Characterizing the Scalability of Erlang VM on Many-core Processors

JIANRONG ZHANG

KTH Information and
Communication Technology

Master of Science Thesis
Stockholm, Sweden 2011

TRITA-ICT-EX-2011:5

Characterizing the Scalability of Erlang VM on Many-core Processors

Jianrong Zhang

January 20, 2011

Abstract

As CPU chips integrate more processor cores, computer systems are evolving from multi-core to many-core. How to utilize them fully and efficiently is a great challenge. With message passing and native support of concurrent programming, Erlang is a convenient way of developing applications on these systems. The scalability of applications is dependent on the performance of the underlying Erlang runtime system or virtual machine (VM). This thesis presents a study on the scalability of the Erlang VM on a many-core processor with 64 cores, TILEPro64. The purpose is to study the implementation of parallel Erlang VM, investigate its performance, identify bottlenecks and provide optimization suggestions. To achieve this goal, the VM is tested with some benchmark programs. Then discovered problems are examined more closely with methods such as profiling and tracing. The results show that the current version of Erlang VM achieves good scalability on the processor with most benchmarks used. The maximum speedup is from about 40 to 50 on 60 cores. Synchronization overhead caused by contention is a major bottleneck of the system. The scalability can be improved by reducing lock contention. Another major problem is that the parallel version of the virtual machine using one core is much slower than the sequential version with a benchmark program containing a huge amount of message passing. Further analysis indicates that synchronization latency induced by uncontended locks is one of the main reasons. Low overhead locks, lock-free structures or algorithms are recommended for improving the performance of the Erlang VM. Our evaluation result suggests Erlang is ready to be used to develop applications on many-core systems.

Acknowledgements

I would like to thank my examiner, Professor Mats Brorsson, for his support and guidance throughout the project. I would also like to express my appreciation to Richard Green and Björn-Egil Dahlberg of the Erlang/OTP team at Ericsson, who introduced me the implementation of the Erlang runtime system and answered me many questions. Without their help, the project would take longer time to complete. Moreover, I need to thank the Erlang/OTP team for providing us benchmark programs. I have to thank researchers in Kista Multicore Center at Swedish Institute of Computer Science (SICS), e.g. Karl-Filip Faxén, Konstantin Popov, for their valuable advices.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The number of processing units integrated into a single die or package is increasing. We will see more and more general-purpose or even embedded processors with dozens, hundreds, or even thousands of cores. The many-core era is approaching. A many-core processor contains a large number of cores. Although the threshold is not definite, usually a processor with more than 30 cores can be considered as many-core. It requires more efficient techniques than traditional processors. For example, an on-chip network may be used to interconnect all cores on a chip.

1.1 Motivation and Purpose

How to fully utilize many-core systems imposes a great challenge on software developers. Programs have to be parallelized to run on different cores simultaneously. Workload should be balanced on these cores. The access of common resources has to be synchronized between different tasks, and the synchronization overhead must be as low as possible. We need good programming models, tools, or languages to make software development on many-core platforms easy and productive.

Erlang [2][3][4][5] is a language developed for programming concurrent, soft-real-time[1], 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 explicitly indicate pieces of work that can be executed simultaneously by spawning light-weight Erlang processes. The schedulers in the runtime system distribute workload carried by these processes to different cores automatically. Erlang processes are synchronized by asynchronous message passing only. When a process has finished some work, it sends messages to other processes which are waiting for it. Programmers don't have to think about locks, mutexes, semaphores, and other synchronization primitives, since there is no shared memory. All these error-prone and tedious synchronization mechanisms are hidden by the runtime system. Shared memory and related synchronization methods are only used in the

[1] For a soft-real-time system, it is tolerable if some operations miss their deadlines

Erlang VM to implement higher level features such as message passing. The scalability of Erlang applications is dependent on the performance of the VM.

The objective of this project is to study the implementation of parallel Erlang VM, evaluate its scalability[2] on a many-core platform, identify bottlenecks and provide some recommendations for improvement. The study also analyzes major parts of code in the VM that are related to the performance on many-core processors, such as synchronization primitives, memory management and scheduling algorithm. Techniques currently in use are introduced, and better techniques are investigated. The study result could give insights about the readiness of the Erlang VM to support the software development on many-core platforms.

1.2 Methodologies

A state-of-the-art processor TILEPro64[3] developed by Tilera is used in this project. TILEPro64 is a typical general-purpose many-core CPU (Central Processing Unit) with 64 cores. It integrates on-chip networks [40] which are 8x8 meshes to interconnect the cores, memory subsystem and other peripherals. The on-chip networks [12] provide more bandwidth than traditional bus or crossbar interconnection, and are more scalable when core count increases.

Some Erlang benchmark programs are utilized to evaluate the performance of the Erlang VM. Test results indicate the current version of Erlang VM achieves good scalability on the TILEPro64 processor. Some benchmarks achieve maximum speedup[4] from about 40 to 50 on 60 cores. There is also possibility for improvement by reducing lock contentions. The major problem found during benchmarking is that the parallel version of the VM using one core is much slower than the sequential version with a benchmark program. Further analysis indicates that synchronization latency induced by uncontended locks is one of the main causes. Low overhead locks, lock-free structures or algorithms are suggested to improve the performance of the Erlang VM.

1.3 Limitations

This project only investigates the scalability of the Erlang runtime system. Ideally, performance should increase linearly as the number of cores increases if an application has enough parallelism. In other words, the execution time of a program should decrease linearly as the core count increases. The metric for comparison of scalability is speedup, which indicates the ratio of improvement comparing execution time on multiple cores with that on a single core.

To evaluate the Erlang runtime system comprehensively, the performance should also be compared with other programming languages', such as C and C++. But that is not considered in this project, since the objective of this project is to investigate the new problems that are introduced on many-core systems.

[2] In this context, scalability means the ability of a system to accommodate an increasing number of cores.
[3] http://www.tilera.com/products/processors/TILEPRO64
[4] Performance gain of utilizing multiple cores

Also, only the core part of the Erlang runtime system is analyzed. Erlang comes with a set of development tools, libraries and databases, which is called OTP (Open Telecom Platform) [38]. These features are not concerned. Moreover, we focus on the execution of bytecode, and don't study the execution of natively compiled code. The networking and I/O (Input/Output) functions are not investigated too.

The benchmarks used are not real applications. They are synthetic benchmarks or micro-benchmarks. As a result, the conclusions made from this project may not reflect the actual performance of the Erlang VM very precisely. It is better benchmarked with a standard benchmark suite, which contains a diverse set of workloads. But there is no such suite for Erlang yet. Furthermore, to investigate the scalability on many-core systems, sequential benchmarks are not used since their performance cannot be improved with multiple cores. Even with parallel applications, if they don't contain enough parallelism or their performance is mainly constrained by other factors, such as network bandwidth, they are not used in this project.

Erlang/OTP is an evolving platform. The runtime system is optimized constantly by its maintainers. In this project, the release R13B04 is used, and therefore all the description and results stated hereafter are based on this version. We also focus on SMP (Symmetric MultiProcessing) VM which is the parallel version of the Erlang VM. The newer R14B released near the end of this project has similar performance on many-core processors except optimized readers-writer lock[5]. In addition, the test and analysis are based on the Linux operating systems (OS) unless otherwise specified. The SMP Linux OS used is specially built by Tilera for TILEPro64 with kernel version 2.6.26.7-MDE-2.1.0-rc.94454, and the compiler is *tile-cc* with version 2.1.0-rc.94454.

1.4 Thesis Outline

The thesis is organized as follows. In Chapter 2, background of Erlang, TILEPro64 processor and speedup calculation is described. Some related work and the contributions of this thesis are also introduced. Chapter 3 presents study result of the implementation of the Erlang VM in more details. Emphasis is given to aspects that have a great impact on many-core performance, such as message passing, synchronization primitives, memory management and scheduling. In Chapter 4, evaluation results are described and analyzed. Then some optimization suggestions are given. Chapter 5 concludes the thesis and makes recommendations for future research.

[5] A lock that can be acquired by either multiple readers or one writer

Chapter 2

Background

2.1 The Erlang System

2.1.1 Introduction

Erlang is a general-purpose, concurrent, and functional programming language developed by Engineers from Ericsson in 1980s. It was invented to provide a better way of programming telecom applications [4]. Telecom systems are highly concurrent, distributed and soft real-time systems. They are inherently concurrent. For example each telephone call is an independent transaction except interacting with other support functions such as billing occasionally, and there are a huge number of such transactions ongoing simultaneously in a system. Telecom applications are also intrinsically distributed. A phone call is served by many network elements that are physically distributed in different locations. Even in the same equipment, different phone calls may be processed by different boards. In telecom software, many operations have timing requirements. Furthermore, telecom systems have to be robust and fault-tolerant. The average downtime of a telecom system should be less than a few minutes per year.

Today, these requirements are applicable to many other applications, such as servers, financial systems and distributed databases [8]. As a result, Erlang gains more popularity in these areas. Interest in Erlang also increases for its suitability of software development on multi-core processors. With its support of light-weight concurrency, it is very convenient to develop parallel applications. Moreover, the message passing paradigm provides a higher level abstraction of synchronization mechanism than locks. As the core count increases, cache coherence will be expensive, and shared memory synchronization cost will increase dramatically due to lock contention [18]. Although lock contention can be reduced by some techniques such as data partitioning, it is not sustainable in many-core era. Regarding a many-core processor as a distributed system, in which a node consists of a core or a group of cores, and performing synchronization between nodes by message passing might be more suitable when the number of cores is very large [6]. Erlang applications can be ported to many-core systems without change if parallelism is sufficiently exposed at the beginning.

Erlang is also a high level declarative language. Declarative languages are expres-

sive. Programs developed in Erlang are usually more concise than their counterparts implemented in other traditional languages, such as C and C++, and it also takes less time to develop [32]. Shorter time to market can be achieved. In addition, the resulting code is more readable and maintainable.

While Erlang is productive, it is not a solution for all needs, and it is by no means trivial to write correct and efficient Erlang programs. It is not suitable for some application domains, such as number-crunching applications and graphics-intensive systems. Ordinary Erlang applications are compiled to bytecode and then interpreted or executed by the virtual machine which is also called emulator. Bytecode is an intermediate representation of a program. It can be considered that the source code is compiled according to an intermediate instruction set[1] that is implemented by the virtual machine and different from the one implemented by the underlying real processor. The bytecode is translated into the instructions that can be run on the real machine by the emulator. Because of this extra translation step, applications running on a VM are usually slower than their counterparts 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 [21][22][35]. But if an application is time critical and compute-intensive, and its execution time should be reduced as much as possible, such as some scientific programs, Erlang is not always a good choice [10] and a fast low-level language may be better. In one word, Erlang should be used in the right place.

2.1.2 Erlang Features

In general, Erlang has the following features[2]:

- Concurrency - A separate task, or piece of work, can be encapsulated into an Erlang process. It is fast to create, suspend or terminate an Erlang process. Erlang process is much more light-weight than OS process[3] or thread[4]. An Erlang system may have hundreds of thousands of or even millions of concurrent processes. A process' memory area can vary dynamically according to requirements. Each process has a separate memory area, and there is no shared memory. As a result, a process cannot corrupt another process' memory. Asynchronous message passing is the only way of inter-process communication provided by the language. Message sending is non-blocking. A process continues execution after sending a message. A process waiting for a message is suspended if there is no matching message in its mailbox, or message queue, and will be informed when a new message comes.

- Robustness - Erlang supports a catch/throw-style exception detection and recovery mechanism. A process can also register to receive a notification message if another process terminates even it is executing on a different machine in a

[1] The set of instructions implemented by a processor

[2] http://www.erlang.org/white_paper.html

[3] A process is an instance of a program that is being executed.

[4] A thread is a part of a process that can be executed concurrently and scheduled by operating system separately.

network. With this feature, processes can be supervised by others. If a process crashes, it can be restarted by its supervisor.

- Hot code replacement - Due to the high availability requirement of a telecom system, It cannot be halted when upgrading. Erlang provides a way of replacing running code without stopping the system. The runtime system maintains a global table containing the addresses for all the loaded modules. The addresses are updated when new modules replace old ones. Future calls invoke functions in the new modules. The old code is phased out. Two versions of a module can run simultaneously in a system.

- Distribution - Erlang applications can be executed in a distributed environment. An instance of Erlang virtual machine is called a *node*. Multiple nodes can be run on one machine or several machines which may have different hardware architectures or operating systems. Processes can be spawned to nodes on other machines, and messages can be passed between different nodes exactly as on one node.

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

- Memory management - Memory is managed by the virtual machine automatically. It is not allocated and deallocated explicitly by a programmer. Every process' memory area is garbage collected[5] 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. Also a better real-time property can be achieved. If the memory of all processes is garbage collected at the same time, without a sophisticated memory collector that can do incremental garbage collection [36] the system will be stopped for a long time.

In addition to the above main features, Erlang is a dynamically typed language. There is no need to declare variables before they are used. A variable is bound to a value when it first occurs, and the value cannot be changed later, which is called single assignment. All variables are local to the functions in which they are bound. Global variables don't exist. There is an exception that data associated with a key can be stored in the process dictionary and retrieved in the life time of that process before they are erased. It behaves like a global variable. The value associated with a key can also be changed. Using the process dictionary is not encouraged, since the resulting program is hard to debug and maintain. Erlang provides some way of sharing data, such as the ETS (Erlang Term Storage) table [15] and the Mnesia database [31].

Erlang's basic data types are *number*, *atom*, *function type*, *binary*, *reference*, *process identifier*, and *port identifier*. *Numbers* include *integers* and *floats*. An *integer* can be arbitrarily large. A large number that doesn't fit into a word is represented with arbitrary number of words, which is called *bignum*. The precision of a floating-point value is the same as that of a 64-bit double precision number defined in the IEEE 754–1985

[5]Garbage collection is to reclaim the memory occupied by data objects that are no long in use. It may compact the remaining data by moving them closer.

standard. *Atoms* are constant literals. It is like enumeration types in other languages. In the Erlang VM there is a global table storing actual values or literals of all the atoms used in the system, and atoms are indices to the table in fact. There is no separate *Boolean* type. Instead, the atoms *true* and *false* are used with Boolean operators. Since Erlang is a functional programming language, a *function* can be considered as a type of data. Functions can be passed as arguments to other functions, or can be results of other functions. They also can be stored in composite data structures such as *tuples* and *lists*, or sent in messages. A *binary* is a reference to a chunk of raw, untyped memory, or a stream of ones or zeros. It is an efficient way of storing or transferring large amounts of data. Because other data types are heavily tagged [2][33], which means in the internal representations there are extra tags indicating the types of data objects. For example, each integer has a tag. With binary, less tag overhead is introduced. A binary can be manipulated on bit level. It's a good way to implement messages or packets of communication protocols like HTTP. *References* are unique values generated on a node, and can be used to label and 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 which may be written in other programming languages, such as C and Java. An external program runs in a separate OS process, and is connected to a port via pipes[6] on Linux. In an Erlang node, a port behaves like a process. For each port, there is an Erlang process, named connected process, responsible for coordinating all the messages passing through that port.

Erlang's composite data types are *tuples*, *lists* and *records*. *Tuples* and *lists* are used to store a collection of items. Items are data values that can be of any valid Erlang types. The difference between tuples and lists is that they are processed differently. We can only extract particular elements from a tuple. But lists can be split and combined. Especially, a non-empty list can be broken into a *head*, the first element in the list, and a *tail*, a list that contains all the remaining items. *Characters* and *strings* are not formal data types in Erlang. They are represented by integers and lists of integers respectively. *Record* is similar to structure in C programming language. It is a data structure with a fixed number of *fields.* Fields have names and can be accessed by their names, while in tuples, fields (items) are accessed by positions.

Erlang programs consist of *modules*, each of which contains a number of related *functions*. Functions can be called from other modules if they are explicitly exported. A function can include several *clauses*, and a clause is chosen to execute at runtime by pattern matching according to the argument passed. Erlang doesn't provide loop constructs, so that loops are built with recursive function calls. To reduce stack consumption, tail call optimization is implemented. A new stack frame is not allocated when the last statement of a function is a call to itself.

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

[6]Pipe is used to communicate with a process by reading from or writing to associated file descriptors.

2.1.3 Erlang's Concurrency primitives

Spawn, "!" (send), and *receive* are Erlang's primitives for concurrent programming. These primitives allow a process to spawn new processes, and communicate with other processes through asynchronous message passing. When spawning a process, node name, module name, function name, and arguments to the function are passed to the built-in function *spawn()*. A process identifier is returned if the spawning is successful. Messages are sent with the *Pid* ! *Message* construct, in which *Pid* is a process identifier, and *Message* is a value of any valid Erlang data type. The *receive* statement is used to retrieve a message from a process' message queue, which has the following form:

```
receive
  Pattern1 when Guard1 -> expressions1;
  Pattern2 when Guard2 -> expressions2;
  Other                -> expressionsother
after   % optional clause
  Timeout -> expressionstimeout
end
```

In the statement, *after* clause (timeout mechanism), *other* clause and *guards* are optional. When a *receive* statement of a process is executed, the VM checks each message in the message queue of the process to see whether it is matching one of the patterns. The patterns are matched sequentially. If a pattern is matching and the corresponding *guard*, which is a test, succeeds, the expressions follow that pattern are evaluated, and the following patterns are not matched any more. When there is no message in the queue or no matching message, the process is suspended and scheduled out. A suspended process waiting for a message becomes runnable if it receives a new message, and is appended to the run queue of the scheduler which the process is associated with. Then when the process is selected to execute, the new message is matched to the patterns in the receive statement again. It is possible that the new message doesn't match any patterns, and the process is suspended once more. Sometimes, the last pattern *other* is set to match all messages, and if a message doesn't match any previous patterns, the expressions following the last pattern will be executed and the message is removed from the message queue.

When there is an *after* clause and the process is suspended waiting for a message, it will be woken up after Timeout milliseconds if it doesn't receive a matching message during that time and then the corresponding expressions are executed.

2.2 TILEPro64 Processor

Figure 2.1 is the block diagram of TILEPro64 processor. A TILEPro64 CPU integrates 64 identical processor cores or *tiles* interconnected by Tilera's iMesh on-chip networks. There are six independent networks for different purposes. It also integrates memory and some I/O controllers. Each tile is a complete processor with L1 (Level 1), L2 caches and a switch connecting the tile to the 8X8 meshes, as shown in Figure 2.2. A full operating system can run on each tile independently. In this project, we run a single

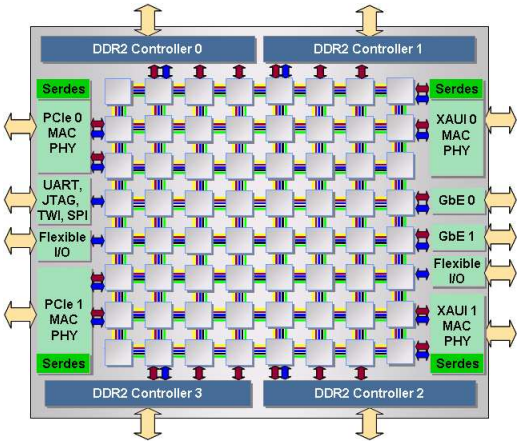

Figure 2.1: TILEPro64 Processor Block Diagram
(Downloaded from Tilera website)

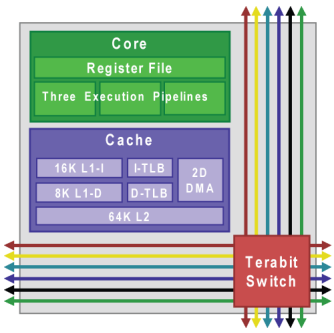

Figure 2.2: TILEPro64 Tile Block Diagram
(Downloaded from Tilera website)

SMP Linux OS on multiple tiles, and the processor used runs at 700 MHz frequency with 4 GB (GigaByte) main memory.

2.2.1 Cache Coherence

A *cache* is a memory component between processor and main memory for reducing average memory access time. Usually a processor is much faster than the main memory which is typically a DRAM (Dynamic Random Access Memory). Particularly, the interval between the moments a memory access request is issued and the requested memory can be used by a processor, i.e. memory access latency, is relatively large. The cache is faster and smaller than the main memory. It stores memory blocks recently accessed by processors. If an instruction or data requested by a processor can be found in the cache later, which is a cache hit, the access is much faster than fetching it from

the main memory every time. But if there is a cache miss the instruction or data still has to be retrieved from the main memory. Data are transferred between the cache and the main memory as blocks with a fixed size and stored in cache lines. If a part of a memory block is requested by a processor and its cache doesn't have a valid copy of that block, the whole block is copied from the main memory. Also, when a part of a memory block stored in a cache is modified and has to be written back to the main memory, the whole block is transferred.

The memory address used by an OS process is virtual address. Different processes may use the same virtual addresses, but they are mapped into different areas in the main memory except for some shared objects. In addition, memory space is divided into many equally sized blocks known as *pages*. Besides instruction and data caches, there are also caches for buffering information about the mapping between virtual addresses and physical addresses of the memory pages, which are called TLBs (Translation Lookaside Buffer).

System performance is improved with cache by exploring the principle of locality. Many programs exhibit good spatial locality and temporal locality. Spatial locality means if a memory location is accessed (referenced), it is very likely that its nearby locations will be accessed in the near future. For instance, instructions in a program are usually executed sequentially except when branch instructions are encountered. Temporal locality means if a memory location is referenced, it is very likely that this location will be referenced again in the near future. For example, instructions in a loop are executed repeatedly.

The cache subsystem is critical for providing high performance. Multiple levels of caches can be included in a computer system. In each tile of a TILEPro64 processor, an L1 instruction cache is 16 KB (KiloByte) and direct-mapped, with cache line size 64 bytes. For a direct-mapped cache, each memory block can only be cached in one cache line according to its physical address. Each L1 data cache is 8 KB and two-way associative with cache line size 16 bytes. For a two-way set associative cache, each memory block can be cached at any cache line of a set consisting of two lines. Each L2 cache is a unified cache containing data and instructions. It is 64 KB and four-way associative with cache line size 64 bytes. Each L1 instruction or data TLB has 16 entries, and is fully associative. In a fully associative cache, a memory block can be placed in any cache line.

The TILEPro64 processor provides hardware cache coherence [18] (while it could be disabled). The data stored in different caches are consistent, which means they can't contain different values for the same data. L1 cache is private to every tile, while all the L2 caches form a common and distributed L3 cache (4 Megabyte). Each cache line has a home tile. If *hash-for-home* feature is enabled, cache lines in a memory page are homed at different tiles according to a hash function, and otherwise they are homed at the same tile. By default, only stacks are not hashed-for-home. For a multithreaded program, the stack of each thread is homed at the tile where the thread is running on. When a processor core accesses a variable or a memory location, if it is not in the L1 or L2 cache (cache miss) of the same tile, it will be fetched from the L2 cache of its home tile which can be regarded as L3 cache. The L2 cache in the home tile is responsible for data consistency.

2.2.2 Processing Engine

TILEPro64 is a 32-bit VLIW (Very Long Instruction Word) processor. Two or Three instructions can be combined into a 64-bit instruction bundle which is scheduled by compiler. The processing engine in a tile has 3 pipelines, and therefore up to 3 instructions can be executed per cycle. The instruction bundles are issued to the pipelines in order. The pipelines are not stalled on load (read) or store (write) cache misses. It keeps executing subsequent instruction bundles until the data are actually required by another instruction. That means if two instructions read or write to different memory locations, they may finish execution or retire out of program order, while true memory dependencies are enforced. This achieves better performance by overlapping cache miss latency with useful work. When a cache miss happens, it will introduce high latency, since the data has to be fetched from the caches with higher levels, main memory or even hard disk, which are slower. Because the read and the write to different addresses can be retired out of order, special cares have to be taken when developing parallel programs. Memory fence instruction can be used to guarantee that all the memory operations before it are finished and visible to other tiles before the instructions that follow it are executed.

2.2.3 Memory Consistency

Memory consistency model [18] specifies the orders in which memory operations especially data writes of a processor core are observed by other cores. TILEPro64 employs a relaxed consistency model [37]. Memory store operations performed by a tile become visible simultaneously to all other tiles, but the issuing tile may see the results earlier than other tiles. Because the results can be bypassed to later instructions in the execution pipelines of the issuing tile before they are transferred to the L2 cache in its home tile. As a result, although data dependencies, such as RAW (Read After Write), WAW (Write After Write) or WAR (Write After Read) to the same memory location, are enforced on a single tile, other tiles may see them in different order. The order can be established by the memory fence instruction. Another instruction test-and-set is atomic to all tiles.

The main memory is shared by all tiles. A traditional shared memory programming model can be used to develop concurrent or parallel software applications. It also supports message passing programming paradigm. Programmers can explicitly pass short messages between tiles through one of the interconnection networks, User Dynamic Network (UDN).

2.3 Many-core Speedup

Many-core speedup is the ratio

$$Speedup = \frac{\text{Program execution time on one core}}{\text{Program execution time on multiple cores}}$$

A program's speedup can be calculated using Amdahl's Law [18]. Amdahl's Law states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used, that is

$$Speedup_{overall} = \frac{1}{(1-Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$Fraction_{enhanced}$is the fraction of code can be enhanced by using multiple cores or can be run in parallel. As a result, the overall speedup achievable is affected by the ratio of the sequential and parallel portion of a program. In this project, since we don't investigate how much programs can be parallelized with Erlang, we are mainly interested in benchmark programs with high parallelism. Benchmarks with great sequential portion complicate the problem. But pure parallel programs are rare. When measuring execution time, we try to avoid the sequential part as much as possible.

2.4 Related Work

Interest in suitability of software development with Erlang on multi-core processors is increasing. For instance, Convey et al. [10] investigate the relative merits of C++ and Erlang in the implementation of a parallel acoustic ray tracing algorithm. Marr et al.[27] analyze virtual machine support for many-core architectures including Erlang. But as far as we know, there are few literatures presenting researches on the scalability of Erlang on multi-core or many-core systems more comprehensively.

Many parts of the Erlang VM implementation are investigated in different literatures. [2] gives an overview of initial Erlang implementation. [17] documents the first attempt of building multithreaded Erlang VM, while the current implementation is not quite like that one. Erlang process' heap[7] architecture, message passing and garbage collection are introduced in [23]. Implementations of garbage collection schemes currently in use for process-local heap and binary heap are also briefly mentioned in [36].

2.5 Contributions

The major contribution of this thesis work is that we provide some insights about the feasibility and readiness of software development on many-core platforms with Erlang. We also expose the aspects of the Erlang VM that can be optimized, especially regarding to the scalability on many-core systems. In addition, we introduce many parts of the Erlang VM implementation which may hinder performance from improving on many-core systems in more details, such as synchronization, memory management, message passing and scheduling. In particular, there was no detailed description of the scheduling algorithm of the parallel Erlang VM in literatures.

[7]Heap is an area for dynamically allocated memory. It is managed by C library functions like malloc and free.

Chapter 3

Erlang Runtime System

Currently BEAM[1] is the standard virtual machine for Erlang, originating from Turbo Erlang [16]. It is an efficient register-based abstract machine[2]. The first experimental implementation of SMP (parallel) VM occurred in 1998 as a result of a master degree project [17]. From 2006, the SMP VM is included in official releases.

The SMP Erlang VM is a multithreaded program. On Linux, it utilizes POSIX thread (Pthread) libraries. Threads in an OS process share a memory space. An Erlang scheduler is a thread that schedules and executes Erlang processes and ports. Thus it is both a scheduler and a worker. Scheduling and execution of processes and ports are interleaved. There is a separate run queue for each scheduler storing the runnable processes and ports associated with it. On many-core processors, the Erlang VM is usually configured with one scheduler per core or one scheduler per hardware thread if hardware multi-threading is supported.

The Erlang runtime system provides many features often associated with operating systems, for instance, memory management, process scheduling and networking. In the remainder of this chapter, we will introduce and analyze the different parts of the current SMP VM implementation (R13B04 as mentioned before) which are relevant to the scalability on many-core processors, including process structure, message passing, scheduling, synchronization and memory management.

3.1 Erlang Process Structure

Each Erlang process includes a process control block (PCB), a stack and a private heap. A PCB is a data structure containing process management information, such as process ID (IDentifier), position of stack and heap, argument registers and program counter. Besides the heap, there might be some small heap fragments which are merged into the main heap after each memory garbage collection. The heap fragments are used when there is not enough free space in the heap and garbage collection cannot be performed to get more free memory. For instance, when a process is sending a message to another

[1] Bogdans/Björn's ERLANG Abstract Machine

[2] A model of a computer hardware or software system

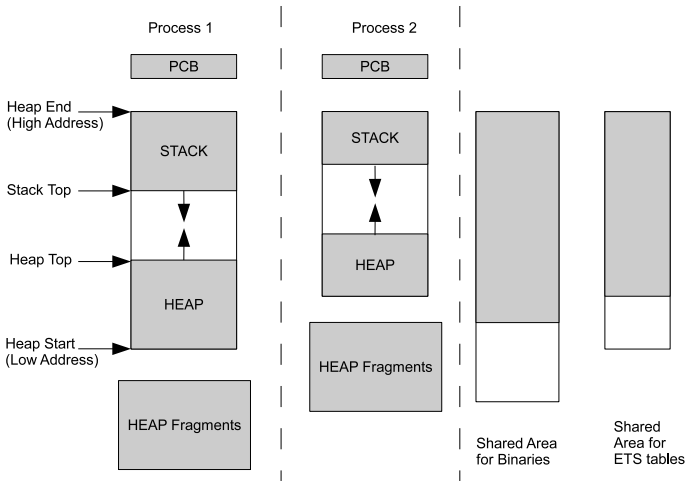

Figure 3.1: Heap Structure

process, if the receiving process doesn't have enough heap space to accommodate the incoming message, the sending process doesn't invoke a garbage collection for the receiving process in the SMP VM. In addition, binaries larger than 64 bytes are stored in a common heap shared by all processes. ETS tables are also stored in a common heap. Figure 3.1 illustrates these main memory areas (there are also other memory areas, such as for atom table).

As Figure 3.1 shows, the stack and heap of an Erlang process are located in the same continuous memory area which is allocated and managed together. From the standpoint of an OS process or thread, this area belongs to its heap, which means the stack and heap of an Erlang process actually are stored in the heap of its VM. In the area, the heap starts at the lowest address and grows upwards, while the stack starts at the highest address and grows downwards. Heap overflow can be detected by examining the heap top and the stack top.

The heap is used to store some compound data structures such as tuples, lists or big integers, while the stack is used to store simple data and references (or pointers) to compound data in the heap. There are no pointers from the heap to the stack, which eases garbage collection. Figure 3.2 shows an example of how lists and tuples are stored in the stack and heap.

Erlang is a dynamically typed language. A variable is associated with a type at runtime. Its data type cannot be determined at compile time. In the internal implementation of data, there are tags indicating the types. The two or six least significant bits of a word, which is 32 bits on a 32-bit machine or 64 bits on a 64-bit machine, are used as a tag. For a tuple, the value in the stack contains a pointer to an object in the heap. The object is stored in a consecutive memory area. It contains all the elements which can be of any valid Erlang types, even tuples or lists. It also includes a header indicating the length of the tuple. A tuple's elements can be located fast since it is an array.

On the other hand, a list is implemented as a linked list. There is no header indi-

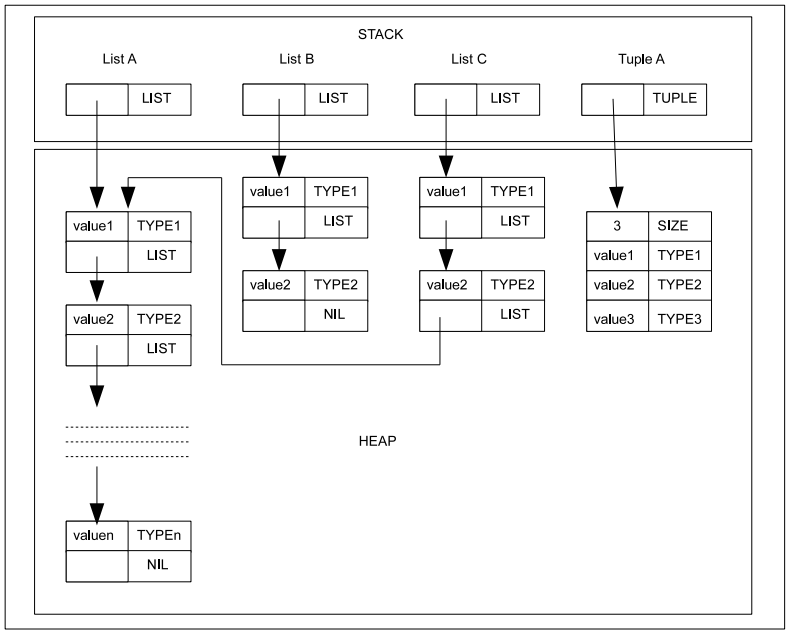

Figure 3.2: List and Tuple Layout

cating its length. Each element of a list is followed by a pointer to the next element except the last element which is followed by a null pointer NIL. Two elements may be separated by other data in the heap. Lists are used extensively in Erlang, because they can be appended, joined or split. Figure 3.2 also shows the memory layout of a list, List C, which has been constructed by appending List A to List B. First all the elements of List B were copied, and then the last pointer was modified and pointed to the first element of List A. If List B is long, the operation would take a long time to complete. Thus it is better to append a long list to a short list. Proper list manipulation is essential to write efficient Erlang applications. From the structure of a list, we also can see that to get the size of a list, all the elements have to be traversed.

The structure of List C shows that there is some memory sharing between variables in a process. But it is not between processes. If List C is sent in a message to another process, the whole list has to be copied. The message in the receiving process cannot have a pointer to list A in the sending process. In addition, if List A is sent to the same receiving process later, the content of List A will be copied again. This will result in more memory usage in the receiver than the sender.

An Erlang process starts with a small stack and heap in order to support a huge number of processes in a system. The size is configurable and the default value is 233 words. In general, Erlang processes are expected to short-lived and have small amounts of live data. When there is not enough free memory in the heap for a process, it is garbage collected, and if less memory can be freed than required it grows. Each process' heap is garbage collected independently. Thus when one scheduler is

collecting garbage for a process, other schedulers can keep executing other processes. The private heap architecture has high message passing overhead since messages are copied from the senders' heaps to receivers' heaps. However with this architecture garbage collection causes less disturbance to the system since every process is separately garbage collected, and when a process exits, its memory is simply reclaimed. Besides the default private heap architecture, the Erlang VM can also be compiled to use a hybrid architecture [23]. In hybrid mode, private data are stored in private heaps while messages are stored in a common heap for all processes. Message copying is not needed in that mode, and message passing has a constant time cost by passing pointers to messages. The problems with the hybrid architecture are: the garbage collection of the common message heap may stall all processes' execution if the garbage collector is not very sophisticated and the garbage collection time is higher since the root set contains all processes' working data. It needs an incremental garbage collection mechanism [36]. Currently the hybrid heap version of the Erlang VM is experimental and doesn't work with SMP. It also lacks compiler support. The compiler has to predict that which variables are likely to be sent as messages, and then assigns them to the common heap.

3.2 Message Passing

Message passing between two processes on the same node is implemented by copying the message residing on the heap of the sending process to the heap of the receiving process. In the SMP VM, when sending a message, if the receiving process is executing on another scheduler, its heap cannot accommodate the new message or another message is being copied to it by another process, the sending process allocates a temporary heap fragment for the receiving process to store the new message. The heap fragments of a process are merged into its private heap during garbage collection. After copying, a management data structure containing a pointer to the actual message is put at the end of the receiving process' message queue. Then the receiving process is woken up and appended to a run queue if it is suspended. In the SMP VM, the message queue of a process actually consists of two queues. Other processes send messages to the end of its external or public queue. It is protected by locks to achieve mutual exclusion (see Section 3.4). A process usually works on its private queue when retrieving messages in order to reduce the overhead of lock acquisition. But if it can't find a matching message in the private queue, the messages in the public queue are removed and appended to the private queue. After that these messages are matched. The public queue is not required in the sequential Erlang VM and there is only one queue.

If a process sends a message to itself, the message doesn't need to be copied. Only a new management data structure with a pointer to it is allocated. The management data in the public queue of the process cannot contain pointers into its heap, since data in the public queue are not in the root set of garbage collection. As a result, the management data pointing to a message in the heap is put to the private queue which is a part of the root set, and otherwise the message would be lost during garbage collection. But before the management data pointing into the heap is appended, earlier management data in the public queues have to be merged into the private queue. The order in which

the messages arrive is always maintained. Messages in the heap fragments are always reserved during garbage collection. The message queue of a process is a part of its PCB and not stored in the heap.

A process executing *receive* command checks its message queue for a message which matches one of the specified patterns. If there is a matching message, the corresponding management data are removed from the queue, and related instructions are executed. If there is no matching message, the process is suspended. When it is woken up after receiving a new message and scheduled to run, the new message is examined against the patterns. If it is not matching, the process is suspended again.

Since messages are sent by copying, Erlang messages are expected to be small. This also applies to arguments passed to newly spawned processes. The arguments cannot be placed in a memory location that is shared by different processes. They are copied every time a process is spawned.

Message passing can affect the scalability of the Erlang VM on many-core processors. First, on many-core systems access to the external message queue of a process has to be synchronized which introduces overhead. Second, the allocation and release of memory for messages and their management data also require synchronization. All the scheduler threads in a node acquire memory from a common memory space of an OS process which needs to be protected. A memory block for a message or a management data structure may be allocated from a memory pool whose memory can only be assigned by the sending scheduler. But if the message or management data structure is sent to a process on another scheduler, when the memory block is deallocated and put back to its original memory pool, synchronization is still required to prevent multiple schedulers from releasing memory blocks to the pool simultaneously. Third, if many processes can run in parallel, their messages can be sent in an order that is quite different from the order in which they are sent on the sequential Erlang VM. When messages arrive differently, the time spent on message matching can vary, which means the workload can change. As a result, the number or frequency of message passing in an Erlang application has an influence on the scalability. It is also affected by how the messages are sent and received.

3.3 Scheduling

There are four types of work that have to be scheduled, process, port, linked-in driver and system-level activity. System-level tasks include checking I/O activities such as user input on the Erlang terminal. Linked-in driver is another 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, the external program written as a linked-in driver is executed as a thread in the OS process of an Erlang node. It also relies on a port to communicate with other Erlang processes. The following description of scheduler is focused on scheduling processes.

3.3.1 Overview

Erlang schedulers are based on *reduction* counting as a method for measuring execution time. A reduction is roughly equivalent to a function call. Since each function call may take a different amount of time, the actual periods are not the same between different reductions. When a process is scheduled to run, it is assigned a number of reductions that it is allowed to execute (by default 2000 reductions in R13B04). The process can execute until it consumes all its reduction quantum or pauses to wait for a message. A process waiting for a message is rescheduled when a new message comes or a timer expires. Rescheduled or new processes are put to the end of corresponding run queues. Suspended (blocked) processes are not stored in the run queues.

There are four priorities for processes: *maximum*, *high*, *normal* and *low*. Each scheduler has one queue for the maximum priority and another queue for the high priority. Processes with the normal and low priority share the same queue. Thus in the *run queue* of a scheduler, there are three queues for processes. There is also a queue for ports. The queue for each process priority or port is called *priority queue* in the remainder of the report. In total, a scheduler's run queue consists of four priority queues storing all the processes and ports that are runnable. The number of processes and ports in all priority queues of a run queue is regarded as run queue length. Processes in the same priority queue are executed in round-robin order. Round-robin is a scheduling algorithm that assigns equal time slice (here a number of reductions) to each process in circular order, and the processes have the same priority to execute.

A scheduler chooses processes in the queue with the maximum priority to execute until it is empty. Then it does the same for the queue with the high priority. When there are no processes with the maximum or high priority, the processes with the normal priority are executed. As low priority and normal priority processes are in the same queue, the priority is realized by skipping a low priority process for a number of times before executing it.

Another important task of schedulers is balancing workload on multiple processors or cores. Both work sharing and stealing [7] approaches are employed. In general, the workload is checked and shared periodically and relatively infrequently. During a period, work stealing is employed to further balance the workload. Every period one of the schedulers will check the load condition on all schedulers (or run queues). It determines the number of active schedulers for the next period based on the load of the current period. It also computes migration limit, which is the target number of processes or ports, for each priority queue of a scheduler based upon the system load and availability of the queue. Then it establishes migration paths indicating which priority queues should push work to other queues and which priority queues should pull work from other queues.

After the process and port migration relationships are settled, priority queues with less work will pull processes or port from their counterparts during their scheduling time slots, while priority queues with more work will push tasks to other queues. Scheduling time slots are interleaved with time slots (or slices) for executing processes, ports and other tasks. When a system is under loaded and some schedulers are inactive, the work is mainly pushed by inactive schedulers. Inactive schedulers will become standby after all their work is pushed out. But when a system in full load and all avail-

able schedulers are active, the work is mainly pulled by schedulers which have less workload.

If an active scheduler has no work left and it cannot pull work from another scheduler any more, it tries to steal work from other schedulers. If the stealing is not successful and there are no system-level activities, the scheduler thread goes into waiting state. 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 a while waiting to be woken by another scheduler. If no other scheduler wakes it up, the scheduler thread is blocked on a conditional variable (see Subsection 3.4.6). When a scheduler thread is blocked, it takes longer time to wake it up. A scheduler with high workload will wake up another waiting scheduler either spinning or blocked. The flowchart in Figure 3.3 shows the major parts of the scheduling algorithm in the SMP VM. The balance checking and work stealing are introduced in more details in the remainder of this section.

3.3.2 Number of Schedulers

The load of an Erlang system (a node) is checked during a scheduling slot of an arbitrary scheduler when a counter in it reaches zero. The counter in each scheduler is decreased every time when a number of reductions are executed by processes or ports on that scheduler. The counter in the scheduler which checks balance is reset to a value (default value 2000*2000 in R13B04) after each check. As a result, the default period between two balance checks is the time spent in executing 2000*2000 reductions by the scheduler which does the balance checks. If a scheduler has executed 2000*2000 reductions and finds another scheduler is checking balance, it will skip the check, and its counter is set to the maximum value of the integer type in C. Thus in every period there is only one scheduler thread checking the load.

The number of scheduler threads can be configured when starting the Erlang VM. By default it is equal to the number of logical processors in the system. A core or hardware thread is a logical processor. There are also different options to bind these threads to the logical processors. User can also set only a part of the scheduler threads on-line or available when starting the Erlang VM, and by default all schedulers are available. The number of on-line schedulers can be changed at runtime. When running, some on-line schedulers may be put into inactive state according the workload in order to reduce power consumption. The number of active schedulers is set during balance checking. It can increase in the period between two consecutive balance checks if some inactive schedulers are woken up due to high workload. Some of the active schedulers may be out of work and in the waiting state.

As illustrated in Figure 3.4, the active run queues (or schedulers) are always the ones with the smallest indices starting from 0 (1 for schedulers), and the run queues which are not on-line have the largest indices. Off-line schedulers are suspended after initialization.

The objectives of balance check are to find out the number of active schedulers, establish process and port migration paths between different schedulers, and set the target process or port number for each priority queue. The first step of balance checking is to determine the number of active schedulers for the beginning of the next period based on the workload of the current period. Then if all the on-line schedulers should

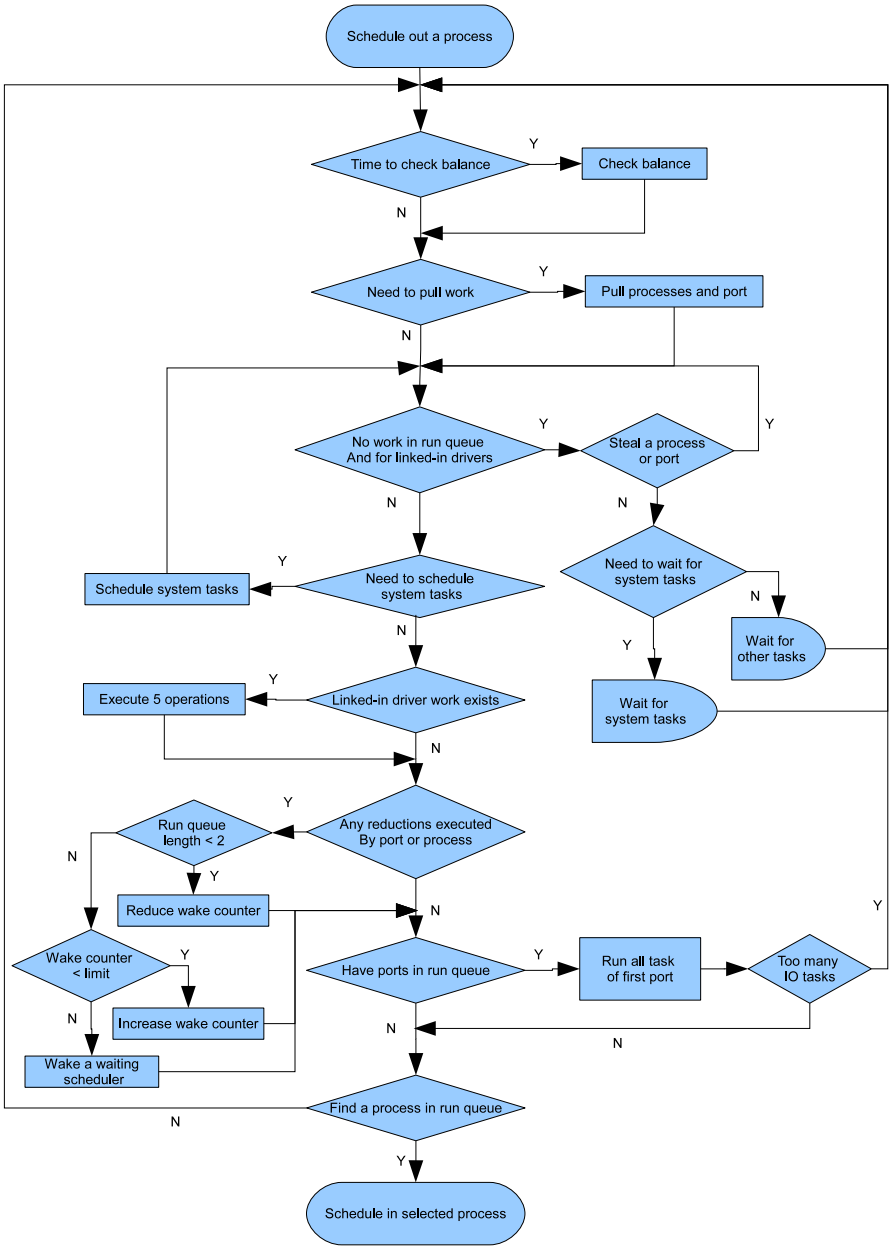

Figure 3.3: Scheduling Algorithm

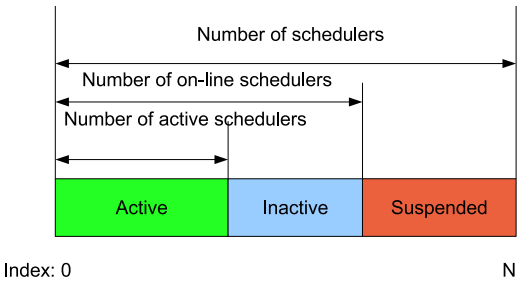

Figure 3.4: Number of schedulers

be active, migration paths and limits are determined to share workload between priority queues.

3.3.3 Number of Active Schedulers

There are two flags in the data structure for each run queue indicating whether it has been in the waiting state during a whole balance check period and the second half period (1000*2000 reductions), which are out of work flag and half time out of work flag. With these flags, the number of schedulers which are never in the waiting state for the full period, N_{full_shed}, and the number of schedulers which are never in the waiting state for the second half period, N_{half_shed}, can be counted. The number of active schedulers for the beginning of the next period, N_{active_next}, is determined with the following formula.

$$N_{active\_next} = \begin{cases} N_{online} & \text{if } N_{half\_shed}{=}N_{online} \text{or multi-scheduling is unblocked} \\ N_{act\_next2} & \text{otherwise} \end{cases}$$

N_{active_next} is set to the number of on-line schedulers N_{online}, if N_{half_shed} is equal to N_{online}. That means if all the on-line schedulers are not out of work for the whole second half period, they will be kept active in the next period. N_{active_next} is also equal to N_{online} if multi-scheduling feature is unblocked during the period. When multi-scheduling is blocked, only the first scheduler is available.

When some on-line schedulers have been in the waiting state during the second half period, and no multi-scheduling unblocking has happened in the whole period, N_{act_next2} in the previous formula is decided as follows.

$$N_{act\_next2} = \begin{cases} N_{act\_next\_min} & \text{if } N_{act\_next3} < N_{act\_next\_min} \\ N_{online} & \text{if } N_{act\_next3} > N_{online} \\ N_{act\_next3} & \text{otherwise} \end{cases}$$

N_{act_next2} cannot be larger than N_{online}. In addition, there is a minimum value for it, $N_{act\_next\_min}$. If N_{half_shed} is greater than 1, $N_{act\_next\_min}$ is equal to N_{half_shed}, otherwise it is set to 1. That means the number of active schedulers at the beginning of the next period is at least equal to the number of schedulers which keep working in the second half of the current period. N_{act_next3} is got with the following equation.

$$N_{act\_next3} = \begin{cases} N_{active\_current} & \text{if } N_{active\_pbegin} < N_{active\_current} \\ N_{prev\_rise} & \text{else if } N_{act\_next4} < N_{prev\_rise}, \text{and load decrease} < 10\% \\ N_{act\_next4} & \text{otherwise} \end{cases}$$

As mentioned before, during a period of balance check some schedulers may be out of work and in the state of waiting. They might be woken up by other schedulers with high workload later. For an active scheduler that is waiting, its state is not changed to inactive. There is another counter with each scheduler for waking up other schedulers. Every time when a scheduler has more than one process or port in its run queue, the counter will increase a number of reductions proportional to the run queue length, and otherwise decrease a number of reductions. When the counter reaches a limit, another scheduler is woken up. It tries to wake up a waiting active scheduler first, and then an inactive scheduler. If an inactive scheduler is woken up, its state is changed to active. Thus the number of active schedulers can increase in a period between two consecutive balance checks. The number of active schedulers can only decrease during balance checking.

N_{act_next3} is equal to the number of schedulers which are active currently, i.e. at the moment of the balance checking, $N_{active\_current}$, if $N_{active\_current}$ is greater than the number of active scheduler at the beginning of the period N_{active_pbegin}, which was calculated during the previous round of balance check. In other words, if the number of active schedulers has increased or some inactive schedulers have been woken up during the period, the active schedulers stay in the active state. The increase of active schedulers is also recorded for later use.

If the number of active schedulers doesn't increase in the current period, N_{act_next4} (introduced later) is compared to the number of active schedulers which was recorded at the last time when the number increased, N_{prev_rise}. If it is smaller, the maximum value of all run queues' maximum length, and the sum of reductions executed by processes and ports on all the run queues in the current period, $reds_{sheds}$, are compared with the old values which were also recorded at the last time when the number of active schedulers increased. If they are in the range of more than ninety percent, N_{act_next3} is set to N_{prev_rise}. As a result, if the number of active schedulers is increased in a period, it is not going to be decreased very easily in later periods. However, it will decrease when the maximum run queue length or total reductions of a period have fallen more than ten percent. N_{act_next4} is calculated with the following formula.

$$N_{act\_next4} = \begin{cases} \lfloor reds_{sheds} / period_{blnchk} \rfloor & \text{if some schedulers haven't waited} \\ N_{active\_pbegin} - 1 & \text{otherwise} \end{cases}$$

If some schedulers haven't been in the waiting state during the current period, N_{act_next4} is equal to the total reductions executed on all schedulers $reds_{sheds}$ divided by the balance check period (default value 2000*2000 reductions in R13B04) $period_{blnchk}$. The division result is rounded down to the nearest integer. If all the schedulers are out of work sometime in the period, N_{act_next4} is equal to the number of active schedulers at the beginning of this period minus one. As a result, if all the schedulers are waiting for work, the number of active schedulers will decrement after each balance check.

From the above description, we can see the schedulers are easier to become active than to become inactive in order to accommodate workload increase.

3.3.4 Migration Path with Under Load

For each priority queue in a scheduler, there are migration flags showing whether it should push work (*emigration flag*) or pull work (*immigration flag*). There are also fields in its data structure indicating which scheduler's priority queue with the same priority it can push work to or pull work from, and the migration limits of itself and its counterpart. The migration limits control the number of processes or ports that can be pulled or pushed, while they don't limit the work stealing. When a scheduler pulls processes or ports from another scheduler's priority queue, it should stop if either the limit of its own priority queue or the other's is reached.

If the number of active schedulers for the next period N_{active_next} is less than the number of on-line schedulers N_{online}, for the N_{active_next} active schedulers, migration flags are cleared and active flags are set. They will not push or pull work in the next period. For inactive schedulers, inactive flags are set and emigration flags are set for every priority queue. As mentioned before, the active schedulers have smaller scheduler indices than inactive schedulers. For a priority queue in an inactive scheduler with run queue index $index_{inactive}$, the queue with the same priority in an active scheduler whose run queue index equals to ($index_{inactive}$ moduloN_{active_next}) is chosen as the target for process or port emigration (push).

In this case, the system is under loaded, and not all of the on-line schedulers will be active in the beginning of the next period, while it is possible that all or some of the inactive schedulers will be woken up in that period. The active schedulers will not pull work in the next period but can steal it. An inactive scheduler can keep pushing processes or ports until there is no work, and there is no migration limit for it. A process or port is pushed when it is supposed to be added to an inactive scheduler's run queue. The push can occur when a new process or port is spawned (or created), or an old process or port has finished its time slice of execution and is being put back to the run queue.

3.3.5 Migration Limit

If N_{active_next} is the same as the number of on-line schedulers, migration limit for each priority queue of every run queue is calculated. Then migration paths are established based on the migration limits and maximum length of each priority queue. The migration limit of the *m* priority queue in a run queue with the index *n* is calculated as follows.

$migration\_limit_{m,n} = \lfloor (\sum_{n=1}^{N_{online}} maxlength_{m,n}) * (avail_{m,n} / \sum_{n=1}^{N_{online}} avail_{m,n}) \rfloor$

In the equation, *m* can be maximum, high, normal, low, or port. Although normal and low priority processes share the same queue, some of their control information, such as migration limits and migration paths, is stored separately. We can imagine a virtual low priority queue here. $maxlength_{m,n}$ is the maximum queue length of the *m* priority in the run queue with the index *n* recorded during the current period. $avail_{m,n}$ is the availability of the *m* priority in the run queue with the index *n* which will be introduced later. The first term in the right of the above equation is a sum of maximum length of all priority queues with the priority *m*, and the second term is a ratio of a priority queue's availability to the sum of availability of all the priority queues with

priority *m*. Hence migration limit is a distribution of the sum of all the maximum run queue length values according to each priority queue's availability.

$avail_{m,n}$ is calculated based on the reductions executed on the *m* priority queue in the run queue *n*, on the whole run queue *n*, and on all the run queues:

$$avail_{m,n} = \begin{cases} 1 & \text{if run queue waited} \\ availq_{m,n} * (N_{full\_shed} * fullreds_n)/fullredsall & \text{otherwise} \end{cases}$$

For run queues that have been in the waiting state in the current period, the availability of every priority queue $avail_{m,n}$ is 100%. For other run queues, availability is calculated in two steps. The first step is to calculate $availq_{m,n}$ only based on the reductions executed on a priority queue and on its run queue.

$$availq_{m,n} = \begin{cases} 0 & \text{if } redrq_n = 0 \\ 1 & \text{else if } m = max, port \\ (redp_n - red_{max,n})/redp_n & \text{else if } m = high \\ (redp_n - red_{max,n} - red_{high,n})/redp_n & \text{else if } m = normal, low \end{cases}$$

First if the sum of reductions spent on all the process priorities and port of a run queue, $redrq_n$, is zero, the $availq_{m,n}$ of each priority queue of that run queue is 0. For a scheduler whose $redrq_n$ is not zero, the availability of its maximum priority or port queue is 100%. The execution of ports is interleaved with the execution of processes, and therefore the execution of processes doesn't affect the availability of port execution. In the above formula, $redp_n$ is the total reductions spent on all process priorities of the run queue *n*, and $red_{m,n}$ is the reductions spent on processes with priority *m* of that run queue. High priority processes are always executed after maximum priority processes, and normal and low priority processes are always executed after maximum and high priority processes. Thus the calculation of $availq_{m,n}$ for a priority queue is intuitive. The normal and the low priority processes are stored in the same queue and they have the same availability.

In the second step, the $availq_{m,n}$ is adjusted according to the total reductions spent on all the run queues that are never out of work in the period to get $avail_{m,n}$. In $(N_{full\_shed} * fullreds_n)/fullredsall$, N_{full_shed} is the number of run queues (schedulers) whose out of work flags are not set during the balance check period, as mentioned before. $fullredsall$ is the sum of $fullreds_n$ of all the run queues whose out of work flags are not set. $fullreds_n$is calculated as follows:

$fullreds_n = (\sum_{i=t}^{t-7} redchange_{i,n})/8$

$redchange_{i,n}$ is a historical value of reductions spent on the run queue with the index *n*. For example $redchange_{t,n}$ is the number of reductions executed in the current period and $redchange_{t-7,n}$ is the number of reductions executed in the period that precedes the current period 7 times. If in a period a run queue is out of work, the reduction entry of that period, $redchange_{i,n}$, in its history list is set to a fixed value (2000*2000 in R13B04), otherwise it is the sum of reductions actually spent on all the processes and ports.

Figure 3.5 is a simple example of migration limit calculation. In Figure 3.5, we assume there are only processes with the normal priority which is the usual case, and each priority queue has the same availability. Then the calculation of migration limit is a simple averaging operation.

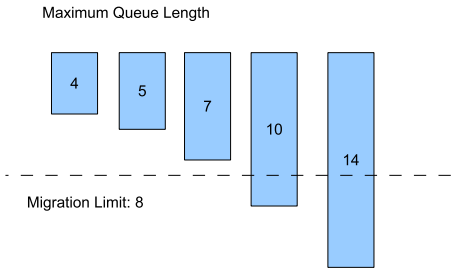

Figure 3.5: Migration Limit

3.3.6 Migration Path with Full Load

After migration limits are calculated, the next work is to establish migration paths. A migration path indicates which priority queue to transfer tasks for another priority queue having longer maximum queue length than its migration limit. For each priority queue of every run queue, its maximum queue length is subtracted by its migration limit. If the result is positive, the queue is a candidate of work emigration (push). If the result is negative, the queue has less work and is a candidate of pulling work from another queue.

For each priority, the queues are sorted according to the subtraction results. A migration path is set between the queue with the largest positive difference and the queue with the least negative difference, and then between the queue with the second largest positive difference and the queue with the second least negative difference, and so on. The emigration (push) flag is set on a queue with a positive difference, and the immigration (pull) flag is set on a queue with a negative difference. For a queue with zero availability another flag *evacuation* is set. The target for emigration (push to) or source for immigration (pull from) is also set, and there is only one target or source for each queue. A queue is either pushing or pulling, but not both.

It is possible that the number of queues with positive differences is not equal to the number of queues with negative differences. If there are more emigrating queues, the emigration flags are set on the remaining emigrating queues. For these queues, their target queues for emigration are chosen starting from the queue with the least negative difference. So there may be more than one queue pushing work to a queue. But the pulling queue only has one source for immigration. If there are more immigrating queues, the immigration flags are set on the remaining immigrating queues. The sources of immigration are chosen starting from the queue with the largest positive difference. Thus there may be more than one queue pulling work from a queue. But the corresponding pushing queue only has one target for emigration.

Figure 3.6 is an example of migration paths. There are more pulling queues in the figure. Both queues with the maximum length 7 and 4 pull work from the queue with the maximum length 14, but only the queue with the length 4 is set as the emigration target for the emigrating queue. Maximum queue length is a value recorded in a period, and it doesn't mean that the run queue has that number of processes or ports at the time of balance check.

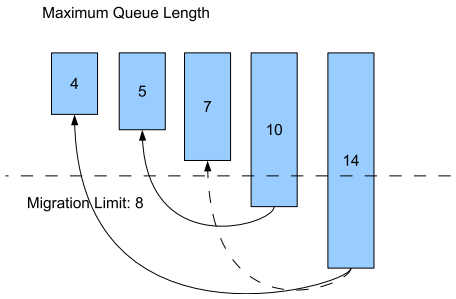

Figure 3.6: Migration Path

After the migration paths are established, in every scheduling slot of a scheduler if at least one of its priority queues has the immigration flag set, the scheduler tries to pull one process or port for each priority queue with a set immigration flag. The times of migration is limited by the migration limits set during balance checking as stated before. The processes or ports are pulled from the head of source queues.

An active scheduler with the emigration flags set doesn't push its tasks repeatedly. The emigration flag is checked when a process or port is going to be added to a priority queue. If it is set, the task is added to the end of the migration target's queue instead of the current queue. The emigration flag for that priority queue is cleared after one process or port is pushed. Thus for a priority queue of an active scheduler, it pushes work only once. Tasks are mainly pulled by priority queues which are the migration destinations.

3.3.7 Work Stealing

If an active run queue is still empty after task pulling attempt and there is no other work to do, it tries to steal a task from other schedulers. It tries to steal a task from an inactive run queue first, and then an active run queue. When stealing from inactive run queues it starts from the one with the index that is equal to:

$index_{victim\_rq} = index_{first\_inactive\_rq} + (index_{current\_rq}) mod (N_{inactive})$

$N_{inactive}$ is the total number of inactive run queues. When stealing from active run queues, it starts at the next run queue with larger index than the current run queue. Thus the stealing victims are distributed. When trying to steal from a run queue, the attempt is made from the maximum priority queue to the normal and low priority queue, and then the port queue. The stealing is successful and terminated if a process or port is stolen. The process or port is stolen from the end of a queue.

3.3.8 Scheduling and Scalability

The performance of scheduling algorithm has a great impact on scalability. Most importantly, if workload is not evenly distributed, the ideal speedup is not achievable. Another drawback induced by scheduling for many-core systems is that the cache performance may be worse. When processes are migrated from one processor core to

another, their data and code should also be transferred to the caches associated with the new core, and cache misses can occur before they are brought into the caches. But for the sequential Erlang VM running on one core, there is no process migration.

From the above description of the scheduling algorithm in the Erlang VM, we can see that it has the mechanism of distributing workload to multiple cores. It predicts the workload and the availability of each Erlang scheduler for the next period based on the data of the current period and previous periods. It puts some schedulers into the inactive state to save power consumption when a system is not fully loaded. When it is fully loaded, the scheduling algorithm tries to balance the workload periodically. It actually attempts to balance the number of processes or ports for each priority queue as the example in Figure 3.6 shows, since it is not possible to know the exact workload of each process in advance. Assuming there are only processes with the normal priority which is the usual case (excluding some Erlang system processes), the round-robin algorithm also requires each process has equal chance to execute. On a homogeneous many-core processor in which each core has the same processing power, there should be the same number of processes on every core, and otherwise the processes on cores with fewer processes will be assigned more time slices to execute than those on cores with more processes.

In reality, the numbers of processes on different cores are difficult to be kept the same with a variable workload. First, the migration limits are calculated based on the maximum length for each priority queue observed during a period. The actual number of processes at the end the period is very likely less than the sum of these maximum values. Second, processes are pushed or pulled one by one. It takes an amount of time before all the migration limits are reached. Third, the number of processes can change in a period between two balance checks because of process spawning and termination. Inside the period, processes cannot be shared except stolen. But working stealing occurs only when a scheduler is out of work. If a process on a scheduler spawns a lot of new processes, the scheduler will have much more tasks before the next balance check. Thus the properties of round-robin are not easily to be maintained on many-core systems. However on a single core, they are still kept.

This feature can affect the execution time of an individual process, although it has less effect on the total performance. For example, for an Erlang application that executes shorter than the period of balance check, it may keep every scheduler busy because of work stealing, and achieve nearly ideal speedup if all the schedulers finish their work at the same time. But a process in the application which is stolen by a scheduler occupies the whole scheduler or core, and can finish execution much earlier. Assuming there is a main process in the application and it spawns all other processes, and there are much more processes than schedulers, the processes which are not stolen are all on the same scheduler where the main process resides. They will finish execution later. In other words, different processes in an Erlang application can achieves different speedup on many-core systems. It has to be considered if the timing of an individual process is important. The speedup for different processes in an Erlang application is not guaranteed to be the same.

In addition, since schedulers in the waiting state have to be woken up by other schedulers, it also affects the speedup. The time for waking up a scheduler is dependent on the state of the scheduler and the workload of another scheduler which wakes it up.

When a scheduler is out of work and in the waiting state, it spins on a variable first. Another scheduler can wake it up quickly by changing the value of the variable. After spinning on the variable for an amount of time, if no one wakes it up the waiting scheduler thread will wait on a condition variable and be blocked. Longer time is needed to wake up a blocked scheduler thread. The counter in a scheduler for waking up other schedulers increases proportionally to the number of processes or ports on the scheduler. When there is more workload on a scheduler, it wakes up other schedulers more frequently. The time spent in waking up idle schedulers will make significant impact if the work comes in bursts.

In general, regarding the overall performance of the Erlang VM on many-core systems, a balanced workload can be expected if there are a sufficient number of processes or ports and the total execution time is long. For each individual process the speedup can vary.

3.4 Synchronization

At present, most existing commercial multi-core or many-core computer systems are with shared memory architectures. The main memory and/or a few levels of cache of a system are shared by all cores. The most efficient way of communication in these systems is through shared memory. In Erlang, although shared memory is abstracted away from Erlang application programmers, system developers still have to deal with the shared memory to build an efficient Erlang virtual machine. Many high-level features like message passing are based on shared memory. On TILEPro64, short messages can be passed between tiles directly via one of the on-chip networks. This feature may be utilized to build a different flavor of message passing or other functions. However it is not guaranteed to be faster.

Access to shared memory has to be synchronized on processors with multiple cores, otherwise programs may not behave as they are expected [18]. For example, when two threads on two different cores try to increment a variable simultaneously, if the access is not serialized they may read the same value, increment it, and write the same value back. The value ends up with being increased by only once. Synchronization is a necessity for developing shared memory multithreaded programs.

3.4.1 Overview

Synchronization introduces overhead since the progress of execution is delayed. First, a lock (introduced later in this section) has to be acquired and later released every time when a block of shared memory is accessed. This cost must be paid even when there is no contention of lock attempts by different cores. The latency introduced by acquisition and release of a lock can be different for different kinds of locks. It is affected by the speed of the memory subsystem and contention. If the memory access latency of a system is shorter, the lock overhead will also be smaller. When there is less lock contention, the overhead is likely to be smaller. In particular, when there is only one core, every attempt of lock acquisition is always successful, and the following release and new acquisition attempts will be faster if the lock variable is kept in the cache. If

there is a lot of contention, the memory copies of the lock variable in the caches of the cores which contend for it will be invalidated frequently, and cache misses will be large. This can produce an effect on scalability, since when core count increases, the contention tends to be higher.

Second, when contention occurs, threads fail to acquire a lock have to wait until it is released by its current owner. Obviously if more threads wait, more time is wasted which results in poorer speedup. The waiting time also depends on the time for releasing the lock, and the time for executing the code protected by the lock which is called a critical section. The size of a critical section or lock grain should be appropriate. With a small critical section, less shared memory is protected by a lock, and more locks may be needed. Then more lock latency is introduced. However with a large critical section, all threads which fail to acquire the lock protecting it have to wait a long time before the lock is free. Each contending thread experiences different waiting time since the access to a lock is serialized and they claim the lock one after another.

Third, for some types of locks, when contention occurs, the OS processes or threads which have failed will attempt to acquire the access again after the lock is released. Extra overhead is introduced between the time when a lock is released and the time when it is acquired again. Another issue arises when a short critical section is protected by a lock of these types. All the cores which see that the lock is free will try to lock it, although only one of them will succeed. When the critical section is short, the core having the lock will finish the execution soon and then release the lock. But before the lock release operation is performed by the memory subsystem, some lock acquisition attempts issued earlier than it may not have been executed. They are accessing the same lock variable, and usually these lock operations have to performed by the memory subsystem before the unlock operation. In that case more overhead is introduced duo to the delay of lock release operations.

In general, lock overhead can be divided into two parts. For an uncontended lock, it still introduces latency. For a contended lock, there is extra serialization cost induced by the contention. When there are more cores accessing some shared memory, the possibility that they contend with each other is greater. More contention leads to more waiting time which is wasted. As a result the total execution time is longer and the speedup is smaller. The penalty induced by contention can affect scalability greatly. In many-core systems, synchronization overhead is a potential bottleneck if the contention is high. Even with little contention, the lock latency should also be as small as possible, otherwise the parallel version of a program will waste too much time on synchronization comparing to its sequential counterpart. Basically we need low latency and low contention locks.

There are many different types of synchronization method used in the Erlang virtual machine. They can be roughly categorized into three classes: atomic primitives, locks, and condition variables. There are also different kinds of locks, spin lock, mutex, and readers-writer lock. Another synchronization method, thread gate, is built based on mutex and condition variables. Other high-level lock functions are constructed on top of these basic methods for synchronizing some specific data structures, such as run queues and process table.

Atomic primitives are used for synchronizing simple variables in order to reduce overhead, since they are lock-free and fast if they are directly built from atomic instruc-

tions implemented by the hardware. The Erlang virtual machine also tries to reduce lock contention by partitioning data structures. For instance, there is one run queue for each scheduler. Each scheduler works on its own run queue most of the time, and hence the lock contention can be reduced. To achieve a good performance, memory should be shared as less as possible.

3.4.2 Atomic Functions

User-level synchronization mechanisms usually rely on hardware primitives, i.e. assembly instructions supported by hardware, which can atomically read and modify a memory location [18]. If hardware provides these primitives, synchronizing some simple shared data, for instance an integer, with atomic operations is much faster than with lock operations, because lock overhead is eliminated. When atomic instructions updating the same memory location are issued simultaneously by several cores, they are serialized by the memory subsystem. This is like lock contention which occurs when multiple cores try to acquire a lock simultaneously. For simplicity, in this report all the contention caused by different synchronization operations is called lock contention sometimes.

The Erlang virtual machine utilizes many atomic functions, such as *atomic increment*, *decrement*, *add*, *exchange*, and *compare exchange*. There are native atomic function implementations for *x86/x86_64*, *SPARC32/SPARC64*, and *PowerPC* architectures, which means the functions are built with hardware atomic instructions provided by these architectures. For example in x86/x86_64, ordinary increment, decrement, add, exchange, and compare exchange instructions can be turned into atomic instructions by preceding them with the prefix *lock* [19].

There is also a native atomic function implementation for TILEPro64. Since there are only two instructions, *test-and-set* and *memory fence*, for building atomic operations, these atomic functions are in fact not implemented with hardware primitives but with locks. TILEPro64's test-and-set instruction loads a 32-bit word from a memory location into a register and atomically writes the value 1 into that location [37]. Since it only can write the value 1 into memory atomically, it is not possible to implement other atomic operations with it directly, for instance the atomic increment. Other atomic functions have to be built with locks.

The test-and-set instruction is suitable for building lock operations where the value 0 means that the lock is free and the value 1 indicates that the lock is not available. A processor core tries to acquire a lock, which is an integer, by writing the value 1 to the memory location assigned for the lock and examining the previous value of the lock returned in a register. If the previous value is 0, the core acquires the lock. If the value is 1, then another core has already locked it, and the lock attempt fails. The core can keep trying to acquire the lock, spinning around a loop until it succeeds which happens when the core having the lock releases it by writing the value 0 to it. This mechanism is called *spin lock*.

Storing a value in each loop introduces much unnecessary memory traffic when there are some other processor cores also trying to acquire the lock, because the value modification has to be propagated to other processor cores. The scheme can be optimized by utilizing the cache coherence feature of a processor. Instead of spinning on

read and write operations, a processor core can spin on the read operation only [34]. If the lock value is not changed by other core which owns the lock, it will keep reading the copy of the lock value stored in its cache without extra memory traffic on the networks interconnecting the processor cores and the memory subsystem. It continuously reads the lock until it is changed or freed. After the lock value is changed, the core's copy of lock is invalidated, and updated because of cache coherence when a read miss occurs. Then it races to acquire the lock again by writing the value 1 to the lock. It goes back to wait if the lock is acquired by another core.

If there are a lot of cores contending for a lock, lock contention can be reduced by introducing exponential back-off [1]. A core that failed to acquire the lock for the first time has to delay for a while before the next attempt. The delay is exponentially increased each time when it fails. The first attempt is not delayed in order to achieve low lock latency when there is no contention.

Atomic operations can be built upon spin locks if there are not appropriate hardware primitives, however they are not truly atomic, since a programmer can bypass the locks and modify the memory directly which breaks the atomicity. To obtain atomicity, memory access can be guarded by spin locks. Before modifying the memory, a lock has to be acquired avoiding multiple cores accessing it simultaneously. As the memory address space is large, there will be lots of lock contention if all the addresses are protected by a single lock. Yet it is also impractical to assign every memory address a lock. A lock table is usually used, which is the case in the implementation of atomic operations on TILEPro64 processor. On TILEPro64, the lock table resides in kernel space. Each lock or unlock operation has to cross into kernel mode[3]. It can produce substantial overhead. For a lock table, there is a trade-off with its size. Larger table introduces lower lock contention, but more memory consumption. How memory addresses are mapped into the lock elements protecting them in a lock table also affect the performance. Usually a hash function is used.

On TILEPro64, atomic functions used by the Erlang VM are implemented with C library API (Application Programming Interface) functions in *atomic.h*. These API functions are in turn built on other functions, such as atomic update and compare exchange. Atomic update and compare exchange functions invoke software interrupts, and cause corresponding system calls to be executed. These system calls implement atomic update or compare exchange based on spin locks with exponential back-off.

For other architectures without native atomic implementations, the Erlang VM implements its own atomic functions with Pthread spin lock functions. If it cannot find the Pthread spin lock implementation, Pthread mutex is used instead.

3.4.3 Spin Lock

Lock functions in Erlang VM are also built with atomic assembly instructions and Pthread routines. On Windows, Windows thread functions are used. Because this project investigates the performance of the Erlang VM on Linux, windows implementation is omitted in the following description.

[3]In kernel mode, a processor can execute any valid instructions and has unrestricted access to the hardware resources including memory. It is usually reserved for the operating system.

There are native spin lock implementations for x86/x86_64, SPARC32/SPARC64, and PowerPC architectures which are built with atomic instructions. The mechanism is similar as stated in the previous subsection, except that it doesn't employ exponential back-off. This may introduce large overhead if lock contention is high. Other methods to reduce lock contention are essential. The Erlang VM implements spin lock functions with Pthread spin lock or mutex functions for the architectures that it doesn't implement a native one. There is no native spin lock implementation for TILEPro64. It uses Pthread spin lock instead. The Pthread spin lock implementation on TILEPro64 is efficient, and implemented with test-and-set and exponential back-off.

Spin lock is efficient if the lock is held for a short period of time. Since it doesn't block the thread when lock contention occurs, the latency to lock it is low after it is released. There is re-scheduling and context switching overhead when a thread is blocked. Before a blocked thread is re-scheduled, the operating system scheduler may schedule other threads to run. Context switch consume time and have penalty of TLB invalidation. However if a lock is usually held for a long duration, spin lock is not a good choice. It wastes time that could be utilized by other threads. It increases the possibility that a thread holding the lock is preempted by the scheduler, and the threads scheduled to run later try to acquire the lock. That would waste even more time.

Simple spin locks without other technics like exponential-backoff don't scale well the number of cores is large. Each core that sees a free lock which it is waiting for will try to perform a lock operation. That will introduce a lot of contention and extra traffic on memory subsystems. With the exponential-backoff technic, there are fewer cores contending for a lock at any time, and therefore less extra memory traffic is produced. *Queuing lock* (or ticket lock) *[29]* is another technic to improve the performance of spin locks. For a queuing lock, a thread that fails to claim it will keep checking a separate variable. When its turn comes, it is informed by changing the variable that it spins on. The queuing lock can provide fairness by granting the lock according to the order when the lock requests are issued. It introduces less overhead when core count is large or contention is high, because unlike normal spin lock, when a queuing lock is released no waiting threads make acquisition attempts again, and the ownership is simply transferred to another waiting thread. Queuing locks require more memory since each core needs a separate variable.

3.4.4 Mutual Exclusive Lock

Mutual exclusive lock (mutex) is used to avoid the simultaneous use of common resource by multiple processor cores. Only one process or thread can access the memory or run code protected by a mutex lock. Mutex lock functions in the Erlang VM are implemented with Pthread mutex. Pthread mutex is not like spin lock, because it will block a thread when its lock attempt fails. To improve performance, Pthread mutex allows the thread to spin on the lock for a while in user mode[4] before calling kernel to queue up and block its execution [14][13]. Pthread mutex needs to maintain the context information for each thread. Thus it has high overhead especially when it has to go

[4]In user mode, the executing code cannot access hardware directly and run some privileged instructions. It is only allowed to reference a part of the whole memory space.

into kernel mode when contention occurs. It is relatively slow when a blocked thread is re-scheduled after the lock becomes free. It has an advantage that when a thread fails to acquire a lock it is blocked and its remaining time slice can be utilized by other threads. Since Pthread mutex locks can cooperate with the scheduler in an operating system, the thread which has failed may yield its time slice to the thread holding the lock. Pthread mutext locks are better to be used to protect critical sections that take longer time to execute than the time spent in blocking and re-scheduling a thread.

3.4.5 Readers-Writer Lock

Readers-writer locks also control access to shared resources. It allows multiple threads to read the shared memory concurrently. But if a thread needs to write the memory, it should acquire an exclusive lock. Since there might be many readers and the writer cannot grab the lock if it is acquired by one reader, it can cause write-starvation. To avoid write-starvation, writers usually have higher priority which means when a writer is waiting for the lock, a new lock request from a reader is not granted.

There are two types of readers-writer lock functions used in the Erlang virtual machine. The first one is a wrapper[5] of Pthread read-write lock implementation or constructed on top of Pthread mutex and condition variables if there is no Pthread read-write lock implementation. The second one is low-weight which is built with atomic instructions, or Pthread spin lock for some architectures that the VM doesn't implement a native one. But if Pthread spin lock functions also don't exist, the second type is implemented with the first type.

When using Pthread mutex and condition variables to build a readers-writer lock, the lock is a data structure consisting of the number of readers, waiting readers and waiting writers, condition variables for readers and for writers, and a mutex lock. A reader acquires the readers-writer lock when there are no waiting writers after it has acquired the mutex lock. The mutex lock is released after a reader has acquired the readers-writer lock, and therefore another reader can try to acquire the readers-writer lock later. If there are some writers waiting to acquire the readers-writer lock, the reader waits on the condition variable for readers, which will be broadcast by a writer unlocking the readers-writer lock when it is ready for the readers to acquire. After the condition variable is signaled, a reader continues the lock attempt. A writer acquires the readers-writer lock after it has acquired the mutex lock, and when there are no readers holding it and no earlier waiting writers. The mutex lock keeps locked if a writer is holding the readers-writer lock, in order to prevent other writers and readers from obtaining the lock. If there are some readers holding the readers-writer lock, the writer waits on the condition variable for writers, which will be signaled by the last reader unlocking the readers-writer lock.

The readers-writer lock is a data structure containing a spin lock and a counter if it is implemented with Pthread spin lock. The modification of the counter is protected by the spin lock. After acquiring the spin lock, a writer tries to acquire the readers-writer lock by setting the 31th bit, the highest significant bit of an unsigned integer, to the value 1. If all other bits are 0, which means there is no reader holding the lock, the

[5] A wrapper function is a different interface for another function. It mainly calls that function.

writer gains the lock, otherwise it retries. The first writer, which locks the spin lock and finds all the readers have released the readers-writer lock, acquires it. A reader acquires the readers-writer lock when the bit 31 is not set after it has acquired the spin lock, and then increments the counter, otherwise it waits until the bit 31 is cleared. The 31th bit of the counter is a flag indicating whether there is a waiting writer.

The method to implement readers-writer lock functions with atomic instructions is similar as when they are implemented with Pthread spin lock, except that the lock is an integer. When the lock is greater than 0, there are some readers holding the lock. If it is an extremely small negative value, it is acquired by a writer. A reader atomically increases the value of the lock by one. It acquires the lock if its old value is not negative. A writer tries to acquire the lock by atomically adding it with a small negative value. If the previous value of the lock is 0, it succeeds.

3.4.6 Condition Variables and Thread Gate

The Erlang virtual machine's condition variable functions are wrappers of Pthread condition variable functions. While other locks control access to shared data, condition variables provide a method for threads to synchronize based on the value of the data. By calling the function *pthread_cond_wait()*, a thread waits on a condition variable and is blocked until the condition variable is signaled by another thread calling function *pthread_cond_signal()* or *pthread_cond_broadcast()*. With condition variables, a thread doesn't waste time in polling the condition to check if it happens.

Thread gate controls the progress of threads waiting on a gate. The gate is a data structure consisting of a Pthread mutex lock, a condition variable, and a variable showing the number of threads allowed passing the gate. A thread waits on the gate by invoking the *pthread_cond_wait()* routine. The gate is opened by another thread calling *pthread_cond_signal()*, or *pthread_cond_broadcast()* if the number of threads allowed to pass is more than 1. After a thread passed a gate, the number of threads allowed to pass is decremented.

3.4.7 Lock Functions for Specific Data Structures

A lot of approaches are employed to reduce synchronization overhead in the Erlang VM. Many critical data structures are divided or partitioned. For example, there is one separate run queue for every scheduler, storing the processes and ports for that scheduler. In each data structure, there may be different locks to protect different fields. These methods reduce lock contention by making the locks more fine-grained. Special lock functions are built based on basic synchronization functions described in the previous subsections for some data structures to meet their special requirements. There are specific lock functions for run queue, process, port, and driver etc. Most of these functions are tailored for different data structures based on basic functions with little modification. The locks for the process data structure are a bit more complex.

3.4.8 Process Lock

The data structure for an Erlang process is protected by several locks. Accessing some fields needs to acquire only one lock, while accessing some others needs to acquire multiple locks. These locks represented by bits are combined into an integer flag. Lock order is implemented to avoid deadlock. Each lock has a sequence number equals to its position in the lock flag. A lock with a smaller sequence number is locked before a lock with a larger sequence number. Then when locks with the same sequence number on different processes are going to be locked, They are locked starting from the process with the lowest process ID to the process with the highest process ID.

At the beginning, the process lock function tries to grab all the locks needed, which are specified when the function is invoked, at once. This is implemented with *atomic compare exchange* operation, which updates a memory location with new value if the old value at the memory location equals an expected old value and returns the actual old value, otherwise the memory value is not changed and the operation fails. The function spins around a loop that atomically compares and exchanges the value of the lock flag with expected new value that it would be when the needed locks are successfully acquired, until the operation succeeds. Then by checking the old value of the lock flag, it knows whether one of the locks that it tries to acquire was locked by another thread before the atomic compare exchange operation. If there is no such lock, all the needed locks are acquired by the function.

If some required locks are already locked by other threads, then the lock function tries to lock a part of the locks that are free at the same time. Since it should enforce the lock order, the lock function finds the free lock with the highest sequence number in the required lock set, and meanwhile all other locks in the set with lower numbers are also free. After that it tries to acquire these locks simultaneously with the atomic compare exchange operation again. This procedure repeats until all the locks in the set are acquired or the times of repetition has exceeded a predefined threshold. In each iteration, the locks that it tries to lock can change, because during the period some locks which are released by other threads can be collected.

On condition that the above procedure repeats certain times and there are still locks that cannot be claimed, the function tries to acquire as many locks as possible one at a time in order. It is implemented with *atomic bitwise or* operation similar as with atomic compare exchange. It stops at the first lock that cannot be acquired. Then the lock request is put into a FIFO (First In First Out) queue associated with a process that owns the lock which it tries to acquire. The requesting scheduler thread is blocked by waiting on a thread gate.

When a scheduler thread releases a lock, it will dequeue one lock request from the head of the lock request queue related to the released lock and transfer the lock to it. The unlock function also tries to acquire all the other locks left for the dequeued lock request one by one. If not all the remaining locks can be acquired, the lock request is again put into another lock queue. However if the unlock function finds that a dequeued lock request has grabbed all the needed locks, it will open the thread gate for the thread which issued the request.

3.4.9 Synchronization and Scalability

Building synchronization functions and using them are complex and tricky. Synchronization routines are usually developed by system software developers and used by other application programmers, while in Erlang only virtual machine developers build and use them. Erlang application developers don't need to use these low-level synchronization mechanisms such atomic primitives, locks and condition variables, at all. They work with message passing as the only way of synchronization between different processes or ports.

Software development with locks is difficult. It is hard to make the programs work properly, and they are also hard to test and maintain. In particular, a lot of efforts have to be made to avoid deadlock. Although synchronizing by message passing is not guaranteed to be deadlock-free, it provides a higher level of abstraction. We can consider the synchronization mechanisms on a higher level. It is easier to think about the message interaction needed for each application and verify its correctness. Nevertheless on shared memory machines, the most efficient way of implementing all those high-level features provided by Erlang like message passing is using shared memory. Thus to a great extent, the scalability of the Erlang VM is dependent on the performance of the synchronization methods.

Synchronization functions are used everywhere in the Erlang VM whenever there are shared data and their access needs to be serialized, for example when processes are migrated between different run queues and when messages are sent between different processes. Some data structures are global that each has only one instance in a system, and they are accessed by all the schedulers. For instance, there is one global process table for an Erlang node containing the PCBs of all the processes. For global data, if they are accessed frequently the lock contention will be high, resulting in poor performance. Other data are divided and consist of several instances, such as run queues and memory allocators. Data partitioning can reduce the lock contention since most of the time each thread only accesses on a subset of the data.

There are many factors related to synchronization that can affect the scalability of the Erlang VM on many-core systems. First, the scalability is dependent on the characteristics of each Erlang application. For example, if an application spends most of its time in passing messages between processes, the locks protecting these messages will introduce big overhead to the total execution time. It will not scale well if the contention is large. Furthermore, the performance of different applications can be limited by different locks. For instance, for an application in which many processes generate a lot of other processes the lock guarding the process table may become a major bottleneck, while for an application with a large number of messages, its performance may limited by the locks protecting the passing of messages.

Second, as mentioned in the above subsections, the scalability is also dependent on the types of locks used and where they are used. Every type of locks has its special properties. For each critical section or a block of shared data, a suitable lock (or atomic primitive) has to be chosen according to its execution time, the contention rate, and the number of cores.

Many different types of locks are utilized in the Erlang VM to fit different data. New technics are continuously employed to make it more scalable. Due to the complex

Allocator type	Description
temp_alloc	Allocator for temporary allocations
eheap_alloc	Allocator for Erlang heap data
binary_alloc	Allocator for Erlang binary data
ets_alloc	Allocator for ETS data
driver_alloc	Allocator for driver data
sl_alloc	Allocator for memory blocks that are expected to be short-lived
ll_alloc	Allocator for memory blocks that are expected to be long-lived
fix_alloc	Fast allocator for some data with fixed size based on ll_alloc
std_alloc	Allocator used for most memory blocks not allocated by above allocators
sys_alloc	Usually default malloc implementation of the OS
mseg_alloc	Memory segment allocator that caches deallocated segments

Table 3.1: Allocators

nature of the locks and their pervasive use in the VM, it is quite likely that there are many things that still need to be improved. Synchronization overhead is a potential bottleneck, especially for applications that can make many schedulers in the VM access some shared data frequently and simultaneously. Whenever there is significant lock contention, the ideal speedup is hard to achieve.

3.5 Memory Management

The amount of memory that is used by the Erlang virtual machine changes dynamically. When a process is spawned, new memory has to be allocated for its stack and heap, and a slot in the preallocated process table is assigned for its process control block. A process' heap can also grow and shrink according to the memory demand. When a message is passed, memory is allocated for it and for its management data structure. If a process exits, its memory is reclaimed by the VM.

Since memory is not deallocated by programmers explicitly, the VM is responsible for collecting memory that is not used anymore by a process. When a process' heap doesn't have enough space to accommodate new data, it is garbage collected. It expands if the garbage collection couldn't free enough free memory.

3.5.1 Overview

The Erlang VM contains an internal memory allocator library, *erts_alloc*[6], for allocating memory blocks dynamically at runtime. Currently there are 11 types of allocators as shown in Table 3.1. Eight of them, excluding *fix_alloc*, *sys_alloc* and *mseg_alloc*, belong to an internal framework called *alloc_util*. The purpose of having multiple types of allocators is to reduce memory fragmentation by separating different kinds of memory blocks, and reduce time spent in finding suitable memory blocks that are frequently allocated.

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

sys_alloc and *mseg_alloc* allocators acquire memory from the OS by calling *sbrk()* or *mmap()* functions. They are the foundations of other allocators. Other allocators manage their own memory pools allocated by the *sys_alloc* and *mseg_alloc* allocators. Different types of allocators use different structures to organize their free memory blocks, such as binary search tree and linked list.

The *sys_alloc* and *fix_alloc* allocators are always enabled. The *mseg_alloc* allocator is enabled if the system has the *mmap()* implementation and other allocators using it are enabled. Other *alloc_util* allocators can be enabled or disable by users. *sys_alloc* is the default replacement if a type of allocator is disabled. The number of allocator instances for each *alloc_util* type can also be set according to the number of scheduler threads. There can be one instance per scheduler thread for some alloc_util types. One allocator instance per scheduler thread reduces lock contention, but also introduces more memory consumption. It should be configured based on the characteristics of a specific application. If the application uses a lot of memory or there is a lot of message passing, one allocator instance per scheduler may benefit the performance. For a compute-intensive application, fewer allocator instances may result in less memory footprint[7] and better performance. Figure 3.7 shows the relationship between different types of allocators. For simplicity, only one *alloc_util* allocator instance is illustrated in the graph. The remainder of this section will introduce these allocators in more details.

In the Erlang VM, different garbage collection mechanisms [24] are applied on different heap areas. A copying generational garbage collector is used for process heaps. The common binary heap is garbage collected with reference counting. Each binary data contains a counter indicating the number of processes with references (pointers) pointing to it. It is reclaimed when the counter reaches zero. The common heap for ETS tables is not recycled automatically. Instead, programmers need to delete the tables manually. However, a table is linked to the process that created it, and when the process exits the table is deleted by the VM. The table for storing atom values is also not garbage collected and it cannot be deleted. It keeps growing when new atom values are used.

3.5.2 sys_alloc

On Linux, this type of allocator is a wrapper of *malloc* [25] implementation of GNU C library by default. The Erlang VM also implements its own *malloc* functions for some operating systems if their native implementations don't performance well. It is an *address order best fit* allocator based on Red-Black binary search tree [11]. *Best fit* means the allocator tries to find a memory block that is equal to the size required, and if there is no such free block, a block with larger and the closest size is selected. For *address order best fit*, when there are multiple free blocks with the required size, the block with the lowest address is chosen.

Memory blocks are acquired from the OS via the system call *sbrk()* when there are no suitable free blocks. A process' virtual memory space is divided into different seg-

[7]The amount of main memory used by a program while it is running

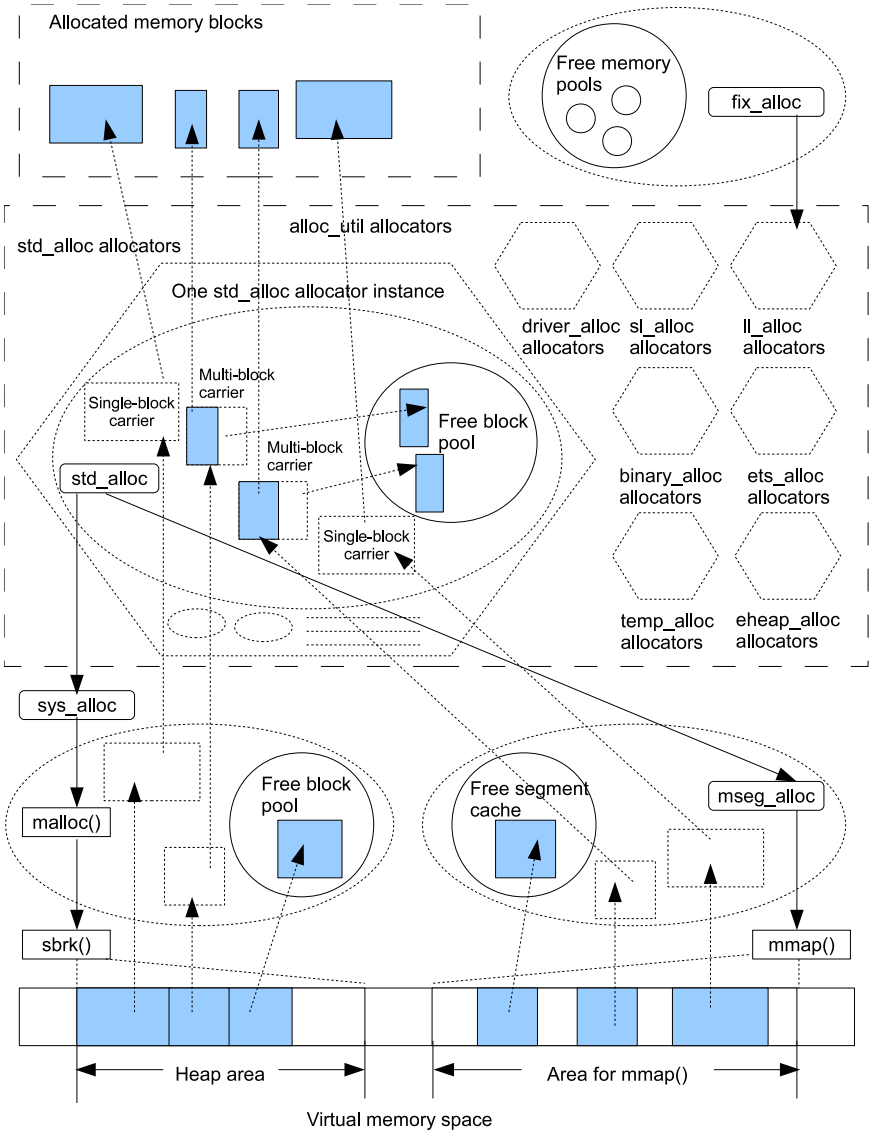

Figure 3.7: Relationship of allocators

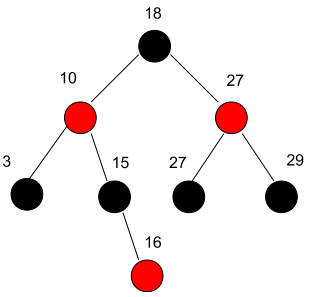

Figure 3.8: A Red-Black tree

ments, such as text[8], stack, data[9], BSS[10] and heap. *sbrk()* increments the heap segment by a user specified size. As the heap segment incremented by *sbrk()* is consecutive, a memory block cannot be freed to the OS before other blocks that were allocated later than it are freed. Memory blocks acquired by the system call *mmap()* don't have this limitation, and therefore the Erlang VM also provides the *mseg_alloc* allocator based on *mmap()*. The Erlang VM's implementation of *malloc* doesn't call sbrk() to shrink the segment size, and hence the allocator doesn't return memory to the OS.

The performance of an allocator is affected by how the free memory blocks are organized. The *malloc* implementation of the Erlang virtual machine links the free memory blocks as a balanced binary search tree, Red-Black tree. This kind of trees guarantee that search, insert, and delete operations take O($log_2 n$) time in the worst case, where n is the number of nodes in the tree.

A Red-Black tree is a binary search tree with one extra bit per node indicating its color which is either *red* or *black*. If a child or the parent of a node doesn't exist, it is regarded pointing to a dummy external node. It satisfies the following properties: every node is either red or black; the root is black; the leaves (external nodes) are black; if a node is red, both its children are black; all paths from a node to its descent leaves contain the same number of black nodes. Figure 3.8 is an example of Red-Black tree, in which external nodes are omitted.

Each node contains pointers to its left and right children, and its parent. It also contains a key field. In the free memory pool of an allocator, a memory block corresponds to a node in the tree with its memory size and/or address as its key. All the keys in the left subtree of a node are not greater than the key of that node, and all the keys in the right subtree of a node are not less than the key of that node. If there are two free memory blocks with the same size, the Erlang VM's *malloc* implementation puts the one with lower memory address to the left subtree of the other node.

Searching for a block for memory allocation from a free block tree is simple. Starting from the root node, if the size of the current node, i.e. a memory block, is less than the required, its right subtree is checked afterwards, otherwise this node is marked as

[8] A segment contains program code.

[9] A data segment contains initialized global and static variables.

[10] A data segment contains uninitialized global variables and static variables that are initialized to zero by default.

a candidate and its left subtree is checked for a more suitable one. This procedure will continue until a leaf node is reached. The last marked candidate node is the *address order best fit* node.

How the selected node is removed from the free memory block tree is dependent on the number of its child nodes. If it has no child (excluding any dummy external nodes), it is removed from the tree directly. If it has only one child node, it is removed, and its parent node is connected with the child node. When the selected node has two children, it is replaced by its successor which doesn't have a left child, and the successor's right child is connected to the successor's parent node. A node's successor has the smallest key that is greater than the node's if all the keys are distinct. If a node has a right child, its successor is the leftmost node of its right subtree. Thus in the case of node deletion, the successor doesn't have a left child because it is the leftmost.

Then if the removed node or its successor replacing it is black, the properties of Red-Black tree may be violated, and as a result a procedure to restore the properties is needed which can be found in [11].

When a memory block is deallocated, it is inserted into the free block tree. Before the insertion, if the block with the memory address immediately before or after the deallocated block is also free, they are merged after the preceding or tailing block is unlinked from the free tree to reduce memory fragmentation. Then the merged block is inserted back to the tree. There is a flag in each block header indicating whether the preceding block is free.

The process of looking for the position to insert is also simple. Staring from the root node, if the size of the merged block or deallocated block without merging is less than the size of the current node or equal to the size of the current node but with lower address, it is inserted as the left child of the current node if the current node doesn't have a left child (excluding the external node), and the current node's left child is checked if there is one already. If the size is greater than the size of the current node or equal to the size of the current node but with higher address, it is inserted as the right child of the current node if the current node doesn't have one, and otherwise the current node's right node is checked. The blocks are inserted as leaves (excluding the dummy external nodes).

A newly inserted block is set as red if it is not the root node. If its parent is a red node, the properties of Red-Black tree are violated. The procedure to restore the properties can be found in [11].

3.5.3 mseg_alloc

An *mseg_alloc* allocator acquires memory blocks or segments from the OS via the system call *mmap()*. It also releases memory to the OS using *munmap()*. Before releasing deallocated memory segments, they are cached for a while to save time for later allocation. The cache is a linked list with fixed size. A segment is removed from the end of the cache periodically. When allocating a memory segment, the cache is checked for a best fit one before acquiring a new one from the OS, which reduces the number of system calls.

mmap() is a POSIX system call that maps files or devices to memory. The Erlang VM utilizes anonymous mapping that maps a certain area of virtual memory backed

by swap space[11] instead of a file if it is supported by the OS. If it is not, the virtual memory is mapped to a special file */dev/zero*.

3.5.4 alloc_util allocators

An *alloc_util* allocator manages multiple memory carriers as a pool for allocation. A carrier is a segment of memory either allocated by *mseg_alloc* or *sys_alloc*. A *single-block carrier* stores one block, and a *multi-block carrier* contains several blocks. When allocating, if the required block size is larger than a threshold parameter, a single-block carrier is assigned, otherwise it is placed in a multi-block carrier. Usually there is a main multi-block carrier that is never deallocated for each allocator. Thus if there is one instance per scheduler for some *alloc_util* allocator types, the allocator instances acquire memory from the OS even when they are not used.

There is a total limit for the number of carriers that are allocated by the *mseg_alloc* allocator. When the limit is reached, new carriers will be allocated by the *sys_alloc* allocator. For each type of the *alloc_util* allocators, there are also limits for single-block carriers and multi-block carriers that the *mesg_alloc* allocator can allocate. For every allocator if these limits are not reached, new carriers are acquired from the *mseg_alloc* allocator.

If a memory block being allocated should be placed in a single-block carrier, the *mseg_alloc* or *sys_alloc* allocator is called to allocate a memory segment for the carrier. When a block in a single-block carrier is deallocated, the carrier is freed by the *mseg_alloc* allocator if it was allocated by it, and otherwise it is freed by the *sys_alloc* allocator. There is no free block list for single-block carrier.

When allocating a block that should be placed in a multi-block carrier, the free blocks in currently allocated multi-block carriers are searched before allocating a new carrier. If a free block found in a multi-block carrier is larger than the required size and has extra capacity to make a new free block, it is split. When a block is deallocated, it is coalesced with the preceding and/or following block to reduce memory fragmentation if both or one of them is also free. A multi-block carrier is released by the *mseg_alloc* or *sys_alloc* allocator when the whole carrier is free. The free blocks in multi-block carriers are managed according to the allocation strategy that an *alloc_util* allocator uses. There are four allocation strategies: *best fit*, *address order best fit*, *good fit*, and *A fit*. *Address order best fit* is similar as the one used in the Erlang VM's own implementation of *malloc*.

The *best fit* strategy is also implemented with a balanced binary search tree, but it is slightly different with the one used in the *address order best fit*. All the nodes in a tree have distinct keys, i.e. memory sizes. If some memory blocks have the same size as that of a node in the tree, they are linked as a list, and the node contains an extra pointer pointing to the list.

A deallocated block is inserted at the head of the list if there is a node in the tree with the same size, while allocation starts from the tail of the list. This can reduce allocation time when lists are not long, because if a tree node is removed it may take extra time to

[11] An area on hard disk holds some data temporarily for main memory when it doesn't have enough space.

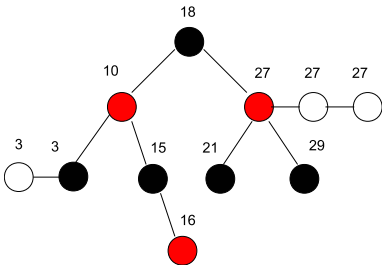

Figure 3.9: A Red-Black tree with lists

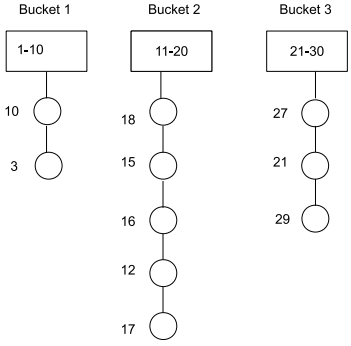

Figure 3.10: Buckets

find its successor and restore the Red-Black properties. Figure 3.9 is a simply example of the structure described above (the key values are not realistic memory block sizes).

Good fit is a special algorithm implemented in the Erlang VM. The free blocks are organized as segregated free lists or buckets. Each free list stores blocks with sizes in a specific range, as illustrated in Figure 3.10 (block sizes are not realistic).

When a block of a multi-block carrier is deallocated, it is linked as the head of a list according to its size. When allocating a block, the bucket that covers the required size is searched first if it is not empty. If the search fails and there is a non-empty bucket covering larger size, that bucket will be searched. The second search always succeeds, because all the blocks in the bucket are larger than required. The search in each bucket is limited by the maximum search depth which is small, by default 3. The algorithm tries to find a best fit from the limited number of blocks starting from the list head. All the insert, delete and search operations take O(1) time which means it is not dependent on the number of nodes and sizes of lists.

An *A fit* allocator manages only one free memory block list. A free block is inserted at the head of the list if it is larger than the old head block, otherwise after it. Thus the block at the head is always the largest. When allocating, only the first block in the free list is checked. If the first block is smaller than required, a new carrier is created. The time for block insert, delete and search operations is constant. This allocation strategy is fast, but doesn't use memory efficiently. It is only used by temporary allocator.

A *util_alloc* allocator has data structures to store its configurations, for instance the memory carriers it manages, parameters controlling the sizes of carriers, and allocation strategy it uses. These parameters are chosen to meet specific requirements of different applications, and can be customized by users.

3.5.5 fix_alloc

A *fix_alloc* allocator manages memory pools allocated by *ll_alloc* that are never deallocated. There are different pools for some different data structures with fixed size that are allocated frequently, such as processor structure, atom structure, and module structure. Every time a *fix_alloc* allocator acquires memory that can serve a certain number of requests for a specific data structure if the free memory has run out. The free blocks for a data structure are linked as a list. When there is a memory allocation request, the memory block at the head of the related list is handed out. It is a very fast allocator.

3.5.6 Process Heap Garbage Collection

In the Erlang VM copying generational garbage collector is used for process heaps. A generational garbage collector classifies memory objects into different generations. In the Erlang VM there are two generations, young and old. The collector is based on the observation that the most recently created objects are also the most likely to become unused quickly. By avoiding processing objects with longer life repeatedly the garbage collection time can be reduced. The young generation is garbage collected more frequently than the old generation. In the Erlang VM, for each process garbage collection is performed when there is not enough free space in the heap to store new data. When a process terminates, its memory is simply reclaimed without garbage collection since the heap is private for every process. The garbage collector is a Cheney-type [9] stop-and-copy collector with two types of collection, minor collection and major collection. When collecting garbage, the process is stopped. During minor collection only the young generation is garbage collected, while during major collection, all the generations are collected. Major collection is performed after a number of minor collections, or after that a minor collection couldn't free up enough space as required. The garbage collection algorithm is described as follows.

Besides the ordinary heap, a process may also have a heap storing the data of the old generation. The ordinary heap contains the data of the young generation. Data objects that have survived two or three minor collections are promoted to the old generation. That is controlled by a mark, high water mark, in the young heap. The data objects with lower addresses than the high water mark are older young generation, while the data objects above the mark are younger young generation. The data objects below the mark have survived at least one minor collection or major collection.

During a minor collection a new heap is allocated to store the data of the younger young generation that can survive the collection. Its size is determined by comparing the size of the young heap and heap fragments associated with the process being garbage collected to a table. Entry values in the table grow in a Fibonacci sequence starting from 34, and when the values are greater than 1.3 million they grow proportion-

ally. So the size of the new heap may grow in order to reduce garbage collection times. Data in the heap fragments are copied to the new heap without garbage collection.

Root set, from which what data are working can be derived, includes the stack, process dictionary, argument registers, messages that are not attached, and some other elements in the process structure (PCB) which are not immediate values. Working data referenced by the root set in the young heap are copied to the old heap if their addresses are under the high water mark (older young generation), otherwise they are copied to the new heap, and the references in the root set are updated pointing to old or new heap. After that some references in the new or old heap may still point to data in the young heap. Thus memory objects in the young heap referenced by data in the new heap or older young data in the old heap are copied to the new or old heap according to their original positions.

The young heap is freed after all the working data are moved, and the new heap becomes new young heap, or new ordinary heap for the process. During a minor collection, the old generation data in the old heap which were stored during previous collections are not touched. This reduces garbage collection time. After a minor collection, the high water mark is set to the start of the new young heap if there were older young generation data during the collection, and otherwise it is set to the new young heap's top. The working data copied to the new heap is compacted which means they are stored in a consecutive memory area starting from the heap start to the heap top. Figure 3.11 is a simple example of the memory movement in a minor collection (Heap fragments and stack are omitted).

During a major collection data both in the current heap (young generation) and the old heap referenced by the root set are copied to the new heap. Then the data in the new heap are checked to get the remaining working data, which are indirectly referenced by the root set, from the current and the old heaps. After that the current heap and the old heap are freed, and the new heap becomes new current heap of the process. Figure 3.12 is a simple example of the memory movement in a major collection (Heap fragments are omitted and stack).

3.5.7 Memory and Scalability

Modern processors are usually much faster than the main memory. To address the problem, a hierarchy of memory is introduced in computer systems. A memory subsystem can include registers, multiple levels of caches, main memory, and a swap space on a hard disk. From the registers to the swap space, their speedups decrease, while their sizes increase. A slow memory subsystem can be a bottleneck for the whole system.

For a multithreaded program like the Erlang virtual machine, all the threads share the same virtual address space. When allocating memory for different threads simultaneously, synchronization mechanisms are needed. The contention and latency induced by synchronization may reduce the scalability of the program significantly if it contains a lot of memory allocation. Thus on the Erlang virtual machine, scalability is dependent on the characteristics of each application. If an application requires frequent memory allocation, it may scale poorly.

The number of instances for a type of allocator can also affect the scalability, since when there are more instances it is less likely that contention occurs. If there is one

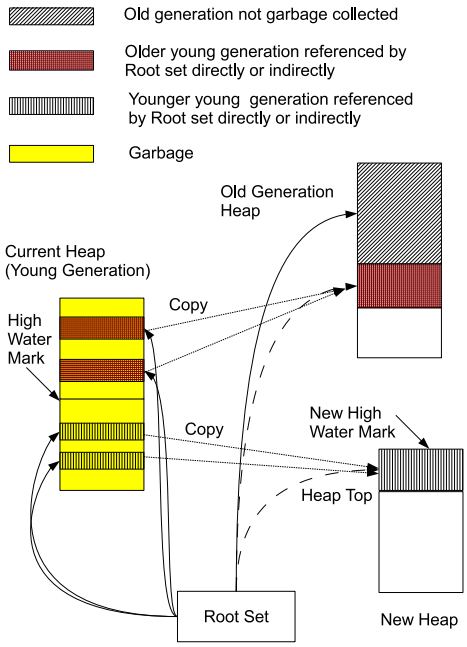

Figure 3.11: Memory movement in minor collection

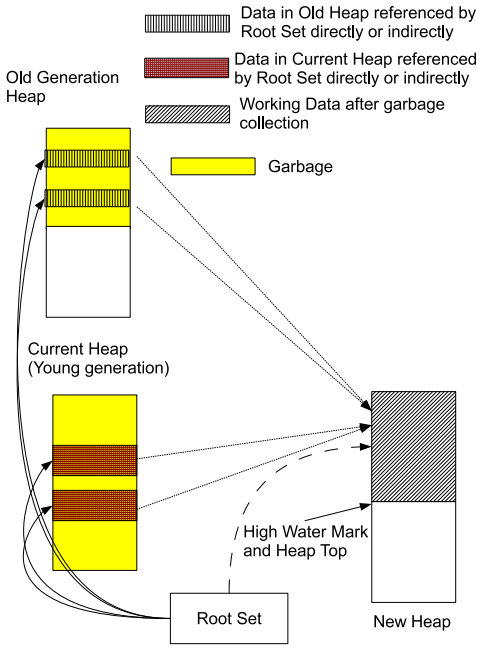

Figure 3.12: Memory movement in major collection

instance for each scheduler, every scheduler can allocate memory from a separate allocator instance. Synchronization is still required if memory blocks can be transferred between schedulers and a scheduler has to deallocate a memory block that was allocated by it, such as the memory blocks for messages in the Erlang VM. When a memory block is deallocated, usually it is not returned back to the OS immediately, but is put into a pool of free memory blocks associated with the allocator which allocated the block earlier. The memory blocks in the pool are organized in a form like tree, bucket or list. If several threads can insert and remove memory blocks from the pool simultaneously, synchronization is needed to protect the structure of free blocks.

The time to allocate or deallocate a memory block is dependent on its position in the free memory pool. For example in a tree if the block suitable for a memory allocation request is at the bottom of the tree it will take a longer time to find it. When there are more instances for a type of allocator, the tree depth can be smaller since free memory blocks are distributed to more pools. It can result in faster memory allocation or deallocation time.

Chapter 4

Evaluation and Analysis

4.1 Experimental Methodology

The performance of an Erlang application is dependent on the configuration of the Erlang runtime system. The VM can be fine-tuned for different applications. There are many parameters that can be adjusted. For instance, since the default process heap size is small, if an application consumes a large amount of memory, there will be many garbage collections. If a larger initial heap size is used, the number of garbage collections can be reduced and the performance is improved.

In this project, we don't attempt to fine-tune the performance for each application. Most of the time, we test with the default setup except scheduler binding and multi-allocator features. Each Erlang scheduler is bound to a different core in the tests. A bound thread cannot be moved to different cores by the scheduler of the OS, which can prevent two Erlang scheduler threads from being assigned to the same core by the OS scheduler. Some *alloc_util* memory allocator types can be configured one instance per scheduler for each type. For some memory-intensive applications, one instance per scheduler can reduce lock contention during memory allocation. But it introduces more memory consumption, and may decrease the performance for some other compute-intensive applications. The default configuration is that there is one instance per scheduler for each *alloc_util* allocator type except *temp_alloc* and *ll_alloc*, when the scheduler count is less than or equal to 16. We set it to one instance per scheduler even when the number of schedulers is greater than 16 for some benchmarks that can benefit from it.

There are two versions of Erlang Virtual Machine, SMP (parallel) and non-SMP (sequential). Usually, the parallel VM with one scheduler is slower than the sequential VM because of synchronization overhead and other differences in the structure of the program. Speedup is normally calculated against the performance of a sequential version of the same program, which is *absolute speedup*. In our test, we also use *relative speedup*, the speedup computed comparing to execute time on the parallel VM with one scheduler.

TILEPro64 is used as a platform for performance evaluation. Some other computer

systems with different processors are also used when there is a need to compare performance differences. All the experiment results described in the remainder of the chapter are measured on TILEPro64 unless otherwise specified. Before presenting the evaluation results, there is a short discussion of the metric that is used as the average value of execution time observations, description of tools and an introduction of benchmarks.

4.1.1 Variability of Execution Time

Performance analysis is a non-trivial task. Contradictory conclusions might be drawn by changing a seemingly innocuous aspect of the experimental setup [30]. The variation of execution time for native programs with the same input are significant on multi-core architectures [28], and it is likely to be more severe on many-core processors. Since Erlang applications are run on a virtual machine, there are more factors that can cause the execution time to vary.

The execution time can be affected by other programs running concurrently with the program that we are measuring. This effect can be reduced by closing other unrelated applications. But there are still some background tasks executed by the operating system. When there are many background tasks in a period, long execution time is observed, and when there are few background tasks, the execution time is shorter.

This imposes a great challenge on the performance analysis of many-core systems, especially scalability. For instance, if we want to compare the execution time on 56 cores with that on 60 cores the difference is only about 1/15 ideally, and it is worse if the execution time on 59 cores is compared with that on 60 cores. The execution time on 60 cores should be large enough so that we can isolate the impact of other tasks, and otherwise we need a huge number of tests to make a reliable conclusion. If the execution time on 60 cores is large, it is likely to take a very long time to execute on 1 core. When a benchmark is configured to run a long time, its size like memory consumption may be bloated, and it doesn't represent an ordinary condition. For example, the number of processes in a benchmark might be increased to make it run longer. With more processes, more memory is used. The system may be working in an extreme condition that there is poor cache performance because of large memory footprint.

The variation of execution time can also be affected by the cache memory system. All the Erlang modules are loaded before they are run. If an application is run for the second time it is likely to be faster, because its instructions and data were brought to the cache when it was run for the first time. To avoid this effect, after each test run the Erlang VM is exited and re-launched during our test. Another alternative way is to run an application several times and record the time spent on each test run except the first one. The first test run warms up the caches. We use the former method because for many benchmarks each test run may take several hours when it is executed on one core. The latter approach is more time-consuming.

On many-core systems, shared memory is protected by locks. Lock contention can introduce variation on the execution time depending on where the contention occurs and how many cores are contending. Variable time can be spent on lock operations. Erlang processes are synchronized by message passing. Depending on when messages are received, the execution time can vary. When receiving a message, if there is no matching message in its mailbox, a process will be suspended. When a matching mes-

sage is at the head of the mailbox, it is processed immediately. The whole mailbox has to be traversed if the only matching message is at the bottom of the mailbox. When there are many cores, the arrival time of messages is likely to vary.

There are also many other factors that cause the variation of execution time. We are not going to investigate all the causes of variation in this project. But because of variability we need a proper metric to represent the average value of execution time spent on a number of test runs.

In practice we observe non-negligible variation of execution time for Erlang programs. To get more reliable results, every benchmark is tested several times with the same input. The average of the observed values is usually represented by arithmetic mean which is obtained by taking the sum of all observations and dividing the sum by the number of observations. The mean is easily affected by outliers which are extremely small or large values in a sample. We notice that single outlier can make a considerable change in the mean execution time especially when the number of cores is very large. Thus we use sample median instead. Sample median is the middle value when the observations are arranged from the smallest value to the largest value. When the number of observations is even, median is the mean value of the two middle observations. Median is resistant to outliers. It is more representative than mean if the observed values are skewed or their distribution is biased [20]. Sample median is also recommended in [28] to report execution time for multi-core programs, and the standard SPEC benchmark suites use median.

4.1.2 Factors Affecting Speedup

There are many factors that either benefit or limit the speedup of the Erlang virtual machine on the many-core processor TILEPro64.

The pros for speedup on the many-core processor are as follows.

- The programs can be executed in parallel.
- When there are more schedulers, more private L1 caches can be utilized since they are bound to different tiles/cores. In the other words, an L1 data or instruction cache serves fewer Erlang processes. The hit rate on L1 caches is likely to increase. Erlang processes with the same priority are executed in round-robin order. If a data cache cannot hold the stacks and heaps for all processes, when a process is executed its data which were brought into the cache during its previous time slice might be taken out and have to be brought back from main memory. A memory block is taken out when a new bock is going to be stored in the same cache line.

If the benefit of more L1 caches prevails, the speedup can be greater than the number of cores used, i.e. super linear.

The cons for speedup are:

- Many programs include sequential part that cannot be run in parallel.
- It is possible that a benchmark doesn't have enough parallelism to fully utilize all cores, particularly when the core count is large.

- The workload might be not evenly distributed to all cores by schedulers.
- Erlang processes can be migrated to other cores because of workload balancing. This might introduce more cache misses, since after every migration, the code, stack and heap of the process have to be transferred to the new tile.
- By default the heap of an OS process is hashed-for-home. When there is only one scheduler, it uses L2 caches of all tiles which form a common L3 cache. With multiple schedulers, they all share the L3 cache. Since all the schedulers run simultaneously, more memory is needed or accessed at any time. The performance of the L3 cache is likely to be worse because of contention.
- Synchronization cost is another important contributor of slowdown. When there is only one scheduler, the locks are not contended and synchronization cost is the pure overhead of locks. If there are many schedulers, they may contend for locks and extra penalty for contentions is introduced [18].

The above pros and cons are the main factors that can affect the speedup. There are definitely other factors that can affect the speedup, for instance contention on the interconnection networks.

4.1.3 Methods

To analyze the performance of Erlang applications, we need both tools for measuring the Erlang VM and the Erlang code running on the VM. The performance of memory subsystem especially the cache system can be measured with system level profiling tools, such as Oprofile for TILEPro64, CodeAnalyst for AMD processors and VTune for Intel processors. A profiler gathers information, such as frequency and duration of function calls and memory system usage, as a program executes.

The lock contention times and duration can be measured with lock profiler or lock counter which is a tool in Erlang/OTP. The time duration spent on a lock is accumulated by the profiler. Thus it cannot provide accurate information about how much time is spent on a lock for each individual scheduler thread. Most Erlang profilers are intrusive in that they have big impact on the performance of the applications that they are profiling. For instance, a profiler may lock some data structures to atomically get a sample.

The balance of workload can be indirectly investigated by checking the migration of processes and the state of schedulers with tracing and other profiling tools. For example, when there is no work, a scheduler thread is in waiting state or even blocked.

4.1.4 Benchmark Programs

Most benchmark programs used in this project are provided by the Erlang/OTP team. A short description for each benchmark is as follows.

- Mandelbrot Set Calculation - The program calculates Mandelbrot set. Complex value c is in Mandelbrot set if when starting from $Z_0 = 0$ and applying the equation $Z_{n+1} = Z_n^2 + c$ repeatedly, the absolute value of Z_n never exceeds a certain

number. In practice, the number of iteration is limited, and it is 255 in this benchmark. The benchmark takes two arguments, size of image in pixels and number of processes running Mandelbrot set calculation. Each pixel represents a complex value, or a point in the complex plain. Every process checks whether each pixel is in the Mandelbrot set for a different image independently. It is an embarrassingly parallel workload. There is no dependency or communication between processes doing the calculation, except that the main process spawns these processes and waits for them to finish sequentially. This benchmark is compute-intensive and has little memory footprint.

- Big Bang - Big Bang spawns N processes, each of which sends a ping message to every other process, and replies if a ping message is received. The ping message is a tuple consisting of the atom *ping* as the first element and the process ID of the sending process as the second element. The response is a message that is similar to the ping message except the first element is the atom *pong*. The messages are very short. Besides sending and replying messages, the processes don't do any useful work other than call a built-in function to get their process IDs. All the processes are interdependent. A process finishes its work after all its ping messages are replied. But some other processes may finish later. As a result it has to keep replying ping messages from other processes.

- Erlang Hackbench - Hackbench is a benchmark for testing the Linux scheduler. It simulates a chat room, in which each client sends a message to each server in the same group. The program creates N groups of processes. In a group, there are 20 listener processes and 20 writer processes. Each of the 20 writers writes 100 messages to each listener. Processes in different groups are independent. Erlang's version of Hackbench is similar except with adjustable message number.

- Random - The main process of the benchmark spawns a number of processes which is specified by user. Each process randomly generates 100000 integers and appends them to a list. Then the list is sorted and split into two lists. After that the first element of the second list which is a middle value of the original list is returned to the main process. This benchmark has big memory footprint.

4.2 Results and Analysis

4.2.1 Mandelbrot Set Calculation

The Mandelbrot set calculation benchmark contains a balanced workload. In fact, it is not a parallel workload but several sequential computations done in parallel. For a true parallel workload, all the processes should operate on separate parts of an image rather than on separate images. This benchmark is a compute-intensive program which is not in the typical domain that Erlang is used. Erlang is designed for applications with a lot of communications and concurrency. This benchmark is used because it is very likely to show the best scalability of the Erlang VM. Thus the result of this benchmark is closely investigated in the remainder of this subsection.

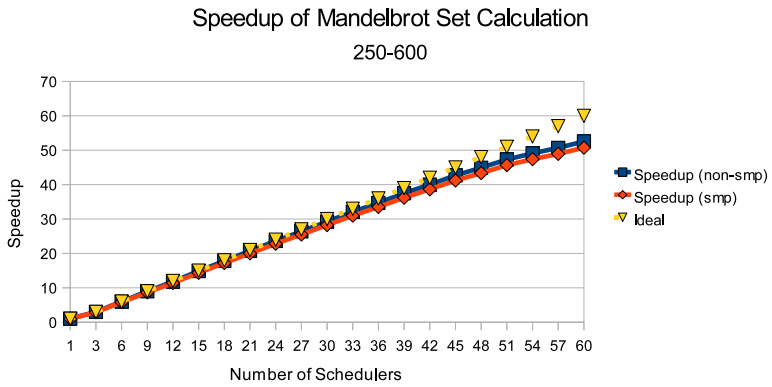

Figure 4.1: Speedup of Mandelbrot Set Calculation 250-600 on TILEPro64

Figure 4.1 shows the absolute and relative speedup of the Mandelbrot set benchmark executed on the parallel Erlang VM with different numbers of schedulers. The x axis is the number of schedulers, and the y axis is the speedup. *Speedup(non-smp)* is the speedup computed by comparing the execution time on a number of cores with the SMP VM to the execution time on one core with the non-SMP or sequential VM, which is the absolute speedup. *Speedup(smp)* is the speedup when the base is execution time on one core (scheduler) with the SMP VM. The sample size is 2, which means every point in the graph is an average of 2 test runs. Image size is 250 * 250 pixels. 600 Erlang processes are spawned to execute independently (600 images).

With 250*250 pixels per image and 600 processes, this benchmark scales very well. It achieves speedup more than 50 at 60 cores. Each process in the benchmark is independent and has the same workload. The result indicates the scheduling algorithm of the Erlang VM achieves good performance with processes that have evenly distributed load.

Figure 4.2 shows the relative speedup of the benchmark with 100*100 pixels per image, 240 processes and 250*250 pixels per image, 180 processes. The sample size is 10, or every point in the graph is the median value of 10 observations. The scalability is not as good as the previous one. The speedup at 60 cores is about 43.

The median values of the execution time on 1 core and 60 cores are shown in Table 4.1. The time in the second row is for the SMP VM with 1 scheduler. Comparing the workload of 180 processes to that of 600 processes with 250*250 pixels per image, the ratio of is about 0.3 ideally since every process has the same workload. The actual execution time with 180 processes on 60 cores to that with 250 processes is 0.343, while the proportion of execution time on 1 core is about 0.3 which is linear to the change of workload. Thus the performance deteriorates when the number of schedulers increases with 180 processes.

Figure 4.3 is a snapshot of the profiling result from the Erlang concurrency profiling tool Percept for the Mandelbrot set calculation benchmark with 240 processes and 100*100 pixels per image on the SMP VM with 1 scheduler. The green area in a bar

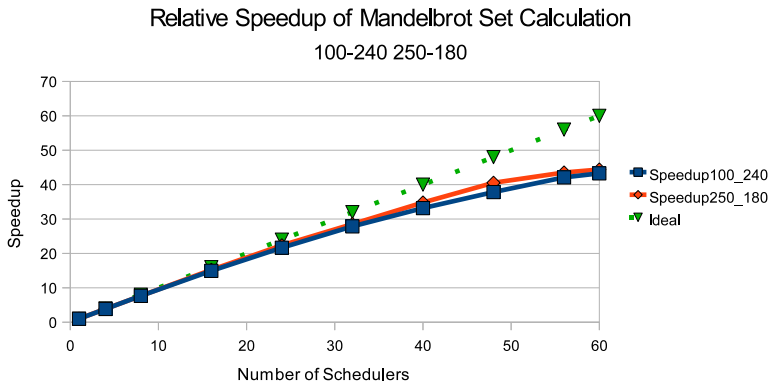

Figure 4.2: Speedup of Mandelbrot Set Calculation on TILEPro64

Schedulers	250-600 (s)	250-180 (s)	100-240 (s)
1	6123.783	1836.562	394.031
60	120.728	41.379	9.107

Table 4.1: Execution Time of Mandelbrot Set Calculation

means the corresponding process is runnable (or running), while the white area means the process is suspended, for example due to waiting for a message. The upper part of the graph shows how many processes are runnable (or running) at an instant of time. The bottom part shows the status of each process. The first process in the graph is the main process that spawns all other processes doing the calculation and waits for them to finish. On 1 scheduler, all processes finish their work nearly at the same time (actually their last time slices are finished sequentially).

The profiling result from Percept on 60 schedulers with 240 processes and 100*100 pixels per image is shown in Figure 4.4. The result indicates that some processes finish

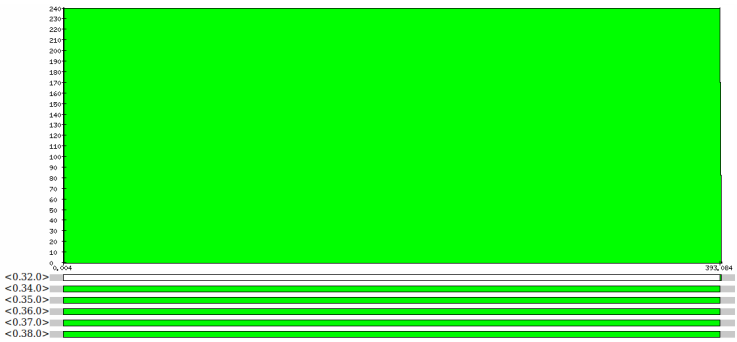

Figure 4.3: Mandelbrot Set Calculation 100-240 on 1 scheduler

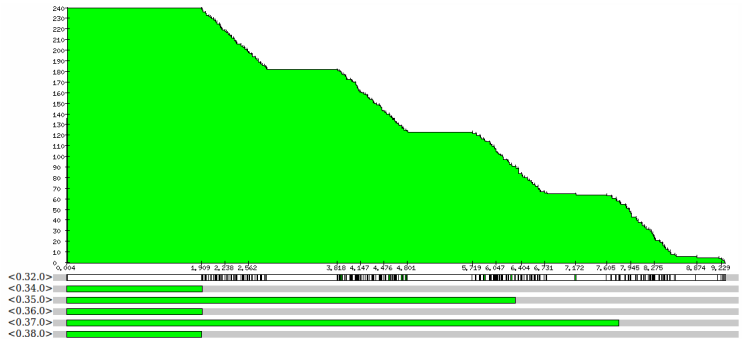

Figure 4.4: Mandelbrot Set Calculation 100-240 on 60 schedulers

execution much earlier than others. We can see steps in the graph. In each step, about 59 processes finish their execution. At the end, there are few processes, which are much fewer than 60, left to be finished.

By examining tracing result, we find that when running the benchmark with 240 process and 100*100 pixels per image on 60 cores the execution time is too short (about 9 seconds) to trigger the workload balancing mechanism as stated in section 3.3. As a result, the schedulers except the one that the main process resides at only can steal the processes. A scheduler steals a process when it is out of work. It steals another process only after the old one has completed its execution. Thus a stolen process occupies the whole scheduler or core, and finishes execution much faster than the processes that are not stolen since those processes share the same core.

Although this doesn't affect the total execution time much, stolen processes behave like having a higher priority than the processes that are not stolen. It breaks the fairness provided by the round-robin algorithm. The period of balance check is the time taken by a scheduler to execute 2000*2000 reductions by default. It is more reasonable that the period reduces as the number of schedulers increases, since when there are more cores more work can be executed in a period. It may have a low limit because if the period of balance check decreases less time is spent in executing useful work. There is a trade-off between the work balance and efficiency.

Figure 4.5 shows the number of schedulers which are not in waiting state starting from the time when the first worker process that does the Mandelbrot set calculation is spawned by the main process. It gives the reason why the scalability with 240 processes is not as good as with 600 processes, because not all the schedulers are working for the whole time. At the beginning all the schedulers except the one that executes the main process are in the waiting state. The waiting schedulers are woken up one by one. This also causes processes to be finished at very different time. Since the total execution time is short, the slowly ramping up of the number of schedulers has big impact on the total performance, while it has less effect with 600 processes. By calculating the area under the line representing the number of schedulers in Figure 4.5, then dividing it by 60, we can roughly get the performance increase when all schedulers are active at the beginning, which is 1.185. Multiplying it with the actual speedup of 43.267, the

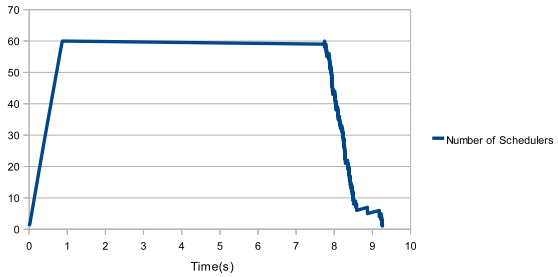

Figure 4.5: Number of scheduler 100-240

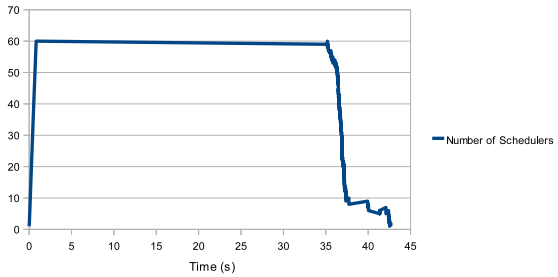

Figure 4.6: Number of Scheduler 250-180

result is 51.225. That is similar to the performance when there are 600 processes and 250*250 pixels per image.

As shown in Figure 4.6 with 180 processes and 250*250 pixels, the execution time is longer and the time of waking up schedulers has less effect on the total performance. But this benchmark has fewer processes and more work per process, and therefore the tail is longer.

The threshold for waking up a scheduler is configurable when compiling the VM, by default 50*2000 reductions. The counter in a scheduler for waking up other schedulers is increased in a scheduling slot proportional to the length of the run queue and the actual reductions executed between the current scheduling time slot and the previous one. Thus the schedulers are mainly woken up by the scheduler where the main process spawns other processes. This benchmark will achieve a better performance if the configured threshold is lower. Thus it is not a big issue. The benchmark doesn't scale well because it doesn't have enough workload.

The ideal speedup is 60 and it is 17.1% higher than 51.225. The rest of the time is likely spent because of lock contention or cache performance deterioration. Figure 4.7 shows the lock profiling result. The lock profiler is a specially compiled VM. It is intrusive and introduces big overhead to count the lock conflicts. The execution time increases from 9.107 seconds to 36.144 seconds, which is 2.969 times larger.

In the result, a lock named *gc_info* stands out. This lock protects shared variables storing statistical data about the times of garbage collection performed and the total size

```
3> lcnt:clear(),mbrot:go(100,240),lcnt:collect().
36143.563
ok
4> lcnt:conflicts().
           lock  id   #tries  #collisions  collisions [%]  time [us]  duration [%]
          -----  ---  -------  -----------  --------------  ---------  -------------
        gc_info    1  9362410      9319477         99.5414 1502993801     4156.2056
      run_queue   60    98572          123          0.1248       4610        0.0127
      proc_msgq  268     1466           40          2.7285        531        0.0015
 alcu_allocator  362 18740345           36          0.0002        197        0.0005
      proc_link  268     1220           24          1.9672        122        0.0003
      proc_main  268    36784            8          0.0217        114        0.0003
    proc_status  268 18834281           75          0.0004         55        0.0002
```

Figure 4.7: Lock Conflicts Mandelbrot Set Calculation 100-240

of memory reclaimed. The lock is global and all the schedulers contend for the lock when they are going to update the information. There are many garbage collections, because the benchmark includes many arithmetic operations which are not allowed to allocate heap fragments and when there is insufficient heap memory for storing calculation result a garbage collection is performed.

In Erlang, variables are immutable. A statement like $x = x + 1$ which is legal in the C programming language is not allowed in Erlang, since the value of x cannot be changed. As a result, in this benchmark for each pixel every intermediate Z_n requires a new variable to store its value. If a pixel is not in the Mandelbrot set, the equation mentioned in Subsection 4.1.4 has to be applied 255 times, and each time a new variable is used which is a complex value. For an image with 100*100 pixels, there are 10000 pixels. Thus a process needs a lot of memory to store these variables. The size is much larger than a process' initial heap size. Every time when the heap is full, a garbage collection is performed. The garbage collection can free nearly all the space of the heap, because all the variables are only used once. As soon as Z_n is calculated, $Z_{n-1}$is not required any more, and becomes garbage. Memory usage of a process' heap keeps growing when new intermediate variables are generated. After a garbage collection, it becomes almost empty. The procedure repeats until the process finishes all the calculations. This interesting phenomenon suggests that an algorithm that is compute-intensive if implemented with some other languages can turn into memory-intensive or garbage-collection-intensive if it is implemented with Erlang.

The lock profiler counts the number of lock acquisition tries and collisions, and also measures the waiting time spent on each instance of a lock class. The *time* in Figure 4.7 is an accumulation of all waiting time spent on different scheduler threads for that type of locks. The *gc_info* lock type has only one instance. It has a high collision rate 99.5414% because it is global. 4156.2056% of the total execution time is consumed because of collisions of this lock, which means for 60 schedulers, the average is about 69.27%. This result doesn't provide us accurate information about the extra time spent on locks when the benchmark is run on a normal VM. But it indicates the lock protecting the updating of garbage collection statistical information may have big impact on the total performance, and the performance can be improved by reducing the lock contention or lock overhead. The *gc_info* lock is implemented with a spin lock which on TILEPro64 is based on Pthread spin lock. Replacing it with the queuing lock might improve the performance when the number of schedulers (cores) is high.

Schedulers	1	60
Bundles / cycle	0.600	0.547
Instruction cache stall / cycle	0.078	0.083
Data cache stall / cycle	0.063	0.099
L1 data cache load miss rate	0.012	0.029
L2/L3 data cache load miss rate	0.112	0.123
L1 data cache store miss rate	0.022	0.065
L2/L3 data cache store miss rate	0.069	0.032
Data TLB miss rate	0.0001	0
Conditional branch mispredict rate	0.334	0.302
Indirect branch mispredict rate	0.594	0.601

Table 4.2: Profiling Result

Table 4.2 shows the profiling result of the benchmark with 100*100 pixels per image and 240 processes from the system profiler, a customized version of Oprofile for Tilera processors. The instruction bundles executed (retired) per cycle with 1 scheduler is about 1.097 times as many as with 60 schedulers. With 60 schedulers, the stall of execution pipeline due to instruction and data memory operations is also larger. It also has higher L1 data cache load and store miss rates, but lower L2/L3 data cache store miss rate. This is very likely caused by lock contention. For example, the spin lock protecting the updating of garbage collection information. The critical section, which includes modification of two global variables, is very short, and therefore the lock owner releases the lock quickly. When the lock is released, many threads will contend for the lock. They use atomic test-and-set instruction to read the old value and write a value one to it. Each write (store) operation will cause the copies of the lock in other tiles' L1 caches to be invalidated. As a result, the miss rate on L1 caches increases. All store operations writes through new values to the L3 cache, and therefore the L3 cache usually contains the newly written values. Lock contention increases the number of stores to the L3 cache. Since the hit rate of the lock is high, the total hit rate increases.

The benchmarking result of the Mandelbrot set benchmark indicates the Erlang VM achieves good scalability with proper workload, which is about 50 on 60 cores. It may be improved if lock contention cost can be reduced, particularly the one protecting updating of garbage collection information. We also suggest the period of balance check should be decreased when the number of cores increases.

4.2.2 Big Bang

The Big Bang benchmark has been tested on a simulated system. It is simulated on Simics, a full system simulator. The simulated system has 128 UltraSPARC T1[26] compatible processor cores running at 75 MHz with the operating system 64-bit Solaris 10. Memory system is not simulated, and memory access time is zero. The benchmark is tested on this platform to gain an insight into how the Erlang VM will scale if there is no memory access latency.

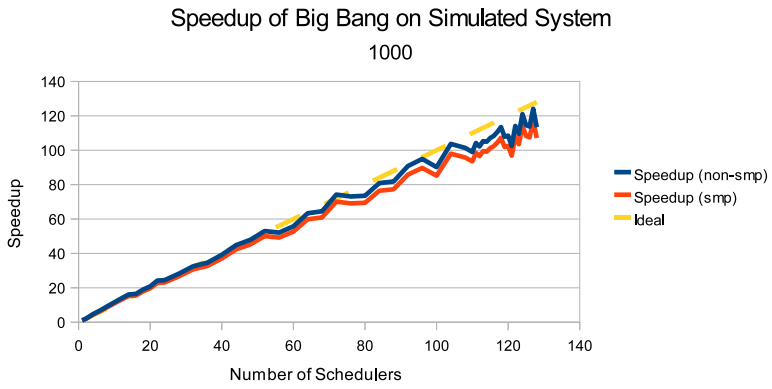

Figure 4.8: Speedup of Big Bang with 1000 Processes on Simulated System

Figure 4.8 is the speedup of Big Bang with 1000 processes on the simulated system. The sample size is only 1, since simulation is very time-consuming. Without averaging, the points vary a lot. Figure 4.8 indicates the Erlang VM scales well with the benchmark Big Bang when there is no memory access latency. The speedup is nearly linear even when the core count is between 64 and 128. It suggests the scheduling algorithm of the Erlang VM achieves very good performance with processes that have similar workload when there is no memory access latency.

The speedup fluctuates more rapidly when the number of cores increases. The execution time of this benchmark is much dependent on the time when messages arrive at the processes. The times of context switching are different if the messages arrive at different time, since a process will be suspended if it has to wait for a message. With more schedulers or cores, the arrival of messages is likely to vary among different test runs.

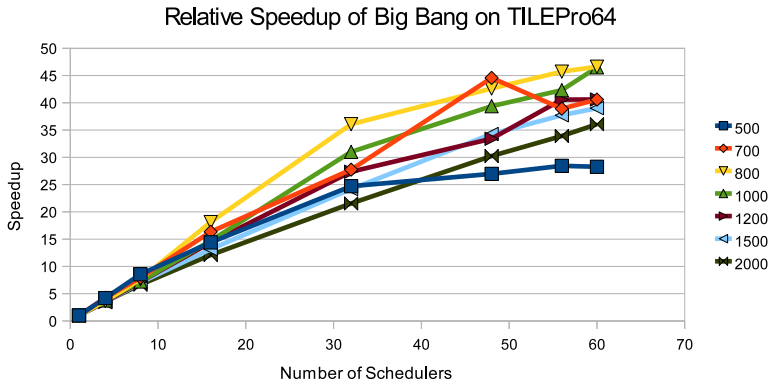

Figure 4.9: Speedup of Big Bang on TILEPro64

Figure 4.9 shows the relative speedup of Big Bang with the sample size 10 on the TILEPro64 board. Among the different process sizes, the one with 800 processes is the best. It achieves the speedup 46.6 at 60 cores. With 1000 processes, the speedup is 46.5 which is very close to the previous one. With 500 processes, the speedup is the worst. The reason is that its execution time is very short (only 1.47 seconds on 60 cores) and it suffers from the time spent on waking up schedulers.

The workloads with 700 (2.98 seconds on 60 cores) and 800 processes (3.69 seconds on 60 cores) are also low. Their execution time is sensitive to cache effect and other influence like running of background tasks. Thus we can observe super linear speedup sometimes. For example, the speedup is super linear with 800 processes on 16 cores and 32 cores. With 700 processes, the speedup jumps on 48 cores. The actual relative speedup for every test run exhibits high variability as shown in Figure 4.10. In the chart, speedup is computed by dividing the execution time of each run to the median value of execution time on one core (scheduler). We can see that at 60 cores the speedup varies rapidly from 36 to 49. This variability decreases when the number of processes or workload increases in general.

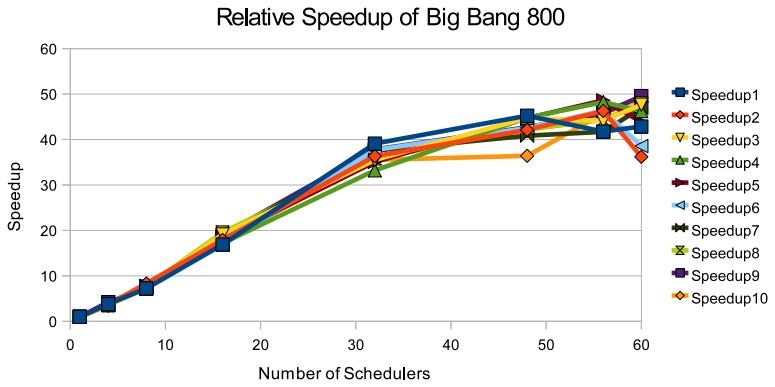

Figure 4.10: Speedup of Each Test Run of Big Bang with 800 Processes on TILEPro64

The performance with the number of processes more than 800 is more representative. When the process count is more than 1000, the scalability decreases which is caused by more synchronization between processes and more memory footprint.

This benchmark seems to be with balanced workload. But actually the workload can vary. This is due to the implementation of message passing. For this benchmark, the workload tends to be less on one scheduler. On one scheduler each process is executed in order. The messages are also sent in a more proper order, and they are processed faster. For example if there are 1000 processes, the 1000th process will receive all pong messages from other processes in the order in which all the processes are spawned. When retrieving, the pong messages are always sequentially matched in the mailbox which reduces the time of message queue traversal. If there are multiple schedulers, the messages are interleaved and the arrival time of messages is more variable. Table 4.3 shows the numbers of reductions executed with different numbers of

No. of procs	Reds 1 sched	Reds 60 sched	Ratio
800	3277560	3414532	1.042
1000	4857191	5252380	1.081
1200	6693462	7386494	1.104

Table 4.3: Number of Reductions with Big Bang

lock	id	#tries	#collisions	collisions [%]	time [us]	duration [%]
alcu_allocator	362	3097055	125556	4.0540	11986480	227.0106
run_queue	60	1400750	19547	1.3955	183442	3.4742
pix_lock	256	2565919	24251	0.9451	175079	3.3158
proc_main	827	1940191	257536	13.2737	166707	3.1572

Figure 4.11: Lock Conflicts Big Bang 800

processes and schedulers excluding the main process. Although reductions are not directly proportional to execution time, we can still get some estimation. From the result, we can see one of the factors limiting the speedup for this benchmark, or maybe this kind of benchmarks with message passing, is that the workload increases as the number of core increases. When there are more processes, messages and synchronization, the workload increases greater.

Comparing the performance of Big Bang with 1000 processes on TILEPro64 to that on the simulated system, the speedup is worse. That is quite reasonable because memory subsystem do have effect on the scalability. At least memory latency is a factor that affects the overhead of locks. The speedup is about 52 at 60 cores on the simulated system, while 46.5 on TILEPro64. But the sample size on the simulated system is small, and two systems have different architectures. Hence this comparison might not be very meaningful.

Figure 4.11 shows the lock profiling result for Big Bang with 800 processes on 60 cores. The accumulated lock collision time of the lock type *alcu_allocator* is about 227% of the total execution time, which is about 3.78% per core. The actual effect of lock contention may be much higher than the average value. It depends on how the total lock contention time is divided among schedulers. It is better if the profiler could provide such information.

Among all *alcu_allocator* locks, the locks protecting some allocators for short-lived data (*sl_alloc*) and Erlang heaps including heap fragments (*eheap_alloc*) have high collision time as shown in Figure 4.12.

This benchmark contains a lot of message passing. When sending a message, two memory blocks have to be allocated, one for the actual message and another for the data structure containing management information for the message. The message is first tried to be copied to the heap of the receiving process. If the receiving process is executing on another scheduler, or another process is coping message to the receiving process, the new message cannot be copied to the heap, and instead a heap fragment is allocated. The heap fragment is allocated with an *eheap_alloc* allocator. There may be one *eheap_alloc* allocator per scheduler which is configurable. If there is one *eheap_alloc* allocator per scheduler, a scheduler always allocates heap memory from

```
6> lcnt:inspect(alcu_allocator).
           lock          id  #tries  #collisions  collisions [%]  time [us]  duration [%]
          -----         ---- -------- ------------ --------------- ---------- -------------
 alcu_allocator    sl_alloc   36482         1194          3.2728    1231433       23.3220
 alcu_allocator    sl_alloc   29096          816          2.8045     687720       13.0247
 alcu_allocator    sl_alloc   35580         1248          3.5076     469034        8.8830
 alcu_allocator    sl_alloc   36892         1246          3.3774     448906        8.5018
 alcu_allocator eheap_alloc    7781          114          1.4651     339474        6.4293
 alcu_allocator eheap_alloc    8681           79          0.9100     337887        6.3992
 alcu_allocator    sl_alloc   52854         2392          4.5257     301815        5.7160
 alcu_allocator eheap_alloc    9465          120          1.2678     280754        5.3172
```

Figure 4.12: Memory Allocator Locks

the allocator associated with it. An *eheap_alloc* allocator is not used for message passing only. It also allocates memory for new main heaps during garbage collections. Heap fragments can also be used with other purposes. After a message sent to another process is retrieved, the process' scheduler calls the corresponding deallocation function of the same *eheap_alloc* allocator which allocated the memory for the message to deallocated the heap fragment. This can cause contention for the lock protecting the memory allocator when the sending process and receiving processes are on different schedulers.

There is a message queue for each process that stores the management data for all the received messages that have not been processed or retrieved by the process. The management data have a fixed size. Every scheduler keeps a preallocated list of free blocks to accelerate the allocation for this type of data. When the list is used up, memory blocks are allocated with an *sl_alloc* allocator for new message management data. The management data are short-lived since when receiving processes have retrieved the related messages they are not needed any more. These preallocated lists and *sl_alloc* allocators are also protected by locks.

The *alcu_allocator* locks are built with Pthread mutex locks. Replacing them with some light-weight locks like queuing locks might improve the performance. Another approach is to reduce the number of collisions. The Erlang/OTP team is going to implement a feature called delayed deallocation, in which a message sender will be responsible for deallocating the message after the receiving process has processed it. Only one scheduler will allocate and deallocate the memory for a message. This optimization can reduce the lock contention. But it may also increase the memory footprint since the messages are not deallocated immediately.

4.2.3 Erlang Hackbench on TILEPro64

Figure 4.13 is the speedup of Erlang Hackbench when there are 700 groups and each writer writes 500 messages to each listener in the same group. The sample size is 10.

The relative speedup is also about 43 on 60 cores. For this benchmark, running on the SMP Erlang VM with one scheduler (1437.163 seconds) is much slower than on the non-SMP VM (662.499 seconds). Table 4.4 shows the execution time of Erlang Hackbench on different platforms with different inputs.

The server in the table has two quad-core AMD Opteron 2376 CPUs with 16 GB RAM. The Laptop has an Intel Core2 Duo T5750 CPU with 4 GB RAM. The Simics row presents the performance on the simulated system as mentioned in the previous

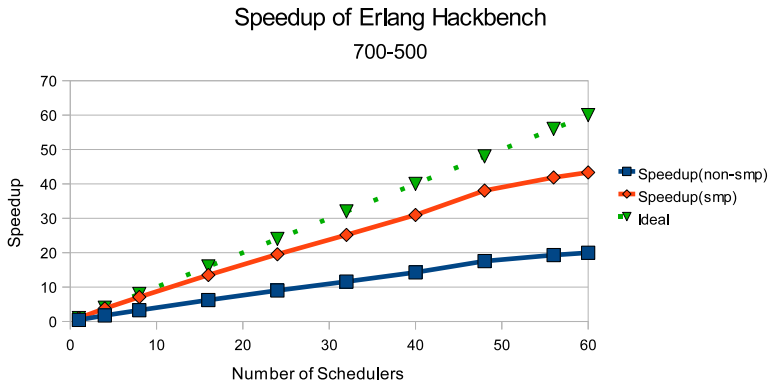

Figure 4.13: Speedup of Hackbench 700 - 500 on TILEPro64

Platform	Program	T (smp 1)	T (non-smp)	Ratio
TILEPro64	EHB100-1000	436.768	185.351	2.36
A server with 64-bit Ubuntu 9.04 Linux	EHB100-1000	25.866	19.3	1.34
A laptop with 32-bit Fedora 11 Linux	EHB100-1000	35.989	16.892	2.13
A laptop with 64-bit Fedora 11 Linux	EHB100-1000	32.678	16.921	1.93
Simics	EHB100-1000	1141.786	774.827	1.47
TILEPro64	EHB700-1000	3023.59	1341.09	2.25
TILEPro64	EHB700-500	1437.163	662.499	2.17

Table 4.4: Execution time on different platforms

VM	SMP with 1 scheduler	Non-SMP
Instructions 32-bit	39909	28911
Instructions 64-bit	36028	27548
Time 32-bit	36.431	17.124
Time 64-bit	32.724	16.749

Table 4.5: Execution time and number of instructions

subsection. The performance of the parallel Erlang VM on the server is much better than on the TILEPro64 board. This may be due to that Opteron has a better single core performance, bigger caches and more atomic instructions for building efficient synchronization functions.

From the table we can see the ratios are lower on 64-bit operating systems. Table 4.5 shows the executed (retired) instructions measured with system profiler VTune for the two version of Erlang VM running Erlang Hackbench with 100 processes and 1000 messages on the same laptop with different versions of Fedora 11. The corresponding execution time of each test run is also included. They are close to the average values in Table 4.4. The sampling rate is 1000 samples per second. The result shows fewer instructions are executed on the 64-bit OS, especially with the SMP VM. Thus it is not very useful to compare the performance between on 32-bit and 64-bit systems. There are fewer instructions retired on 64-bit systems, because a 64-bit instruction can process more data in a cycle. The Erlang VM tries to store more bits in a 64-bit register whenever it is possible. But in average 64-bit Erlang VM is slower than 32-bit VM, since the memory consumption is larger. Not all two 32-bit variables can be combined into one 64-bit variable and processed simultaneous.

We have further investigated the performance of Erlang Hackbench with 100 groups and 1000 messages on the laptop with an Intel CPU, since the profiler VTune works better and provides more information. As shown in Table 4.4, with 32-bit OS the execution time on the SMP VM with one scheduler is about 2.13 times as much as that on the non-SMP VM. Our further profiling result from VTune indicates about 32% of the extra time is spent on the Pthread library, in which Pthread mutex lock takes 90% of the time.

The other extra time is mainly spent on message passing part and the main function of each sheduler thread. These parts also contain other synchronization primitives other than the Pthread mutex lock, including atomic primitives and native spin locks etc. Most of these synchronization primitives are inline functions that cannot be separated with other functions by the profiler. To evaluate the time spent on these synchronization primitives, a VM is built without function inlining. On the VM without inline functions, 30.4% of the time is spent on atomic primitives and other lock functions. The percentage on the normal VM might be lower, since with function inlining the time spent on atomic and lock functions are lower. An inline function can reduce the overhead of calling it. Other functions are also not inlined, and thus this result is more or less significant.

Combing the results of Pthread mutex lock and other synchronization functions implemented in the VM, it suggests that about 60 percent of the extra time is synchro-

lock	id	#tries	#collisions	collisions [%]	time [us]	duration [%]
alcu_allocator	362	332978351	44481362	13.3586	477912510	421.4900
pix_lock	256	285264065	7267946	2.5478	48308089	42.6048
proc_tab	1	57400	30030	52.3171	33929521	29.9238
run_queue	60	21374675	457661	2.1411	30098576	26.5451
proc_msgq	28728	149896026	135586	0.0905	1906569	1.6815

Figure 4.14: Lock Conflicts Hackbench 700-500

nization overhead. With only one scheduler, there is no lock contention, and thus this overhead doesn't include any penalty of lock conflicts. Moreover the program structure is different between the sequential VM and the parallel VM. To run on a many-core system correctly and efficiently, the parallel VM needs more technics. For example, with the parallel VM, a message is first sent to the external public queue of the receiving process' message queue, and later it is merged into the private queue before being retrieved. But with the sequential VM, there is no public queue for each process.

This benchmark includes a large amount of message passing. As a result the overhead of locks related to message passing makes tremendous impact on the total performance. With 700 groups and 500 messages, each of the 20 writers sends 500 messages to each of the 20 listeners. There are more than 150 million messages in total. Profiling result indicates the locks protecting each preallocated list for allocation of message management data as mentioned in the previous subsection contribute a lot to the total execution time along with locks for other memory allocators. The list is used to accelerate memory allocation by assigning one of the blocks in the list to a new request. But in this benchmark, there are too many messages and the list quickly runs out. When new requests arrive, the locks are still acquired to check whether there are free blocks.

The performance can be improved if the lists are checked before acquiring the lock. If there is no free space in a list, the acquiring of lock is skipped, and memory is allocated by an allocator for short-lived data. If there are free blocks in a list, the corresponding lock is acquired. After that the list has to be checked again to see whether there is still some free space since before the lock is successfully acquired the free blocks might be allocated to other threads. The free blocks in a list is allocated only by its owner thread, and other threads return free blocks to the list after messages are retrieved and their management data are deallocated. This feature might be explored to utilize some lock free algorithm instead of lock [39].

The lock profiling result for Erlang Hackbench with 700 groups and 500 messages on the VM with 60 schedulers is shown in figure 4.14. It also mainly suffers performance loss from contention of memory allocator locks since it contains much message passing. The lock for process table, *proc_tab*, has a high collision rate. The process table includes the PCBs of all processes, which is a global data structure. The lock is acquired when spawning (creating) a process. In Big Bang there is only one main process spawning other processes and hence there are fewer collisions on this lock. Each group leader process in this benchmark spawns other processes for its group, and as a result there is much contention for the lock. The contention can be reduced if the process table can be partitioned, for example one table per scheduler. The process table lock is also implemented with a Pthread mutex lock.

The process index lock, *pix_lock*, is used to protect a number of processes in the

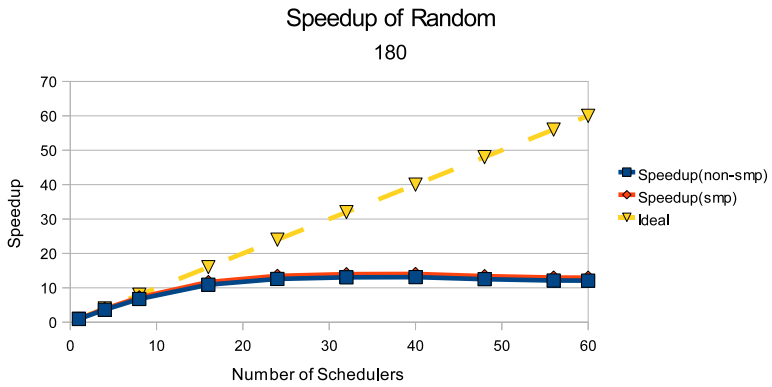

Figure 4.15: Speedup of Random on TILEPro64

process table. It is implemented with the Pthread spin lock on TILEPro64. Each run queue is protected by a *run_queue* lock. For example when a new process is added to a run queue the related lock has to be acquired. The lock is based on Pthread mutex lock. Using some light-weight locks might reduce the overhead for these locks.

4.2.4 Random

Figure 4.15 shows the test result of the Random benchmark with 180 processes. The benchmark scales poorly. There is nearly no performance improvement after the number of cores is greater than 10.

This benchmark is extremely memory intensive. Each process randomly generates a list of 100000 integers, sorts it and splits it. For a small integer it takes four bytes on a 32-bit machine. Each element of a list includes a pointer which is also four bytes on a 32-bit machine. Thus the list is about 800 KB for small integers (big integers in Erlang can be arbitrarily long), and there are 180 lists which is about 140 MB. When there are 60 schedulers, 60 processes can run simultaneously with about 46 MB lists. Much more memory may be needed to store some intermediate results for the list manipulation. Each L1 cache in a tile on TILEPro64 is only 8 KB, and L2 unified cache containing data and instructions is 64 KB. The common L3 cache formed from L2 cache is 4 MB (64 *64 KB). Since hash-for-home feature is enabled for most memory area except stacks, all the L2 caches are utilized even when there is only one scheduler.

When sorting a list, the whole list has to be traversed and a new list is generated. The splitting also needs to traverse the first half of the list. Table 4.6 indicates the benchmark has bad data cache performance.

The lock counting result in Figure 4.16 shows that there is very high contention of locks for memory allocators, which is much higher than for Big Bang or even Erlang Hackbench. Reducing the lock contention can improve the performance.

Schedulers	1	60
Bundles / cycle	0.546	0.477
Instruction cache stall / cycle	0.072	0.077
Data cache stall / cycle	0.114	0.195
L1 data cache load miss rate	0.040	0.052
L2/L3 data cache load miss rate	0.281	0.213
L1 data cache store miss rate	0.271	0.273
L2/L3 data cache store miss rate	0.147	0.147
Data TLB miss rate	0.0009	0.0006
Conditional branch mispredict rate	0.294	0.290
Indirect branch mispredict rate	0.778	0.775

Table 4.6: Profiling Result of Random 180

lock	id	#tries	#collisions	collisions [%]	time [us]	duration [%]
alcu_allocator	98	85574	10877	12.7106	820825583	2579.6084
run_queue	60	367549	587	0.1597	79587759	250.1204
pix_lock	256	1897	5	0.2636	381712	1.1996
gc_info	1	39436	931	2.3608	29647	0.0932

Figure 4.16: Lock Conflicts of Random 180

4.3 Summary

The test results indicate that the Erlang virtual machine scales well on TILEPro64 with normal workload except for an extremely memory intensive benchmark. The scalability is dependent on the characteristics of each application and its input. Maximum speedup of about 40 to 50 on 60 cores is observed in the tests.

The scheduling algorithm is good enough to balance the workload on different cores. The only observed little problem is that when the workload is small and work stealing is the sole method of workload distribution, a stolen process occupies the whole core and behaves like having higher priority, since a scheduler only steals a process when its run queue is empty. We suggest the period of balance check should be reduced when the number of cores increases to achieve more fairness. Moreover when the workload is low, the speedup may be smaller due to the time spent in waking up idle schedulers if they are not working at the beginning.

Synchronization overhead caused by contention is a major bottleneck. The scalability can be improved by reducing lock contention and the the overhead caused by it. We find locks for memory allocators, garbage collection information, process table, run queue and process index have to be optimized. We recommend using some more light-weight locks, such as queuing lock, instead of Pthread mutex lock or simple spin lock whenever it is possible.

Another major problem is that the parallel Erlang VM with one scheduler is much slower than the sequential version when running Erlang Hackbench. Synchronization latency induced by uncontended locks including atomic functions used in synchronization is one of the main causes of the difference. We suggest implementing lock free

algorithm and using locks with lower latency to reduce the overhead.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The upcoming many-core systems will impose a great challenge on software developers. Particularly, the programs developed with conventional languages such as C and C++ will suffer greatly. They have to be rewritten to fully utilize the power of many-core systems. Developing applications on many-core systems is not a trivial work. Tasks running on different cores need to be synchronized. The traditional synchronization methods, such as locks and semaphores, are tedious and error-prone. Great care has to be taken to make the programs deadlock free.

Erlang's message passing mechanism provides a higher level abstract of synchronization. Together with its native support of concurrency, Erlang provides an efficient way of application development on many-core systems. There is nearly no difference between developing applications for single core and for many-core systems. Programmers only have to find out more parallelism for every application. If an application developed for single core systems has sufficient parallelism, it may utilize the power of many-core systems without any change.

This degree project investigated the scalability of the Erlang runtime system which supports the Erlang applications to make full use of many-core systems. Our test results indicate the Erlang VM achieves good scalability with most benchmarks used on a many-core processor, TILEPro64. Maximum speedup from about 40 to 50 on 60 cores is observed depending on the characteristics of the benchmarks. Workload can be well balanced on different cores by the schedulers. A bottleneck of the system is synchronization overhead caused by contention. The scalability can be improved by reducing lock contention. We recommend using more light-weight locks whenever it is possible. Another major problem is that the parallel VM with one scheduler is much slower than the sequential VM when running a benchmark with a huge amount of message passing. Synchronization latency induced by uncontended locks is one of the main causes of the difference. We suggest implementing lock free algorithm to reduce the lock overhead. Several parts of the Erlang VM implementation which can affect the scalability on many-core systems were also studied in this project including scheduling

algorithm, message passing, memory management, and synchronization primitives.

Our result suggests that Erlang is a suitable platform for developing applications on many-core systems. It is ready to be used on these systems and can effectively utilize the power of many-core systems, although the performance of the VM could be further improved.

5.2 Future Work

The results of this project suggest that building scalable memory allocators is very important on a many-core processor with shared memory, especially for message passing. We can try to build a more scalable allocator and investigate the trade-off between memory consumption and scalability. For message passing, the serialization caused by locks can be reduced if the memory blocks for a message are allocated and deallocated by the same scheduler thread. This approach requires that when a message is retrieved by the receiving process, the memory for the message and its management data is not deallocated immediately by the scheduler on which the receiving process resides. It should be deallocated later by the scheduler which allocated them. We have to figure out when and how to deallocate the memory for the sending scheduler.

The scalability of the Erlang VM can be improved by reducing lock contention and the overhead associated with it. The most critical locks are those for memory allocators. They are based upon Pthread mutex locks. We can investigate whether they can be replaced with some lower overhead locks, such as queuing lock. This also applies to other locks implemented with Pthread mutex locks. Native spin lock implementations may also be improved by employing other techniques like exponential back-off. Another more promising method is data partitioning, which has to be tailored to each data structure. For example, the process table is a global data structure. Making the table distributed on different schedulers and consistent globally can greatly reduce the lock contention.

The lock profiler in Erlang/OTP only provides the accumulated collision time for all schedulers on a lock. It doesn't show the actual lock contention effect for each separate scheduler. We can add another feature which accumulates waiting time on each lock for every scheduler.

Reducing lock contention is not sustainable if the number of cores keeps increasing. We can try some new features provided by many-core systems to avoid using locks. For example, one of the on-chip networks in TILEPro64 supports passing short messages directly between tiles. This might be utilized to implement a new message passing mechanism.

Bibliography

[1] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[2] J.L. Armstrong, B.O. Däcker, S.R. Virding, and M.C. Williams. Implementing a functional language for highly parallel real time applications. In *Software Engineering for Telecommunication Systems and Services, 1992., Eighth International Conference on*, pages 157–163. IEEE, 1992.

[3] Joe Armstrong. The development of erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 196–203, New York, NY, USA, 1997. ACM.

[4] Joe Armstrong. A history of erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[5] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall, 1996.

[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, New York, NY, USA, 2009. ACM.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, Washington, DC, USA, 1994. IEEE Computer Society.

[8] Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 2009.

[9] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[10] Christian Convey, Andrew Fredricks, Christopher Gagner, Douglas Maxwell, and Lutz Hamel. Experience report: erlang in acoustic ray tracing. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 115–118, New York, NY, USA, 2008. ACM.

[11] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*, chapter 13. McGraw-Hill Higher Education, second edition, 2001.

[12] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[13] Ulrich Drepper. Futexes are tricky. August 2009.

[14] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the 2002 Ottawa Linux Summit*, 2002.

[15] Scott Lystig Fritchie. A study of erlang ets table implementations and performance. In *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, ERLANG '03, pages 43–55, New York, NY, USA, 2003. ACM.

[16] Bogumil Hausman. Turbo erlang: Approaching the speed of c. In *ICLP-Workshops on Implementation of Logic Programming Systems*, pages 119–135. Kluwer Academic Publishers, 1993.

[17] Pekka Hedqvist. A parallel and multithreaded erlang implementation. Master's thesis, Computer Science Department, Uppsala University, Uppsala, Sweden, June 1998.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2A: Instruction Set Reference, A-M*, December 2009.

[20] Raj Jain. *The Art Of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley Computer Publishing, John Wiley and Sons, Inc., 1991.

[21] Erik Johansson, Mikael Pettersson, and Konstantinos Sagonas. A high performance erlang system. In *PPDP '00: Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 32–43, New York, NY, USA, 2000. ACM.

[22] Erik Johansson, Mikael Pettersson, Konstantinos Sagonas, and Thomas Lindgren. The development of the hipe system: design and experience report. *International Journal on Software Tools for Technology Transfer (STTT)*, 4:421–436, 2003. 10.1007/s100090100068.

[23] Erik Johansson, Konstantinos Sagonas, and Jesper Wilhelmsson. Heap architectures for concurrent languages using message passing. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 88–99, New York, NY, USA, 2002. ACM.

[24] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley and Sons, Inc., New York, NY, USA, 1996.

[25] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[26] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[27] Stefan Marr and Theo D'Hondt. Many-core virtual machines: decoupling abstract from concrete concurrency. In *SPLASH '10: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 239–240, New York, NY, USA, 2010. ACM.

[28] Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. Study of variations of native program execution times on multi-core architectures. In *CISIS '10: Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 919–924, Washington, DC, USA, 2010. IEEE Computer Society.

[29] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

[30] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 265–276, New York, NY, USA, 2009. ACM.

[31] Hans Nilsson, Claes Wikström, and Ericsson Telecom 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.

[32] J. H. Nyström, P. W. Trinder, and D. J. King. High-level distribution for the rapid production of robust telecoms software: comparing c++ and erlang. *Concurr. Comput. : Pract. Exper.*, 20:941–968, June 2008.

[33] Mikael Pettersson. A staged tag scheme for Erlang. Technical Report 2000-029, Department of Information Technology, Uppsala University, 2000.

[34] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for mimd parallel processors. *SIGARCH Computer Architecture News*, 12(3):340–347, 1984.

[35] 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 *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 36–42, New York, NY, USA, 2003. ACM.

[36] Konstantinos Sagonas and Jesper Wilhelmsson. Efficient memory management for concurrent programs that use message passing. *Sci. Comput. Program.*, 62(2):98–121, 2006.

[37] TILERA Corporation. *TILE Processor user Architechture Manual*, November 2009.

[38] Seved Torstendahl. Open telecom platform. Technical Report 1, Ericsson Review, 1997.

[39] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 214–222, New York, NY, USA, 1995. ACM.

[40] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

TRITA-ICT-EX-2011:5

www.kth.se