Erlang - Programming the Parallel World

Shashank Agarwal
shashank@cs.ucsb.edu

Puneet Lakhina
puneet@cs.ucsb.edu

1 Introduction

The speedup in computational power has reached a brickwall in terms of sequential speedup. Microprocessors can no longer execute the same instruction faster and hence in order to provide opportunities for program speedup the microprocessor manufacturers have started supplying chips with multiple processor cores. But the only way now for computer programs to increase their performance is to utilize the multiple CPU cores. This can only be achieved by parallelizing the program execution over these multiple cores i.e. executing instructions on the cores concurrently. Erlang is a programing language that was written with easy concurrency as one of the design goals and is thus ideally suited for the demands of the parallel world. It provides language primitives and thereby a thinking framework to write parallel programs.

The purpose of this study was to not only understand the Parallel programming model provided by Erlang, but also to study the details of its Virtual Machine implementations to understand the techniques used for performance speedup. This report is divided into four sections. In the section 2 we discuss the key features of Erlang which highlight the design choices made by the authors of the programming language. We also bring forward in this section the motivations behind these choices. Section 3 details the various Virtual Machine implementations that have existed for Erlang and the differences between them. We also discuss the High Performance Erlang (HiPE) [6] compiler which compiles Erlang to native code. In section 4 we introduce the concurrency primitives of Erlang and their semantics. In this section we also discuss how the Erlang VM schedules the collaborating processes for execution.

2 Key Features

Erlang is a strict, dynamically typed, functional programming language with support for concurrency, communication, distribution, fault-tolerance, on-the-fly-code reloading, automatic memory management, and multiple platforms [7]. Erlang was created by the Ericsson Computer Science Laboratory at Ellemtel [2] with the primary target being telecommunication equipment. The design goals of Erlang are reflection of the initial problem domain:

- **Concurrency** - Several thousand events, such as phone calls, happening simultaneously.
- **Robustness** - Uptime in telecommunications is a lot more important than domains such as Web/Online services. Thus an error in one part of the application must be caught and handled so that it does not interrupt other parts of the applications. Preferably, there should be no errors at all.
- **Distribution** - The system must be distributed over several computers, either due to the inherent nature of the application, or for robustness or efficiency.

The above concerns have resulted in the following key features:

- **Message Passing Paradigm for Concurrency** - A lot of existing programming languages provide for concurrency through the use of what is known as shared memory. In a shared memory system multiple threads or processes access the same memory and the access to it is synchronized through the use of mutexes or locks. This paradigm of concurrency suffers from a number of problems like deadlock, race conditions etc. In addition it is generally considered hard to get it right [3]. Thus Erlang choose the Message passing approach to concurrency. In this paradigm processes communicate explicitly by passing messages to amongst them. These messages are passed asynchronously in case of Erlang, hence the sender does not need to block. In order to scale the concurrency model it is imperative that the system should be able to support a large no. of processes. Erlang processes are thus ”lightweight processes” or ”green threads” [4] which are implemented and scheduled in the VM and do not map to system level threads or processes. Section three provides a comparative analysis of the speed of spawning an Erlang process as compared to a Java thread and also the number of Java threads that can live in a system as compared to Erlang processes.

  In order to provide the ability to distribute processes over multiple machines, Erlang allows spawning a process on different nodes. Spawning of processes on different CPUs is handled by the system internally.

- **Error Recovery** - In order to build a robust system, it is important to be able to recover from errors and also isolate errors. Erlang provides error recovery by allowing parent processes to receive messages when a child exits unexpectedly. Erlang provides for error isolation by forcing memory to be **immutable**. This allows tracing of errors/bugs easier as only single time variable assignments are possible.

- **Hot Code Replacement** - Since uptime forms one of the primary concerns of a telecommunication system, Erlang provides for a way to replace running system code. This is implemented by the system maintaining a global method table for all the loaded modules, which gets updated when code gets replaced. Thus all future function calls invoke the new method.

3 Erlang Virtual Machine Implementations

Erlang’s execution process is quite similar to any virtual machine oriented programming languages. A compiler compiles Erlang source code to Erlang bytecode which is then interpreted in an emulator. Using the HiPE compiler it is also possible to compile Erlang source code directly to native machine code. Native code can then be combined with emulated code using the HiPE runtime system.

Since the early days of Erlang two implementations of Erlang Virtual Machine have existed **Joe’s Abstract Machine (JAM)** [8] and **Bogdan/Bjrn’s Erlang Abstract Machine (BEAM)**. JAM was the original Erlang virtual machine, inspired by the (Prolog) WAM (Warren’s Abstract Machine).

3.1 JAM

JAM utilized a Stack based virtual machine much like java whereby arguments to functions are passed by explicitly pushing them on the stack and then jumping to the first instruction for the function. JAM’s main purpose was to bootstrap the Erlang development process.

3.2 BEAM

BEAM is the current de-facto implementation of the Erlang system. BEAM was initially an attempt to compile Erlang via C [9]. It made heavy use of named macros functionality of GCC. It was abandoned after benchmarking showed that the resulting code was only faster than VM-based Erlang for small programs [5]. In contrast to JAM, BEAM is a virtual register machine whereby virtual machine registers are used for function calls and parameter passing.

3.3 Function execution in JAM and BEAM

Figure 1 shows the bytecode generated by the different VMs for the factorial program. The overloaded function definitions are referred to as clauses. The function to be executed is semantically based on pattern matching. When for e.g. *factorial(4)* is called first the clause *factorial(0)* is considered, if that doesn't match then the other clause factorial(N) (where N refers to any value is matched), which results in that function being executed. In case of no match an error is thrown. In terms of bytecode BEAM and JAM implement it differently. BEAM uses the register x register to pass a single argument and x(0) refers to the first argument.The *test, is_eq_exact* instruction implements the pattern match by comparing if x(0) is equal to 0. In case is_eq_exact is true it proceeds to execute the next instruction otherwise it jumps to the label identified by *(f,3)* i.e. label 3 where the N*factorial(N-1) clause is executed. In this clause after the memory allocation for N, a built in function is used to execute the subtraction "N-1" and the result is stored in x(1) register. After that in order to setup to call *factorial(N-1)* x(0) is backed up into y(0) register and then x(1) is moved into x(0) thus, setting up the registers for calling factorial with argument N-1. After *factorial(N-1)* returns, a built in function is executed to multiple y(0) (which is holding N) with x(0) which holds the return value from *factorial(N-1)*. The result of this gets stored into x(0) which becomes the return value.

In case of JAM the instruction *try_me_else,label_1* is generated corresponding to *factorial(0)* clause. The semantics of this instruction are to try and execute the following instructions but in case of any failure go to label 1 which is the other clause. In case of factorial(0) clause the *arg,0* pushes the first argument onto the stack and then getInt compares the top of the stack to zero. In case top of the stack isn't zero it causes a failure which results in control moving to label 1 where execution continues. Before moving to label 1 the stack is restored.

3.4 High Performance Erlang (HiPE)

HiPE was initiated as a project to develop a just in time native compiler for Erlang [10]. It was based on JAM and although it outperformed both JAM and BEAM for small benchmarks [10], it had problems scaling up to compile large systems (e.g., tens of thousands of lines of code) and since it was implemented in C, it was difficult to rapidly develop and debug new optimizations in it. Thus, it was decided to implement the HiPE compiler in Erlang itself.

The current HiPE system allows for native code compilation and execution by a two pronged approach. It provides an Erlang to native compiler that can be used to generate native code for specific functions and not just whole modules, which are the conventional Erlang compilation units. In addition it provides a runtime system that handles intermixing of native and emulated functions. The intermixing takes care of hot code reloading and also allows for Erlang's per process generational garbage collection. Although, the current system is tightly coupled with BEAM, it can also be used with the JAM machine. Figure 2 illustrates the architecture of the HiPE system.

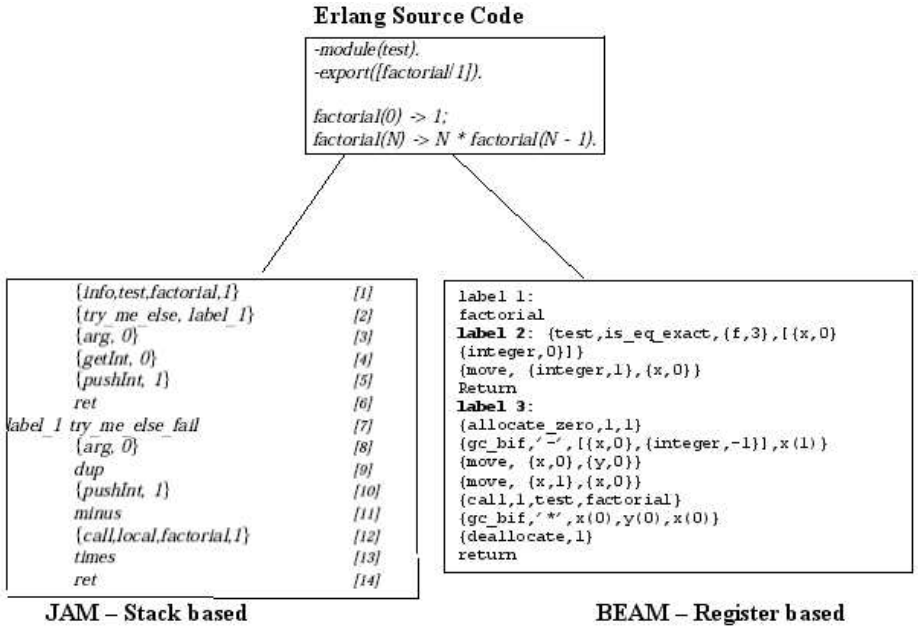

Figure 1: Bytecode generation by JAM and BEAM

3.4.1 HiPE Compilation Process

The Erlang/OTP compiler first performs macro preprocessing, parsing, and some de-sugaring (e.g., expanding uses of the record syntax) of the Erlang source code. After that, the code is rewritten into Core Erlang. Various optimizations such as constant folding, and (optional) function inlining, are performed on the Core Erlang level. After this, the code is again rewritten into BEAM virtual machine code, and some further optimizations are done. The HiPE compiler has traditionally started from the BEAM virtual machine code generated by the Erlang/OTP compiler. The BEAM code for a single function is first translated to ICode, an assembly-like language with a highlevel functional semantics. After optimizations, the ICode is translated to RTL (”register-transfer language”), a low-level RISC-like assembly language. It is during this translation that most Erlang operations are translated to machine-level operations. After optimizations, the RTL code is translated by the backend to actual machine code. It is during this translation that the many temporary variables used in the RTL code are mapped to the hardware registers and the runtime stack. Finally, the code is loaded into the runtime system.

HiPE has extended the Erlang/OTP runtime system to associate native code with functions and closures. At any given point, a process is executing either in BEAM code or in native code: this is called the mode of the process. A mode switch occurs whenever control transfers from code in one mode to code in the other mode, for instance when a BEAM-code function calls a native-code function, or when the native-code function returns to its BEAM-code caller. The runtime system handles this transparently, so it is not visible to users, except that the native code generally executes faster.

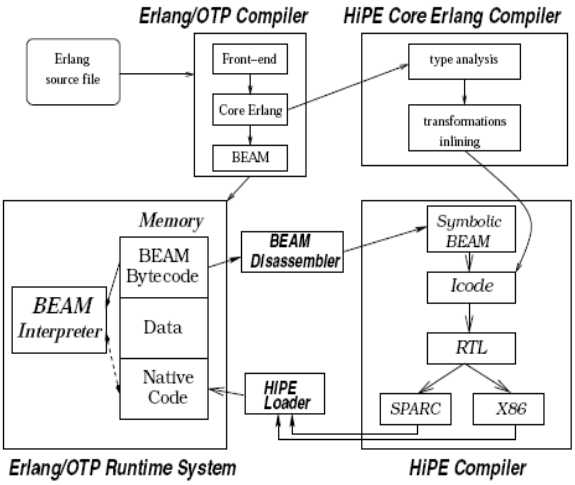

Figure 2: HiPE Architecture

4 Concurrency

Erlang's approach to concurrency is based on the Actor Model of computation, that treats "actors" as the universal primitives of concurrent digital computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received [1]. Erlang's processes are the actors that communicate using message passing.

4.1 Processes:

Erlang's processes are VM level processes, also known as "green threads" that do not map to system level processes or threads. Typically the Erlang VM creates 2 threads per core, one for I/O and the other running the process scheduler. Erlang's processes have the following features:

1. **Small overhead** - The size of a Process Control Block(PCB) is only 300 bytes, which makes it light in comparison to 1024 bytes PCB of a Unix process. The small overhead is essential for an Erlang process as the concurrency model supported by Erlang typically demands the creation of a huge no. of processes. Fig 3 shows some important fields of an Erlang PCB.

2. **Independent Memory** - Each Erlang process has its own stack and heap. No memory sharing occurs between processes. Hence the PCB of each process has heap and stack pointers. This scheme although high on memory usage, prevents memory corruption by one process from cascading into system memory corruption. This is extremely important for Erlang's fault tolerant requirement.

3. **Garbage Collection** - Since Erlang's processes do not share a common heap, the garbage collection is on a per process basis and is performed by the process itself. This allows Erlang to guarantee a *Soft Real Time* behavior as no other process is affected by the memory utilization pattern of an unrelated process.

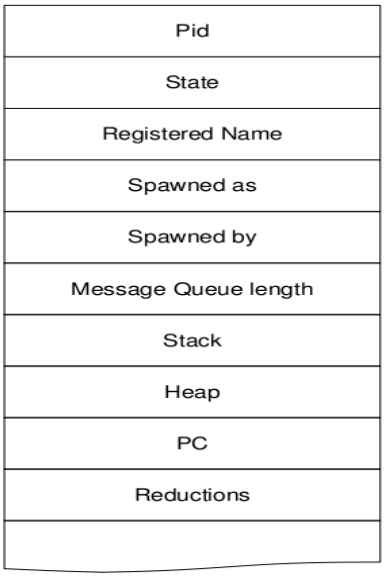

Figure 3: Process Control Block for Erlang process

4.1.1 A comparison of Erlang's processes and Java Threads

Threads in Java are also green threads, i.e. they are created and scheduled inside the VM. Although some of the current Java implementation do create one system thread per Java thread, this is not a VM requirement. Table 1 demonstrates the creation times of various no. of concurrent processes and threads in case of Erlang and Java respectively. Number of messages shown in the table are only relevant to Erlang and are used to highlight the message passing overhead which erlang will have in order to achieve concurrency.

4.2 Message Passing

Message passing involves 2 processes, a sender and a receiver. The sending process first copies the message into a buffer from where it is copied into the heap of the receiving process. Each process has a queue of messages which have been sent to it and are waiting to be processed. This queue is maintained as a list on the heap of each process. After the message has been copied to the receiving process, the message is linked onto the end of the list by copying the pointer of the heap location to the message queue. If the receiving process is currently in the suspended state sender's action also involves waking up the process and adding it to the scheduled queue.

4.3 Concurrency Primitives

In order to implement the message passing paradigm, Erlang provides three concurrency primitives:

- **Spawn(M, F, [A1, A2, ..., An]):** creates a new *parallel* process. This process evaluates the function F in module M with arguments A1, A2...An. Example:
  Pid = spawn(fun() -> wait() end).
- **Send a Message:** *Pid ! Msg*. Sends a message (value of variable Msg) to process (Pid) asynchronously.

Table 1: Process creation time

No. of Processes	No. of Messages	Java(time in msecs)	Erlang(time in msecs)
1000	100	132	67
2000	100	328	83
3000	100	661	125
4000	100	873	182
5000	100	1294	207
6000	100	1950	261
7000	100	OutOfMemoryError	319
10000	100	OutOfMemoryError	420
15000	100	OutOfMemoryError	620
20000	100	OutOfMemoryError	832
30000	100	OutOfMemoryError	1254

- **Receive a message:** On receiving a message(msg) an action is taken by the process.

  receive
  {**msg**} -> **action()**
  end.

  The receiver can filter which messages it wants to process. This is implemented by pattern matching. In the above example the function action() is called only for messages that match the pattern {msg}. Delivered messages are kept in a message queue and are not removed until they match the receiver's message pattern. When a process enters a receive, it gets suspended unless there is a message to be processed. Example:

  receive
  <**pattern 1**> **[when** <**guard 1**>**] ->**
  <**action 1**>**;**
  ...
  <**pattern N**> **[when** <**guard N**>**] ->**
  <**action N**>**;**
  [**after** <**timeout-expression**> **->**
  <**timeout-actions**>]
  end

  As the receiving process has to block until a message that matches the pattern becomes available in the queue, it may have to block indefinitely. In order to provide stricter guarantees of time for message receiving process, a timeout construct is provided. Erlang process waits for the timeout value(msecs) and expires if the message does not arrive for timeout value. The timeout is implemented by a scheduled system interrupts causing a global value to be implemented in the Erlang runtime, which then uses this global value to wake up processes that may have timed out on a receive.

4.4 Process Scheduling:

Process Scheduling in Erlang is based on 4 different queues. The queues are max, high, normal and low in the order of decreasing priority. Each process can be assigned a queue and scheduling in each queue is in round robin fashion. Amongst the different queues the highest priority queue i.e. the max queue is emptied

of all the processes first. This is where the processes with stricter real time requirements get scheduled. After that the high priority queue gets evaluated. Processes from normal and low queue are processed differently. Eight reductions(function calls) are performed in the normal queue and then one reduction is performed in the low queue.

5 Conclusion

Multiple cores have firmly put the burden of performance speedup on the shoulders of Software Developers. It requires a new approach in program design with emphasis on concurrently executing components. Erlang is a programming language that fits well into the parallel scheme of things. Erlang provides language level features for dividing programs into concurrently executing units. In Erlang terms a process forms an execution unit and as our experiments have shown a large no. of concurrent processes can exist in an Erlang VM. Erlang implements the message passing paradigm by whereby processes communicate through explicit message passing. Our discussion on Erlang VMs also highlight the maturity of the Erlang programming language. A stable VM, BEAM in combination with the advanced compilation and runtime facilities provides by the HiPE project has enabled Erlang to become a language ready for industrial grade software.

References

[1] Actor model of computation. `http://en.wikipedia.org/wiki/Actor_model`.

[2] Ericsson - erlang. `http://www.ericsson.com/technology/opensource/erlang/`.

[3] The free lunch is over: A fundamental turn toward concurrency in software. `http://www.gotw.ca/publications/concurrency-ddj.htm`.

[4] The free lunch is over: A fundamental turn toward concurrency in software. `http://en.wikipedia.org/wiki/Green_threads`.

[5] Frequently asked questions about erlang. `http://erlang.org/faq/implementations.html#8.5`.

[6] High performance erlang. `http://www.it.uu.se/research/group/hipe/`.

[7] J. Armstrong. Programming Erlang: Software for a Concurrent World. 2007.

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

[9] B. Hausman. Turbo Erlang: Approaching the speed of C. *Implementations of Logic Programming Systems*, pages 119–135, 1994.

[10] E. Johansson and C. Jonsson. Native code compilation for Erlang. *Uppsala master thesis in computer science*, 100.