Comparison of Erlang Runtime System and Java Virtual Machine

Tõnis Pool
University of Tartu
tonis.pool@gmail.com
Supervised by Oleg Batrashev

Abstract

This report gives a high level overview of the Erlang Runtime System (ERTS) and the Java Virtual Machine (JVM), comparing the two in terms of overall architecture, memory layout, parallelism/concurrency and runtime optimisations. More specifically I'll look at the HotSpot JVM provided by Oracle and the default BEAM implementation open sourced by Ericsson.

1. Introduction

There are many different virtual machines in existence nowadays packing a growing number of clever tricks to make them run as fast as possible. The JVM is perhaps the most known virtual machine and one that's received couple of decades' worth of improvements and optimisations.

Somewhat similarly to Java, Erlang is a general-purpose concurrent, garbage-collected programming language that nowadays runs on a virtual machine and has been around even longer than Java. The whole runtime system together with the language, interpreter, memory handling is called the Erlang Runtime System, but the virtual machine is often referred to also as the BEAM.

1.1. JVM

The term Java Virtual Machine specifies 3 different notions - a specification of an abstract stack machine, a implementation of that specification and a running process. The specification makes sure that different implementations are interchangeable, but on other hand isn't very strict about areas such as memory layout, garbage collection algorithms and instruction optimisations, allowing for many different approaches. The HotSpot VM provided by Oracle is the most widely used implementation, which is why it was chosen for comparison.

The JVM works by executing bytecode statements from a class file, generated by compiling the java source code. This indirection is what gives programmers the ability to compile their source code once and then execute it on different platforms that have JVM implementations.

1.2. ERTS

Unlike the JVM, Erlang doesn't have a formal specification for it's runtime nor for the programming language. This makes it difficult to have various implementations, as they all have to derive the semantics from the de facto BEAM implementation[1]. But similarly to Java ERTS works by executing an intermediate representation of erlang source code, also known as BEAM code. As mentioned there's no specification for the generated instructions, though a few unofficial documents are available[2].

Quite unlike the Java programming language Erlang is a functional programming language based on the actor model, which dates back to 1973[3]. The actor model specifies the actor as the unit of concurrency. In erlang actors are called processes. Actors can only communicate with each other by sending messages, but are otherwise independent, which allows to run them in parallel with each other[4].

Because of this distinction, that the Erlang language focuses specifically on the actor

model, comparing the JVM with ERTS can be seen as comparing apples and carrots, because the ERTS is heavily geared and optimised for this specific view of the world, where the JVM has no such prejudices.

2. Memory Layout

2.1. JVM

Figure 1 shows the general architecture of the HotSpot VM when looked upon as memory areas[5].

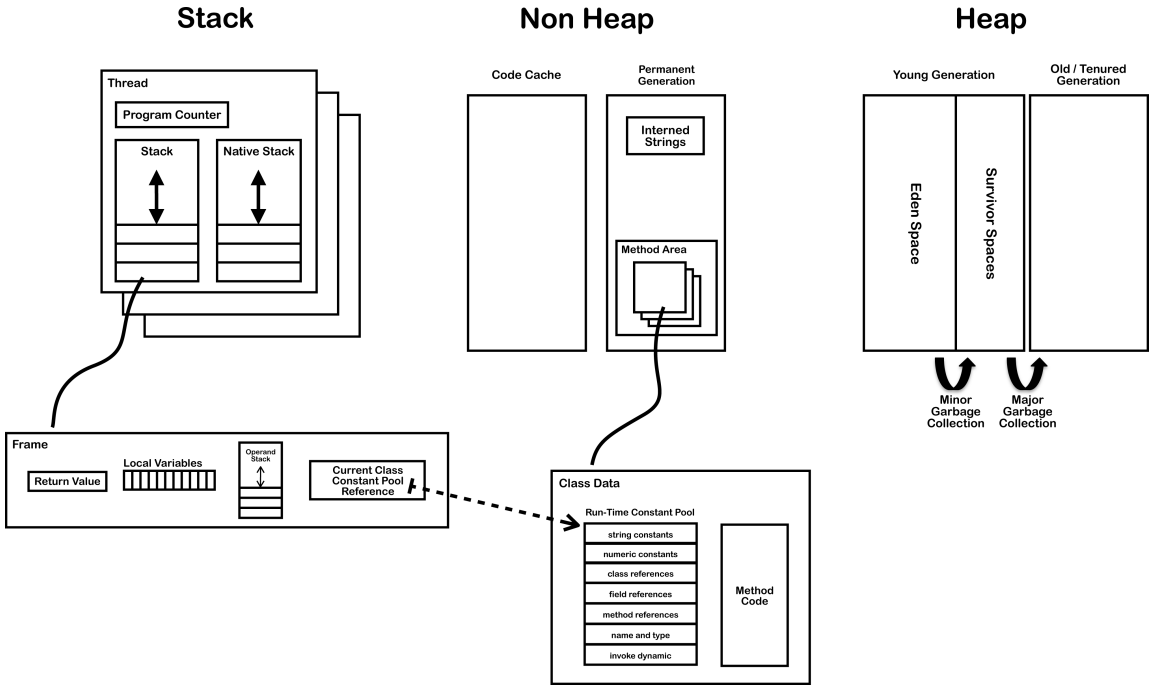

Figure 1: *Simplified JVM memory layout*

There's a shared heap, where all objects and arrays live and then there's a non heap area designated for metadata about the classes loaded and a Code Cache, which is used for compilation and storage of methods that have been compiled to native code by the JIT compiler.

Every thread in the JVM has a program counter, which holds the address of the current instruction (if it's not native), a stack, which holds frames for each executing method and a native stack. Every thread in JVM is mapped one-to-one to an operating system (OS) level thread, which is then scheduled and managed by the OS.

Heap on the HotSpot VM consists of a three distinct areas (also known as generations) called Eden, two smaller Survivor and Tenured spaces. This optimisation stems from the observation that most objects die young, thus it makes sense to put them into a separate area and garbage collect that smaller area more often than the whole heap[6].

The JVM has many different Garbage Collection (GC) algorithms embedded, which cater to different needs of the application. In general to collect garbage from the shared heap a stop-the-world pause is needed, where all application threads are halted. Different GC strategies optimise either for throughput - total GC overhead should be low, but long pauses are allowed - or latency - each stop-the-world pause should be as short as possible. Based on different heuristics (like what kind of machine

it's running on) the JVM will choose a strategy, but users can also enforce some strategy and give it some latency goals for example[7].

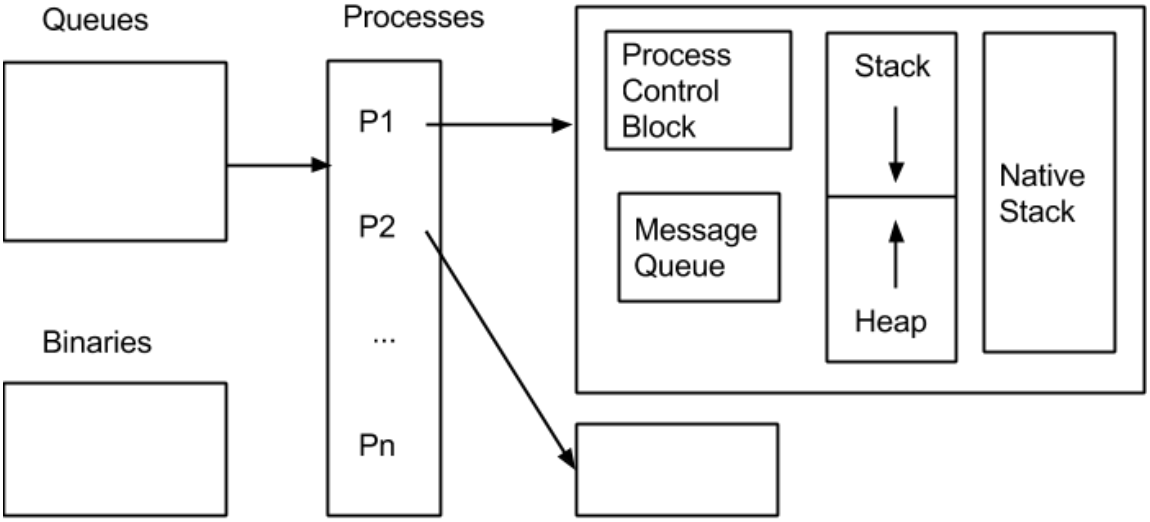

Figure 2: *Simplified ERTS memory layout*

2.2. ERTS

Figure 2 shows the general architecture of the Erlang Runtime System when looked upon as memory areas[8]. The major difference here is that instead of having a single shared heap, each process in Erlang has it's own. The same memory area is also used for the stack of the process, where the two are growing towards each other. Such a setup makes it very cheap to check wether a process is running out of heap or stack space.

Besides a heap and a stack each process has a process control block, which is used to hold various metadata about the process and a message queue. Binary data that is larger than 64 bytes is kept in a separate area, so that different processes could simply refer to it by a pointer.

Similarly to the JVM ERTS uses a generational garbage collection, meaning the heap is divided into a young and old generation. The GC strategy used is a copying collector (except for the binary area, which is garbage collected by reference counting), meaning it will allocate another memory area and then copy over each element that is still referenced, starting with the root set[8].

The per process heap architecture has many benefits, one of which is that because each process has it's own heap and there are presumably numerous process's, each heaps' root set can be very small and can be garbage collected quickly in isolation, without affecting other running processes. Meaning there aren't any global stop-the-world pauses.

When a process dies all it's memory can be deallocated right away, which means processes that are short lived or don't generate much garbage, may never go through a single GC cycle. The above (+ true preemptive scheduling, that is discussed later) gives Erlang the marketed soft real-time capabilities[9], where there aren't global pauses in the application, which often happen on the JVM.

3. Parallelism and Concurrency

Now for the tricky bit. Erlang is marketed as a tool for building massively scalable soft real-time systems with requirements on high availability[9]. It's often used in 99.999% up-

time guaranteed systems[10]. Java on the other hand is a more general-purpose programming language, used in fields such as high frequency trading to embedded devices.

3.1. Shared state

In 1971 Edsger Dijkstra posed and solved the now famous the Dining Philosophers Problem about concurrency. His solution involved a construct commonly known as a semaphore, which restrict access to a shared resource[11]. Dijkstra's solution is often referred to as shared state concurrency, which is the programming model Java uses.

The JVM has a single heap and sharing state between concurrent parties is done via locks that guarantee mutual exclusion to a specific memory area. This means that the burden of guaranteeing correct access to shared variables lies on the programmer, who must guard the critical sections by locks. However this is notoriously difficult to do, leading to widely spread belief that concurrency is hard[12].

There are numerous reasons why shared state between threads brings troubles:

1. Deadlocks - occurs when two threads both need resources A and B. First thread locks A first and then tries to get B, whereas the second thread operates in the opposite order. This results in a situation where both threads have locked one resource and wait for the other thread to release the other resource.
2. Race conditions - occurs when the programmer hasn't correctly guaranteed the order of execution between two threads. Which means that the correctness of the program depends on the OS scheduler, which is not deterministic, resulting in hard to reproduce errors.
3. Spending too much time in the critical section - occurs when a thread after acquiring a shared resources holds it for too long, effectively making all other interested parties wait and do nothing during that time.

As mentioned in the introduction, Erlang uses the actor model, where the only means to share state between concurrent and/or parallel actors is message passing. Furthermore, because each process has its own heap, there isn't any shared state by design (except for the binary and a few other Erlang specific memory areas). Removing shared state and relying only on message passing has the potential to make concurrency easy again[13].

> "Erlang is to locks what Java is to pointers[11]"

Arguably message passing concurrency model is on a higher abstraction level than using locks. Though because it's a higher abstraction it also removes some of the hardships of using locks and is easier to use, because we don't have to guarantee the correctness of the program in terms of concurrency anymore, the runtime does that for us.

Similarly the message passing style of concurrency can be successfully modeled with locks on the JVM. The Akka framework is essentially doing just this and brings the actor model to the JVM[14]. Using the Akka framework on the JVM makes the experience very similar to Erlang with the added benefit of ability to use the much larger Java ecosystem.

3.2. Message Passing

Because of the heap per process architecture in ERTS, message passing is done by coping the message from one process heap to another. This can be somewhat surprising as it would be much cheaper to simply pass a pointer to the message from one process to another, but it would require some shared memory area for the processes.

ERTS has actually experimented with both a shared and a hybrid heap architectures in the past[10], but after multithreading support was added to the BEAM the implementations were broken and later removed from ERTS[15]. A hybrid heap architecture is one where each process still has a separate heap, but messages that are to be sent to other processes are allocated

in a shared memory area to allow fast message passing. Experimentation showed that such an approach has promise, but as stated, the implementation was left unfinished[15].

The original reason for going with the copying strategy was that the destination process might be on another machine. If we are passing messages as pointers among processes on a single machine, but doing something else for processes on different machines, means that the error handling code will be more difficult and has to be handled separately. The underlying reason was to make the semantics of local and remote failures the same[16].

Furthermore, the "free lunch" for performance is long over, meaning processors aren't getting faster, instead multicore processors are standard[17]. Providing a coherent picture of memory to all cores is increasingly expensive and one can think of your cpu as a distributed system[18]. Whether it's hidden from the programmer or not, the cpu will have to do some copying when a cache miss occurs for example. When data sharing times between cores continue to rise it makes sense to do the expensive operation of copying immediately[16].

3.3. Lightweight threads

As mentioned before, the JVM maps its threads one-to-one to OS level threads, which are then scheduled by the OS scheduler, but in Erlang a process is simply a separate memory area and are mapped n-to-m to OS level threads. Since 2006 ERTS supports true multithreading through Symmetric Multi Processing (SMP). With SMP ERTS starts with the same number of schedulers as there are cores[19].

Each scheduler has a run queue that contains runnable processes. A process runs until it tries to receive a message, but the mailbox was empty, or it runs out of reductions. The meaning of reductions in ERTS is not clearly defined, but they should represent "units of work" and are roughly equivalent to function calls. Each process that starts running initially has 2000 reductions, and when it rans out is put at the end of the run queue[20].

The effect is that Erlang is one of a few languages that actually does preemptive multitasking. The reduction count of 2000, which is sub 1ms, is quite low and forces many small context switches between the Erlang processes. But this also ensures that an Erlang system tends to degrade in a graceful manner when loaded with more work[21].

Schedulers will also balance work between each other. The strategy can be configured to either balancing load out as evenly as possible or using the least amount of schedulers (putting the other ones to sleep, conserving power for example)[20].

The important note here is that blocking a process doesn't block a scheduler, it will simply immediately start running the next process. As Erlang processes are not OS level threads they are more lightweight, which is the main reason ERTS can run hundreds of thousands of processes. Erlang processes use a dynamically allocated stack, that starts off much smaller than Java threads. By default an Erlang process will use around 2.5KB[22] on a 64bit machine, wheres as a Java thread starts off with a 1024KB stack on a 64bit machine.

There is a library for Java called Quasar, that tries to bring the same lightweight threads to the JVM as are Erlang processes. In order to really pull it off, Quasar needs to instrument your code to save the execution state and restore it when that lightweight thread starts running again[23]. Though instrumentation has many challenges that add complexity to the program instead of removing it.

The same applies for the JVM actor framework Akka mentioned earlier. They can work very well, if the programmers take care to follow some principles. As Akka actors run on OS Threads an actor that randomly blocks will block the entire thread. Without fundamental changes in how the JVM works, one cannot guarantee that an arbitrary piece of code will not block[24].

4. Runtime Optimisations

4.1. JIT compiler

When looking at the JVM and comparing it with other Virtual Machines then we cannot forget to mention the just-in-tim (JIT) compiler that HotSpot has, which has the largest effect on performance of the JVM[25]. As mentioned in the introduction, both Erlang and Java source code are not directly compiled to executable binaries or native code. Instead they rely on the runtime to execute the statements in the intermediate language (bytecode for java and so called BEAM code for Erlang).

Purely interpreting commands sequentially is slower simply because of having to translate commands over and over again into the machines native instructions, not to mention various optimisations that good compilers do[25]. JIT compiler in the HotSpot works by keeping track of what methods are "hot" (called often) and then optimising and compiling them to native code. This has the benefit that effort is not spent to compile/optimise methods that don't execute or do so rarely.

4.2. HIPE and BEAMJIT

ERTS comes with a ahead-of-time compiler called HiPE (High Performance Erlang). It is sometimes called also a just-in-time compiler, but in reality it's more of a ahead-of-time compiler, because the user has to choose which functions or modules are compiled into native code. It doesn't do it automatically during runtime for most used functions[26].

Compiling ahead of time also strips away many possibilities for optimisation that the HotSpot JIT compiler does. Essentially the HotSpot JIT compiler is able to gamble the system into better performance by doing shortcuts. For example the HotSpot JIT compiler assumes that a method never throws an exception or that most methods are not overloaded and links directly to a specific callsite instead of traversing the class hierarchy each time to find the most specific method[25].

These optimisations are possible on the JVM, because it can keep track of the shortcuts it has made and if any of them become invalid, for example a method does throw an exception, then the JIT compiler will deoptimize the method and compile it again without the optimisation or run it as interpreted. The HiPE compiler however cannot make such shortcuts, because it hasn't got the ability to change gears during runtime, the once compiled code has to be correct and work in all circumstances.

However a true just-in-time compiler for ERTS is in development and called BEAMJIT. Similarly to the HotSpot JIT compiler it uses tracing to decide which methods should be compiled to native code and which not. It has shown increased performance in some benchmarks, but the current implementation doesn't do any Erlang language specific optimisations, which leaves out the best part of having a true just-in-time compiler[27].

5. Summary

As we've seen the JVM and ERTS are quite different underneath. Namely the difference stems from the fact that Erlang takes the Actor model as it's basis and has built it's runtime around that. Whether that's good or bad is up to debate and most likely depends on the problem at hand. Java and the JVM provide enough tools to retrofit any concurrency model out there, but retrofitting anything won't be the same as taking it into the initial design. That said there's a wonderful quote from the one of the creators of Erlang:

> "Erlang accidentally has the right properties to exploit multi-core architectures - not by design but by accident" - Joe Armstrong[16]

References

[1] E. Stenman, "Vm tuning, know your engine - part ii: the beam." `https://www.youtube.com/watch?v=RcyN2yS5PRU`, July 2013.

[2] "Beam file format." https://synrc.com/publications/cat/Functional%20Languages/Erlang/BEAM.pdf, May 2012.

[3] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, (San Francisco, CA, USA), pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.

[4] F. Hebert, *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press Series, No Starch Press, 2013.

[5] J. Bloom, "JVM Internals." http://blog.jamesdbloom.com/JVMInternals.html, November 2013.

[6] B. Goetz, "Java theory and practice: Garbage collection in the HotSpot JVM." http://www.ibm.com/developerworks/library/j-jtp11253/, November 2003.

[7] "Memory Management in the Java HotSpot™ Virtual Machine," tech. rep., Sun Microsystems, 04 2006.

[8] E. Stenman, "Erlang engine tuning: Part 1 - know your engine." http://www.youtube.com/watch?v=QbzH0L_0pxI, April 2013.

[9] "http://www.erlang.org/." http://www.erlang.org/.

[10] E. Johansson, K. Sagonas, and J. Wilhelmsson, "Heap architectures for concurrent languages using message passing," in *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, (New York, NY, USA), pp. 88–99, ACM, 2002.

[11] S. Akhmechet, "Erlang style concurrency." http://www.defmacro.org/ramblings/concurrency.html, August 2006.

[12] D. V. Subramaniam, "What makes concurrency hard," *Healthy Code*, July 2014.

[13] J. Armstrong, "Concurrency is easy." http://armstrongonsoftware.blogspot.com/2006/08/concurrency-is-easy.html, August 2006.

[14] "http://akka.io/." http://akka.io/.

[15] R. Carlsson, "[erlang-questions] why are messages between processes copied?." http://erlang.org/pipermail/erlang-questions/2012-February/064613.html, February 2012.

[16] J. Armstrong, "[erlang-questions] why are messages between processes copied?." http://erlang.org/pipermail/erlang-questions/2012-February/064617.html, February 2012.

[17] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software." http://www.gotw.ca/publications/concurrency-ddj.htm, March 2005.

[18] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs, "Your computer is already a distributed system. why isn't your os?." https://www.usenix.org/legacy/event/hotos09/tech/full_papers/baumann/baumann_html/index.html, March 2009.

[19] K. Lundin, "Inside the erlang vm." http://www.erlang.org/euc/08/euc_smp.pdf, November 2008.

[20] L. Larsson, "Understanding the erlang scheduler." https://vimeo.com/86106434, February 2014.

[21] J. L. Andersen, "How erlang does scheduling." http://jlouisramblings.blogspot.com/2013/01/how-erlang-does-scheduling.html, January 2013.

[22] "Efficiency guide 8.1 creation of an erlang process." http://www.erlang.org/doc/efficiency_guide/processes.html.

[23] R. Pressler, "Jvmls 2014: Lightweight threads." http://medianetwork.oracle.com/video/player/3731008736001, August 2014.

[24] C. Moon, "Actors, green threads and csp on the jvm – no, you can't have a pony." http://www.boundary.com/blog/2014/09/no-you-cant-have-a-pony/, March 2014.

[25] S. Oaks, *Java Performance: The Definitive Guide*. O'Reilly Media, 2014.

[26] K. F. 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 Workshop* (B. Däcker and T. Arts, eds.), pp. 36–42, ACM, 2003.

[27] F. Drejhammar and L. Rasmusson, "Beamjit: A just-in-time compiling runtime for erlang," in *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, (New York, NY, USA), pp. 61–72, ACM, 2014.