the load-protecting mechanism of your system. For good scalability in SMP Erlang, it is good to always have enough runnable processes to keep all schedulers busy. We could not achieve that because the bottleneck for IO-bound applications resides in the function that handles IO tasks in the Erlang VM. This function can be accessed only by one scheduler at a time.

One of the purposes of this study was to answer the question whether it was better to switch from R12 to R15 for IO-bound applications. Eventhough the results are not always in favor of R15, especially for low load, we strongly recommend switching to R15 for good reasons. The main reason is that SMP's functionalities keep improving in every version. For example, later SMP versions offer different strategies for binding the scheduler to different CPU cores. Another improvement in the latest releases is giving the user the possibility to define how aggressive his busy-waiting strategy will be. Scheduler busy wait threshold can vary from none to very_long. Furthermore, they introduce new SMP-related debugging tools in every OTP release. In the latest one (R15B01) they have introduced erlang:statistics(scheduler_wall_time) which gives the total time, and the time each scheduler has been busy during this time.

Speedup between R12 and R15 is more visible for CPU-bound applications, since really good parallelism speedup in Erlang requires a set of active processes during the whole execution which perform balanced CPU-related work. So, there will be no need to use additional mechanisms such as migration or work-stealing during the execution time. In our case, the speedup is very dependent on the load and the function that handles the IO tasks (which is the bottleneck). In general, we would recommend doing a lot of benchmarking prior to using an IO-bound application. Though not included in this study, Erlang offers a few IO related parameters to be considered when starting the VM. For example, "+A n" (where N is an integer > 0) tells the VM to use the asynchronous I/O thread pool mechanism for local disk I/O, and "+K true" uses a different socket ready/activity. In addition, a user can bind a process to a specific core in Erlang. As mentioned earlier, first, schedulers need to be bound to the CPU cores, and then a process can be bound to a specific scheduler when spawning it. This might be useful when a lot of new processes are spawned during execution (such as in Emysql driver) and have to be spread among the schedulers.

Bibliography

[1] M. P. Jagtap, "Era of Multi-Core processors," *Power*, vol. 2, p. 2, 2009.

[2] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan. 1998.

[3] J. Armstrong, B. Dacker, S. Virding, and M. Williams, "Implementing a functional language for highly parallel real time applications," in *Software Engineering for Telecommunication Systems and Services, 1992., Eighth International Conference on*, pp. 157 –163, 1992.

[4] J. Armstrong, "The development of erlang," in *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, (New York, NY, USA), pp. 196–203, ACM, 1997.

[5] J. Armstrong, "A history of erlang," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, (New York, NY, USA), pp. 6–1–6–26, ACM, 2007.

[6] R. Virding, C. Wikström, and M. Williams, *Concurrent programming in ERLANG (2nd ed.)*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[7] S. L. Fritchie, "The evolution of erlang drivers and the erlang driver toolkit," in *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, ERLANG '02, (New York, NY, USA), pp. 34–44, ACM, 2002.

[8] H. Mattsson, H. Nilsson, and C. Wikstrom, "Mnesia - a distributed robust dbms for telecommunications applications," in *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, (London, UK, UK), pp. 152–163, Springer-Verlag, 1998.

[9] "Tsung:an open-source multi-protocol distributed load testing tool." http://tsung.erlang-projects.org/. [Accessed: 10-Jul-2012].

[10] "Synapse - the world leader in automatic device management." http://www.synap.se/. [Accessed: 14-Jun-2012].

[11] F. Cesarini and S. Thompson, *ERLANG Programming*. O'Reilly Media, Inc., 1st ed., 2009.

[12] K. Sagonas, M. Pettersson, R. Carlsson, P. Gustafsson, and T. Lindahl, "All you wanted to know about the hipe compiler: (but might have been afraid to ask)," in *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, ERLANG '03, (New York, NY, USA), pp. 36–42, ACM, 2003.

[13] C. Convey, A. Fredricks, C. Gagner, D. Maxwell, and L. Hamel, "Experience report: erlang in acoustic ray tracing," in *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, (New York, NY, USA), pp. 115–118, ACM, 2008.

[14] K. Sagonas and J. Wilhelmsson, "Efficient memory management for concurrent programs that use message passing," *Sci. Comput. Program.*, vol. 62, pp. 98–121, Oct. 2006.

[15] S. L. Fritchie, "A study of erlang ets table implementations and performance," in *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, ERLANG '03, (New York, NY, USA), pp. 43–55, ACM, 2003.

[16] H. Nilsson, C. Wikstrőm, and E. T. Ab, "Mnesia - an industrial dbms with transactions, distribution and a logical query language," in *International Symposium on Cooperative Database Systems for Advanced Applications. Kyoto Japan*, 1996.

[17] "Erlang – mnesia." http://www.erlang.org/doc/man/mnesia.html. [Accessed: 14-Jun-2012].

[18] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[19] "Innodb." http://www.innodb.com/wp/products/innodb/features/. [Accessed: 16-Jun-2012].

[20] Oracle, "Technical white paper: Mysql 5.5: Storage engine performance benchmark for myisam and innodb." http://www.oracle.com/partners/en/knowledgezone/mysql-5-5-innodb-myisam-522945.pdf, 2011.

[21] "MySQL performance blog." http://www.mysqlperformanceblog.com/.

[22] E. Hellman, "Evaluation of database management systems for erlang," in *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, (New York, NY, USA), pp. 58–67, ACM, 2006.

[23] K. Lundin, "Inside the erlang vm(with focus on smp)," tech. rep., Erlang User Conference, 2008. http://www.erlang.se/euc/08/euc_smp.pdf.

[24] J. Zhang, "Characterizing the scalability of erlang vm on many-core processors," Master's thesis, KTH Information and Communication Technology, Sweden, 2011.

[25] P. Hedqvist, "A parallel and multithreaded erlang implementation," Master's thesis, Computing Science Department, Uppsala University, Sweden, 2008.

[26] T. Nagy and A. Nagyné Víg, "Erlang testing and tools survey," in *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, (New York, NY, USA), pp. 21–28, ACM, 2008.

Declaration

I hereby certify that I have written this thesis independently and have only used the specified sources and resources indicated in the bibliography.

Stockholm, 15. July 2012

.....

Erisa Dervishi