RELEASE

SEVENTH FRAMEWORK PROGRAMME

ICT-287510
RELEASE
A High-Level Paradigm for Reliable Large-Scale Server Software
A Specific Targeted Research Project (STReP)

# D2.2 (WP2): Prototype Scalable Erlang VM Release

Due date of deliverable: March 31, 2013
Actual submission date: April 8, 2013

Start date of project: 1st October 2011 Duration: 36 months

Lead contractor: Uppsala University Revision: 0.1 (April 8, 2013)

**Purpose:** To describe the implementation of key components of the prototype scalable Erlang Virtual Machine release: the implementation of its scalable Erlang Term Storage, its efficient tracing support for profiling and monitoring, and the preliminary port of Erlang VM on a Blue Gene/Q.

**Results:** The main results presented in this deliverable are

- A scalable implementation of the Erlang Term Storage (ETS) mechanism that is already included in Erlang/OTP R16B (released in late February 2013) and the description of alternative designs, which currently exist as prototypes, that will increase the scalability of ETS further.
- An efficient tracing support based on DTrace/SystemTap that Erlang/OTP offers since release R15B01 (April 2012) for profiling and monitoring Erlang applications.
- A preliminary port of the Erlang VM on the Blue Gene/Q and the description of its status.

**Conclusion:** The two key components of the scalable Erlang VM release that we describe in this document, scalable ETS and efficient tracing support, are already part of Erlang/OTP and used by the Erlang community. The Blue Gene/Q port of Erlang/OTP is complete and fully functional on the front end nodes, but significant work still remains for the Erlang VM to execute efficiently on the compute nodes of the machine.

| Project funded under the European Community Framework 7 Programme (2011-14) | | | |
|---|---|---|---|
| **Dissemination Level** | | | |
| PU | Public | | ✳ |
| PP | Restricted to other programme participants | (including the Commission Services) | |
| RE | Restricted to a group specified by the consortium | (including the Commission Services) | |
| CO | Confidential only for members of the consortium | (including the Commission Services) | |

# Prototype Scalable Erlang VM Release

Konstantinos Sagonas <kostis@it.uu.se>
David Klaftenegger <david.klaftenegger@it.uu.se>
Patrik Nyblom <pan@erlang.org>
Nikolaos Papaspyrou <nickie@softlab.ntua.gr>
Katerina Roukounaki <arou@softlab.ntua.gr>
Kjell Winblad <kjell.winblad@it.uu.se>

## Contents

**1 Executive Summary** **2**

**2 Introduction** **2**

**3 Scalable Erlang Term Storage** **3**
3.1 Global Data Structures and Locking in ETS . . . . . 4
3.2 Scalability Problems of the Original ETS Implementation . . . . . 5
3.3 Scaling ETS Using Fine Grained Locking . . . . . 5
3.4 Scaling the ETS Implementation Further . . . . . 7
3.5 Current Status and Future Work . . . . . 8

**4 Efficient Tracing Support for Profiling and Monitoring** **8**
4.1 DTrace/SystemTap Support in Erlang/OTP . . . . . 9
4.2 A Brief Introduction to DTrace . . . . . 9
4.3 Build Support for Dynamic Tracing . . . . . 10
4.4 Virtual Machine Probes . . . . . 12
4.5 Support for Probes in Erlang Code . . . . . 13
4.6 Support for DTrace Tags in Erlang Code and in Probes . . . . . 14
4.7 Profiling in Erlang using DTrace . . . . . 15
4.8 Future Work . . . . . 25

**5 Preliminary Port of Erlang/OTP to Blue Gene/Q** **25**
5.1 Blue Gene/Q Architecture . . . . . 25
5.2 Porting Challenges . . . . . 26
5.3 Current Port Status . . . . . 27
5.4 Future Work . . . . . 28

**6 Concluding Remarks** **28**

**A The `erl-xcomp-powerpc64-bgq-linux.conf` Configuration File** **30**

# 1 Executive Summary

This document presents the second deliverable of Work Package 2 (WP2) of the RELEASE project. WP2 is concerned with improving the scalability of the Erlang Virtual Machine (VM). Towards this goal we have made a fully working prototype release of key components of a scalable Erlang VM (these components are included in Erlang/OTP R16B, released on the 25th of February 2013), and in this document we describe them. More specifically, in this report:

- We review the scalability bottlenecks that we have identified in the implementation of the Erlang Term Storage (ETS) and changes to the Erlang VM that have been performed since the start of the RELEASE project in order to improve the scalability of ETS. (These changes are part of Erlang/OTP R16B.) We also discuss some additional changes to ETS design and propose the use for lock-free data structures for ETS Tables that, on manycore architectures, will increase the scalability of ETS further. (These changes currently exist only as prototypes.)
- We present in detail the efficient tracing support that the Erlang/OTP system nowadays includes for profiling and monitoring Erlang applications. The tracing support is based on DTrace/SystemTap and we describe its configuration procedure, the VM probes that it comes with, how additional probes can be added in Erlang code, the support for DTrace tags in Erlang code and in probes, and how the dynamic tracing offered by DTrace can be used to profile and monitor Erlang applications.
- Finally, we report on the status of a preliminary port of the Erlang VM on the Blue Gene/Q of EDF and the work that remains for Erlang/OTP to run efficiently on the compute nodes of the machine.

# 2 Introduction

The main goal of the RELEASE project is to investigate extensions of the Erlang language and improve aspects of its implementation technology in order to increase the performance of Erlang applications and allow them to achieve better scalability when run on big clusters of multicore machines. Work Package 2 (WP2) of RELEASE aims to improve the scalability of the Erlang VM. The lead site of WP2 is Uppsala University. The objectives of the tasks of WP2 pertaining to this deliverable are:

**Task 2.2:** "*... investigate alternative implementations of the Erlang Term Storage mechanism ...*"

**Task 2.4:** "*... design and implement lightweight infrastructure for profiling applications while these applications are running and for maintaining performance information about them.*"

**Task 2.5:** "*... port the Erlang/OTP system to a massively parallel supercomputer, a Blue Gene/P machine available at EDF ...*"

Towards achieving the objectives of these tasks, this deliverable (D2.2), due exactly in the middle of the duration of the project, accompanies the release of a prototype scalable Erlang VM and describes some of its key components. The last deliverable of WP2 (D2.4) will concern the release of a scalable Erlang VM in which these components will be robust and more efficient than those of the current prototype. In between these two points, deliverable (D2.3) will present a prototype scalable runtime system architecture.

On the 25th of February 2013, Erlang/OTP R16B was released containing many scalability and performance improvements to the Erlang VM done during the duration of the RELEASE project. In particular, compared to releases of Erlang/OTP before the start of the project, among other changes and improvements to Erlang's VM, it contains:

- various scalability improvements in the implementation of ETS (Erlang Term Storage);

- efficient tracing support, based on DTrace/SystemTap, for (offline or online) profiling and monitoring of Erlang applications; and
- supporting infrastructure for a preliminary port of Erlang/OTP on the Blue Gene/Q architecture.

This report describes these additions and improvements in detail.

The work for this deliverable has been done by researchers from Ericsson AB (EAB), the Institute of Communication and Computer Systems (ICCS), and Uppsala University (UU). The breakdown was roughly the following:

- the scalability improvements to ETS contained in Erlang/OTP R16B were performed by the EAB team while the UU team investigated the scalability of ETS and implemented prototypes with the changes and additional scalability improvements that are presented in Section 3.4;
- the tracing support was implemented, documented and included in Erlang/OTP by the EAB team and researchers of ICSS implemented and documented the DTrace/SystemTap Erlang probes that are presented in Section 4.7; and
- the preliminary port of Erlang/OTP on the Blue Gene/Q has been done by the UU team.

Note that, compared with the phrasing of Task 2.5 in the description of work of RELEASE, the preliminary port to the Blue Gene targets a more modern version of the Blue Gene family of machines, namely a Blue Gene/Q that EDF acquired since the beginning of the project to keep company to their older Blue Gene/P machine. Access to this newer machine was only made possible to RELEASE partners other than EDF at the end of January 2013, which is partly responsible for the immaturity of the preliminary port of Erlang/OTP.

The rest of this document consist of three sections that describe the current implementation of the scalable Erlang Term Storage and some designs for further scalability improvements (Section 3), the efficient tracing support that Erlang/OTP contains for profiling and monitoring (Section 4), and the preliminary port of Erlang/OTP on the Blue Gene/Q, its current status and the porting tasks that remain (Section 5). The report ends with a brief section with some concluding remarks.

# 3 Scalable Erlang Term Storage

The Erlang Term Storage (ETS) [8] is a key feature of the Erlang/OTP system. It supports storage of Erlang tuples outside the heaps of their creating process. More importantly, ETS is special for Erlang as it provides a mechanism for sharing data between processes. Furthermore, in contrast to terms stored in process-local heaps or used in messages sent between processes, this data can be mutated. The implementation of ETS uses dedicated data structures called *ETS tables* to store tuples where one of the positions is designated as their key. In essence, ETS tables are mutable key-value dictionaries. ETS provides different table types with different characteristics (`public`, `protected`, `private`, `bag`, `duplicate_bag`, `set` and `ordered_set`) but they all share a common interface.

Due to properties such as the ones described above, ETS is a key ingredient of many Erlang applications. It is used either indirectly, as it is the basis of the implementation of `mnesia` [11], the main memory database of Erlang/OTP, or directly by the code of Erlang applications. In many concurrent Erlang programs, the best way to communicate data between processes is to use messages. In other programs, ETS is heavily used as a convenient and presumably efficient way to achieve data sharing. On the other hand, it is well known that data that needs to be accessed and modified concurrently by several processes is a common scalability bottleneck for parallel programming. Thus, a scalable Erlang Virtual Machine needs to provide an ETS implementation whose performance does not significantly deteriorate as the number of processes that require access to the ETS tables increases.

On the semantics level, an ETS table behaves as if a dedicated stateful Erlang process served requests for insertions, lookups, and other operations on a mutable key-value dictionary. However, on the implementation level, ETS is implemented in C in a more efficient way than what could have been accomplished with a pure Erlang implementation, as Erlang does not have efficient support for working with mutable data.

Erlang tuples that are inserted in an ETS table are copied from the heap of the process that is inserting the data to the table. Erlang tuples that are retrieved from an ETS table are also copied from the memory of the table to the heap of the retrieving process.[1] The reason why the whole tuple is copied and a reference is not just passed to the tuple is the same as the reason why messages are copied between processes during message passing: to allow for efficient process-local garbage collection and cheap (i.e., constant time) deallocation when a process dies.

ETS provides the most common table operations (`insert`, `lookup`, and `delete`) as well as operations for searching for entries matching a specific criterion and for traversing all entries in a table.

In the rest of this section, after briefly reviewing the global data structures and the kinds of locking that ETS operations currently use (Section 3.1), we describe scalability problems that we have found while benchmarking and carefully reviewing the original implementation of ETS (Section 3.2) and present solutions to some of the problems that by now are fully implemented and included in Erlang/OTP R16B (Section 3.3). Finally, we present promising alternative implementations of ETS that we are currently implementing, evaluating, and fine-tuning (Section 3.4).

### 3.1 Global Data Structures and Locking in ETS

The following data structures are maintained on a node-wide level and are used for generic book keeping by the Erlang VM. Low-level operations, like finding the main data structure for a particular table or handling transfers of ownership, use only these data structures.

**meta_main_table** Contains pointers to the main data structure of each table that exists in the VM at any point during runtime. Table identifiers (TIDs) map to indices in this table. Each element in the `meta_main_table` has a corresponding readers-writer lock. These locks are stored in an array called `meta_main_tab_locks` containing 256 elements. Additionally the `meta_main_table` has a write lock which is used to prevent several threads from modifying the lock table itself at the same time.

**meta_name_table** Contains mappings from the names of named tables to the respective TIDs.

**meta_pid_to_tab** Maps processes (PIDs) to the tables they own. This data structure is used when a process exits to handle transfers of table ownership or table deletion.

**meta_pid_to_fixed_tab** Maps processes (PIDs) to tables that are fixated by them.

Different levels of locking are required for different operations on an ETS table. The same lock data structure is accessed before both read and write operations on the data protected by the lock; operations simply request and obtain different levels of access. For example, acquisition of the lock to read the data should not block other read operations but should block all the write operations until the lock is released. Similarly, acquisition of the lock to write on the data should block both read and write operations on the same data.

Operations may also lock different sets of resources associated with a particular operation on an ETS table:

- Creation and deletion of a table require the acquisition of the write lock protecting the `meta_main_table` as well as the corresponding lock in the `meta_main_tab_locks` array.

[1] Big binaries and bitstrings are an exception, in the same way that they are not copied when contained in messages sent between processes.

- Creation, deletion and renaming of a named table also require the acquisition of the write lock protecting the `meta_name_table` and the corresponding lock in the `meta_main_tab_locks` array.

- Read and write operations on a table's entries require the acquisition of appropriate table locks as well as acquisition of the corresponding read lock in the `meta_main_tab_locks` array. Using the default options, each table has just one main lock, used for all entries. Depending on the type and the options specified when a table is created, read and/or write operations for different keys can be performed simultaneously, by locking only a part of the table.

### 3.2 Scalability Problems of the Original ETS Implementation

Contended locks that are held for even a short time can cause scalability problems on multicore computers. In early Erlang/OTP releases, all ETS tables had a global readers-writer lock. In Erlang programs that many processes access ETS tables in parallel, we suspected that this global table lock was a potential scalability problem.

To investigate whether this was the case, we designed and wrote benchmarks where many Erlang processes do ETS write (`ets:insert/2`) operations in parallel. In these benchmarks, we have been able to observe a significant slowdown in the number of operations performed per time unit when using many processes. Ideally, we would like to see a speedup in the number of operations performed per time unit when many processes are doing operations on a table as long as the operations are not modifying exactly the same element in the table. Clearly, this is not possible to achieve with a single global lock per table. In the following section we describe what has been done in Erlang/OTP R16B and prior to improve the scalability of ETS.

### 3.3 Scaling ETS Using Fine Grained Locking

One obvious way to improve the number of simultaneous concurrent accesses to an ETS table is to use fine grained locking, instead of having just one lock per table. In addition, this number can be increased by using locks that are specialized for the type of access (i.e., read vs. write) that is desired. For this reason, ETS nowadays comes with two options, called `read_concurrency` and `write_concurrency`, that can be used to fine tune the performance of ETS tables when many processes are accessing them in parallel. Detailed information about how these options can be used is given in the ETS user manual [8], but we present a brief summary below.

- The `read_concurrency` option was introduced in Erlang/OTP R14B. When `read_concurrency` is enabled the locks used for ETS are optimized for frequent reads. Such a lock has several flags in its memory for readers. For example when an Erlang process running in scheduler $A$ is taking the lock for reading it might write to reader flag 1 while another process running on scheduler $B$ might write to reader flag 2. This can improve performance for reading compared to having a single reader flag because of how the memory cache system is constructed in modern multicore computers. When many cores write to the same memory location, that memory location (cache line) needs to be transferred between the cores, which can be expensive.

- The `write_concurrency` option was introduced in Erlang/OTP R13B02-1. When enabled, fine grained locking will be used for the table types based on hashing (`bag`, `duplicate_bag`, and `set`). Initially, the buckets in a hash table were divided between 16 locks. The number of locks that the buckets are divided between was increased in R16B from 16 to 64 since our benchmarks showed that 16 locks are not enough to provide good scalability on computers with many cores.

The results of a benchmark created to understand how the number of locks affect the scalability of an ETS `set` table can be seen in Figure 1. The benchmark was run with the standard Erlang/OTP release R15B02 which uses 16 bucket locks and with code modifications that use 32, 64, 128, and 256 bucket locks. The Erlang VM was started with schedulers pinned to OS threads and with 64 reader

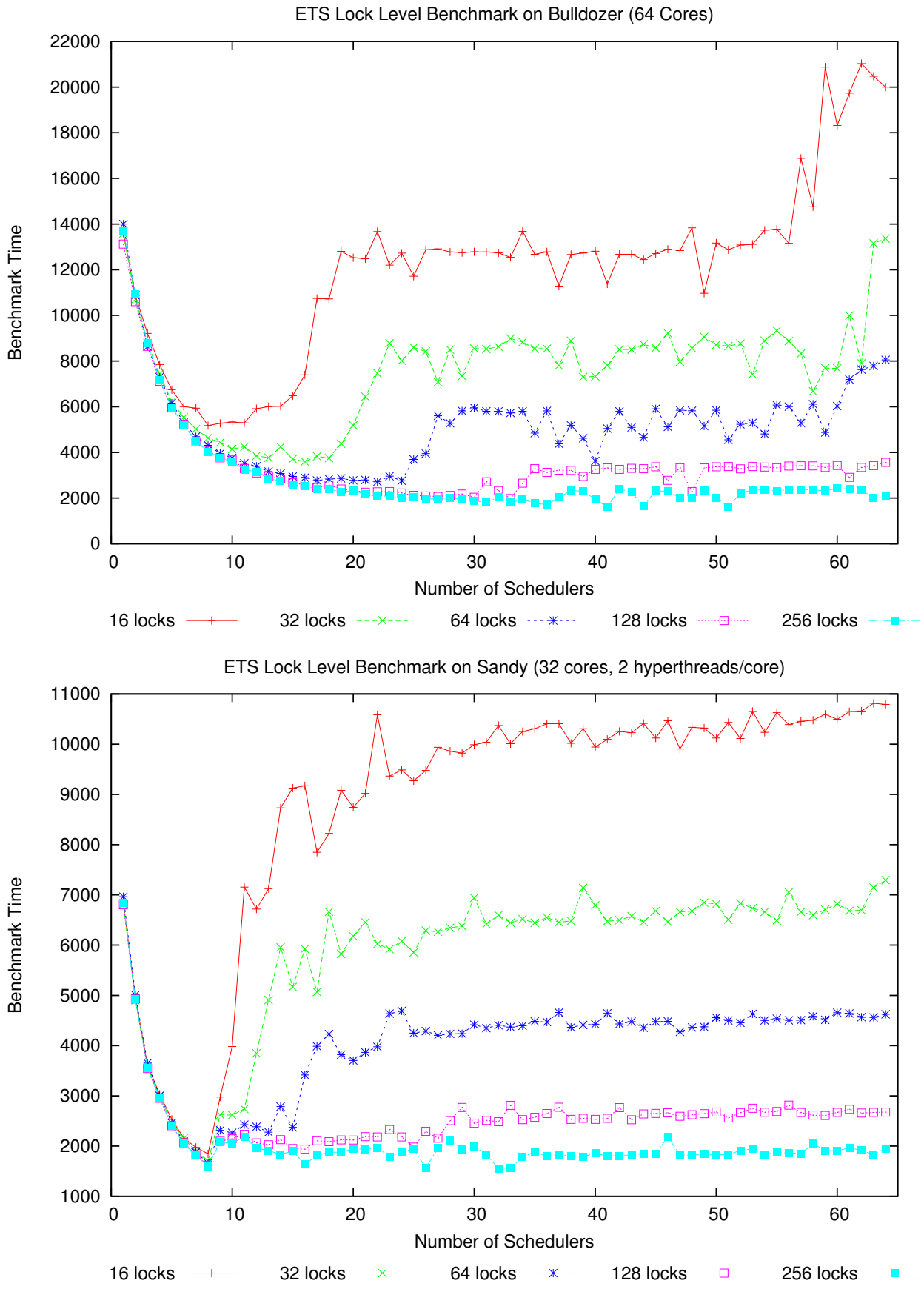


Figure 1: Performance of ETS varying the number of bucket locks on two different architectures.

groups (i.e., `ERL_ARGS="PIN_RG=+sbt tnnps +rg 64"`). We run the benchmark on two different platforms, corresponding to the two figures:

**"Bulldozer":** a machine with four AMD Opteron(TM) Processor 6276 CPUs (2.30 GHz) and 128GB of RAM running Linux 3.2.0-4-amd64 SMP Debian 3.2.35-2 x86_64 (a total of 64 cores).

**"Sandy":** a machine with four Intel(R) Xeon(R) CPU E5-4650 CPUs (2.70GHz), 8 cores each, and also 128GB of RAM running Linux 3.2.0-4-amd64 SMP Debian 3.2.35-2 x86_64 (a total of 32 cores, each with hyperthreading).

Based on these results, we expect that the number of locks may increase further (e.g. to 128 or 256) in a future Erlang/OTP release. A detailed description of the benchmark as well as source code can be found at the following location: https://github.com/kjellwinblad/ets_impl_project.

### 3.4 Scaling the ETS Implementation Further

In this section we describe ongoing and future work to make ETS even more scalable.

**Remove Locks From ETS Meta Table** The ETS meta table is an internal data structure that is used in ETS to map table identifiers to table data memory addresses. The elements in the meta table are protected by readers-writer locks that we call *meta table locks*. The writer part of a meta table lock is only taken when tables are created or deleted. However, a meta table lock is acquired for reading every time a process executes an ETS operation. The meta table locks might be a scalability problem since, if many processes access a single table frequently, there can be contention on memory bandwidth due to the write accesses to take the read lock. In a prototype implementation of the scalable Erlang VM, we have completely removed the need for the meta table locks. In the same prototype the meta table is modified and read by *atomic* operations.

This approach leaves only ETS table deletion as a problem, as the read-locked meta table entry protected unlocked tables from deletions. To solve this issue, we have added an ETS pointer to every scheduler's local data, which points to the ETS table that is currently being accessed. Before a table is deallocated it is first marked as dead in the meta table and then the thread blocks until no ETS pointer is pointing to the ETS table.

**Global Table Lock** Without the meta table lock, the contention shifts to the global lock on every ETS table. An alternative locking scheme was tested in another prototype. In the alternative locking scheme the read lock is acquired by setting the ETS pointer for the scheduler that is described in the previous paragraph. This approach to read locking is similar to the lock optimized for frequent reads described in Section 3.3 but with the advantage that it uses less memory.

This alternative locking scheme is not ready for production yet as these changes make it possible for threads that just acquire the table lock for reading to be starved by threads that acquire the table lock for writing.

Solving this issue is part of ongoing research. Example directions are to allow upgrading from read locks to write locks in order to avoid starvation or to replace the entire locking scheme for the global table lock with a more suitable approach.

For ETS data structures that use fine grained locking or some lock-free technique, most operations only need to acquire the global table lock for reading. We have therefore decided to optimize the lock for frequent reads to allow new lock-free or fine grained locking table data structures to perform well.

**Lock-free Data Structures** Another opportunity to increase scalability of ETS is to use more scalable data structures as the underlying implementation of the ETS tables. Currently, the choices are limited to a linear hashing for tables of type `bag`, `duplicate_bag` and `set` and an AVL tree for tables of type `ordered_set`. While any kind of hash table can be subdivided into an arbitrary

number of parts to allow more parallelism, this is significantly harder for a tree-like structure. Further research could make use of either lock-free or otherwise concurrent data structures to allow faster concurrent access to ETS tables. The changes to remove the locks from the ETS meta table and finding a solution to the issues with the global table lock, which we described in the previous two paragraphs, are however a prerequisite for making such schemes useful. Otherwise, on many-core NUMA machines the locking performance could dominate the overall access performance to the data structure, making proper measurements and conclusions about the advantages and disadvantages of various data structures impossible.

**Improvements of Current Data Structures** The current `ordered_set` table type, which is implemented by AVL tree, uses only one lock for the whole data structure. We would like to replace this implementation with a lock-free data structure or a data structure that uses fine grained locking.

The table types `set`, `bag` and `duplicate_bag`, which are implemented by linear hashing, can use fine grained locking to increase parallelism of write operations. We have, however, identified some problems that might limit scalability even when fine grained locking is enabled. For example, only one insert (or delete) operation can perform table re-sizing at the same time which might degrade performance if many processes are doing inserts for longer time. Another problem could be that the field containing the size of the table might be a bottleneck if many inserts and deletes are performed in parallel. We will investigate whether these potential performance bottlenecks can be lifted.

### 3.5 Current Status and Future Work

Since the start of the RELEASE project, the scalability of the ETS implementation in Erlang/OTP has been improved via the introduction of user-controllable table options that support fine grained locking or locking optimized for frequent reads. Some remaining scalability problems in ETS, like the single global lock for tables of the `ordered_set` type, have been identified and we have developed a suitable set of benchmarks to test scalability of different use case scenarios. Last but not least, we have proposed improvements to the existing implementation and we are currently investigating the suitability of lock-free data structures as an alternative to the existing implementation. Although our prototype implementations are robust and promising, much work remains in order to address all issues that are prerequisites for their proper inclusion into a mature and complex system such as Erlang/OTP and for fine-tuning their performance.

## 4 Efficient Tracing Support for Profiling and Monitoring

The Erlang/OTP runtime system has built-in support for profiling and tracing Erlang applications, and collects information about several types of events that occur during their execution (e.g., process creations, message receptions, function calls, etc.).

In short, the built-in tracing mechanism works as follows: for each traced event that takes place in a traced process, a trace message is constructed and sent to the appropriate tracer (i.e., a local process or port). Erlang programmers are equipped with a number of built-in functions (BIFs) that allow them to enable or disable profiling and tracing, as well as to specify *what* they want to profile or trace and *how* they want to perform these tracing actions.

At the start of the RELEASE project, Erlang/OTP already contained several tracing and profiling tools, all of which were based on Erlang's built-in tracing infrastructure: `cprof` [3], `dbg` [4], `eprof` [5], `et` [6], `etop` [7], `fprof` [9], `lcnt` [10], `percept` [12], `pman` [13], and `ttb` [14] are some of these tools.

Although this existing profiling tracing infrastructure is widely used, it has several drawbacks, the most important of which being the overhead that it imposes on the traced program, and the fact that there can only exist at most one tracer for each tracee (which explains why, for example, the `cprof` and `lcnt` tools cannot run at the same time for the same application).

In an attempt to limit the impact that event tracing has on the traced application and to overcome the "one-tracer-per-tracee" limitation, we designed and implemented an alternative tracing

mechanism, and included it into the Erlang/OTP runtime system. For this purpose, we used the dynamic tracing framework DTrace [2].

### 4.1 DTrace/SystemTap Support in Erlang/OTP

The support for DTrace in the Erlang Virtual Machine (VM) and rudimentary support for writing DTrace/SystemTap probes in Erlang code has been in Erlang/OTP since the release of R15B. The support was originally an open source contribution which provided the basics for the current implementation. Some of the functionality was present in the original contribution by Scott Lystig Fritchie [1], while the rest was implemented by the Ericsson team participating in the RELEASE project.

The DTrace and SystemTap frameworks for dynamic tracing are very similar. SystemTap is more or less a clone of the original DTrace support present in Oracle Solaris and Apple MacOS X. The general view is that SystemTap is "DTrace for Linux". Therefore this document will only differentiate between the two frameworks when necessary. In the rest of this section, the framework name DTrace will be used to describe both the DTrace and the SystemTap support. When differentiation is needed, the differences will be made clear.

The DTrace support in Erlang/OTP can be divided into four areas:

1. Build support.
2. Virtual machine probes.
3. Support for probes in Erlang code.
4. Support for DTrace tags in Erlang code and in probes.

All of these areas require changes to the code of Erlang's VM and everything had to be implemented without any performance loss in the normal build of the VM. The goal was ultimately to have no noticeable performance loss even in builds where DTrace support was enabled. That goal was reached, at least on MacOS X and according to the benchmarks available.

### 4.2 A Brief Introduction to DTrace

DTrace is a dynamic tracing framework which allows one to add probes into a binary executable in such a way that there is not any (or at least not any noticeable) loss of performance as long as the probes are not activated. The probes can be present in a production system at no cost and, when problems arise, the probes can be activated and one can get information about the inner workings of the running program. In MacOS X and Solaris, probes are present throughout the system and one can follow the effects of a program through the kernel API's and therefore get very detailed information about what resources the program consumes under the current conditions. In Linux, the SystemTap support has traditionally been optional, but the basic kernel support is in recent kernels (3.6 and upwards) part of the main branch. In such kernels, which are common in recent Linux distributions, only minimal configuration is needed to benefit from Erlang's SystemTap instrumentation.

To activate a probe in a running program, one loads a `.d` (or `.stp`) script into the kernel, which activates the relevant probes in the executable. When a probe is activated, whenever the program's thread of execution passes the activated probe, information is sent to the kernel, which interprets the `.d` script to generate output to the tracing user. The `.d` script also has the option to drop the message, use the message to update accumulators or in other rudimentary ways manipulate the data sent to it from the probe. The messages contain information enough to identify the probe itself, optionally together with further information that is relevant in the circumstances.

While DTrace can be used to do very fine granular tracing of function entries, returns and system calls, the most useful feature from our point of view is the ability to define custom probes and insert them at relevant points in the program. To define a custom probe, one simply writes a `.d` script defining the probe. As an example, let us define a simple probe for an example program:

```
provider example {
  probe output(int counter);
};
```

Given that we have generated a header from the above `.d` script, in the actual C code we can then simply insert the probe using a macro:

```
EXAMPLE_OUTPUT(i);
```

The parameter `i` is simply a variable in the program. When the program containing the probe is running, we can activate another `.d` script that will enable the probe and will generate some output whenever the probe is hit in the running program:

```
example*:::output {
  printf("Example program output number %d\n", arg0);
}
```

In principle, regardless of dynamic trace implementation, one is able to insert probes in a program that are more or less cost-less until some `.d` script activates them. The activating `.d` script can also be tailored to generate arbitrary output or calculations on the data it receives from the probe, making DTrace a very flexible tool.

Along with the simple probe macros, there are macros for telling from inside the program if a probe is enabled or not, which is useful if some computation can be avoided when no one has activated the probe. Let us for example imagine that the counter value needs to be fetched from an external source in our example C code:

```
int i = get_counter_value();
EXAMPLE_OUTPUT(i);
```

If the value `i` is not otherwise needed in this part of the program, we need not fetch it if no one has activated the probe. We can write:

```
if (EXAMPLE_OUTPUT_ENABLED()) {
  int i = get_counter_value();
  EXAMPLE_OUTPUT(i);
}
```

In this way, we can make even quite complicated probes more or less cost-less when not activated.

Different implementations of dynamic tracing have slightly different syntax both for the probes and the `.d` scripts. The need for header files and object file generation also differs between implementations, where for example MacOS X only requires one to generate a C header file from the probes definitions, while Solaris requires object code generation and linking steps as well. How the probes are actually written in the program also differs slightly. For all these reasons, some wrapping functionality is needed to support more than one variant of dynamic tracing.

SystemTap also differs in the syntax of the actual `.d` scripts, they are actual called SystemTap scripts or `.stp` scripts. The general idea is however similar.

### 4.3 Build Support for Dynamic Tracing

To enable building of a virtual machine with DTrace probes, several things had to be done:

1. Add support in the `configure` scripts for specifying that dynamic tracing was to be enabled and for detecting the flavor of dynamic tracing present on the system.

2. Create files for the probe `.d` scripts and wrappers to hide the peculiarities of this particular flavor of DTrace from the C code programmer.

3. Modify the `Makefile`s of the Virtual Machine so that the DTrace support was properly built regardless of DTrace implementation.

Most notably to the user, we added a new `configure` flag which indicates that dynamic tracing support is to be built into the Virtual Machine and that probes are to be added:

```
--enable-dynamic-trace={systemtap|dtrace}
```

Future support for the LTTng-UST[2] framework is planned. As LTTng-UST is expected to be too heavy to use for all the probes in the virtual machine, but is useful for user defined probes in the Erlang code, there is also an option to disable the probes in the virtual machine, while still retaining the support for dynamic tracing. As long as there is no LTTng-UST support implemented, that option has very limited use.

The wrapper header file `dtrace-wrapper.h` along with the actual probes definition file `erlang_dtrace.d` allows for the Erlang programmer to add probes in the format:

```
DTRACE<N>(<Probe>, <N parameters>);
```

The wrappers support up to eleven parameters to a single probe (with the `DTRACE11` macro) and also contains a macro for checking if a probe is activated:

```
DTRACE_ENABLED(name)
```

These macros will work regardless of framework. The actual provider, which is always `erlang`, need not be specified. All probe names need to be added to `erlang_dtrace.d` prior to use, so for example a probe that is put where a driver calls `driver_select` to stop monitoring a file descriptor could be specified in the `erlang_dtrace.d` file as:

```
probe driver__stop_select(char *name);
```

The probe is then inserted at the relevant place in the C code as:

```
DTRACE1(driver_stop_select, name);
```

Finally, in a `.d` script used for examining the running virtual machine, the probe can be activated with e.g.:

```
erlang*:::driver-stop_select
{
  printf("driver stop_select driver name %s\n", copyinstr(arg0));
}
```

As can be understood from the examples, the sequence `__` has special treatment in the framework. The C code should remove one `_` when referring to the probe and the `.d` script should replace it with a hyphen (`-`).

If SystemTap were used in the example above, the only difference would be that the script activating the probe would be a `.stp` file containing:

```
probe process("beam.smp").mark("driver-stop_select")
{
  printf("driver stop_select driver name %s\n", user_string($arg1));
}
```

The difference is most notable in how the probe is identified and in argument naming. The set of built-in functions is also different in SystemTap, so the user doing the actual tracing needs to be acquainted with the specific framework on the system. However, the Erlang virtual machine developer (or for that sake the Erlang developer) does not need to differentiate between the frameworks.

In the `configure` script, the type of compilation needed for the particular framework is determined. It can be either 1-step or 2-step. There is no special handling of SystemTap when compiling the `erlang_dtrace.d` file, but one needs to differentiate between 1-step platforms, where only a header is generated from the `.d` file during compilation, and the 2-step platforms, where object code is analyzed and a special object file is also generated from the DTrace compilation. The `configure`

[2]LTTng-UST: Linux Trace Toolkit Next Generation – User Space Tracer.

script does determine the type of compilation and the created `Makefile` for the virtual machine looks for the variable `DTRACE_ENABLED_2STEP` to determine if two step compilation is needed.

The `c_src` directory of the `runtime_tools` application has similar logic in its `Makefile`, as probes for Erlang code dynamic tracing (which will be described later) are generated there.

### 4.4 Virtual Machine Probes

With the wrappers and build support in place, we proceeded with adding probes for relevant parts of the virtual machine. The main areas where probes were added are:

- Distribution.
- Driver events.
- File handling.
- Erlang Function calls (including NIFs and BIFs).
- Garbage collection.
- Memory related operations like heap growth and shrinking.
- Message sending and receiving for Erlang processes.
- Operations on ports (like sending data to them).
- Erlang process scheduling events.
- Erlang process creation and destruction.

For each of these areas, there are example `.d` scripts in the `runtime_tools` application, which demonstrates how they can be used. The user doing the tracing can use these example files as a reference for what different probes are available. Whenever new probes are added, the example files are expected to be updated.

Most of the probes are quite trivial to add. In some cases information internal to the virtual machine needs to be serialized (or converted to readable strings) in which case the `DTRACE_ENABLED` macro is used to avoid overhead when the probe is not active. In total, the virtual machine of Erlang/OTP R16B contains 154 probe points (for 61 distinct probes), of which many have special handling of parameters, requiring `DTRACE_ENABLED` to be used.

By far the most complicated instrumentation is done in the file I/O part of the system. For example, there are 67 probe points in the file `efile_drv` alone. One reason is that the file I/O in the virtual machine usually happens in separate OS threads, so that job is dispatched over worker threads doing the actual I/O. There is therefore need of probes both when the I/O is initialized and when it is actually performed, as well as when the result is reported back via the port to Erlang.

To further complicate things when it comes to I/O, the Erlang process actually invoking the I/O operation via a port is usually *not* the process that actually initiated the operation. To begin with, Erlang's file operations are distributed and, to the Erlang process, the file appears as another process identifier, either on the local node or in the network. Furthermore utilities like Disk-based Erlang Term Storage (`dets`) and databases like `mnesia` move the operations further away from the process actually requesting the data from disk. This problem was the reason for the introduction of dynamic trace tags into the virtual machine, a feature described later in this section.

Most of the probes take several parameters, which are described both in comments in the `erlang_dtrace.d` file and in the examples of `runtime_tools`.

If the virtual machine is compiled without dynamic tracing support, all code having anything to do with DTrace is removed. There is not a single unnecessary line added to the virtual machine if it is not configured with `--enable-dynamic-trace`. However, tests have shown that the difference in performance between a virtual machine with dynamic tracing completely disabled and a virtual machine where probes are present but not activated, is next to immeasurable. In the future we will probably see the probes in the default builds on platforms where DTrace is supported.

### 4.5 Support for Probes in Erlang Code

In addition to the probes in the Virtual Machine, a system written in Erlang may want to add its own trace probes. There is no generic support for doing this in the dynamic trace frameworks used. Probes are expected to exist in ELF binaries and are activated by the kernel when a `.d` or `.stp` script is "executed" to enable the probes. While the frameworks could probably be extended to enable virtual machines to report probes in dynamically loaded virtual machine code, there is no such support today. To implement such dynamic support, one would need to implement:

- Dynamic adding of probes not present in the executable of the VM at compile time.
- A callback interface where the VM could get informed about when a probe is active. The interface would have to be asynchronous; polling for enabled probes would give too much latency.
- An interface to dynamically report an event on a probe, i.e., that the code executed by the virtual machine has passed a probe point.

Such functionality is present on some platforms via the `libsdt` library, but not on all platforms. The nature of current frameworks makes such additions somewhat hard to add, if they are not already present. In the future, the optimal solution would however be to have such functionality on all platforms.

The future implementation of support for LTTng-UST might however involve adding such mechanisms. As its name suggests, LTTng-UST is implemented purely in user space and adding and experimenting with such functionality will probably be much easier than with the current frameworks.

The solution chosen for Erlang is instead to add a large amount of static probes. No less than 951 probes, named `user_trace-n0` to `user_trace-n950` are (conceptually) present in a NIF library placed in `runtime_tools`. The Erlang application should try to use distinctive probes for all its probe points, to avoid firing probes that the tracing part is not interested in.

The Erlang programmer can insert the probes in the Erlang code in much the same way as the C programmer does:

```
dyntrace:pn(Number, ...),
```

to insert the probe `user_trace-n` in the code. The parameters can be up to four integers followed by up to four byte-lists (`io_data`).

For example if we add a probe like this to the Erlang code:

```
dyntrace:pn(42, 1, "Hello"),
```

it will be caught by the following `.d` script:

```
erlang*:::user_trace-n42
{
  printf(Probe 42 fired: %s %s %d %d %d %d '%s' '%s' '%s' '%s'\n",
         copyinstr(arg0),
         arg1 == NULL ? "" : copyinstr(arg1),
         arg2, arg3, arg4, arg5,
         arg6 == NULL ? "" : copyinstr(arg6),
         arg7 == NULL ? "" : copyinstr(arg7),
         arg8 == NULL ? "" : copyinstr(arg8),
         arg9 == NULL ? "" : copyinstr(arg9));
}
```

and the output will be:

```
Probe 42 fired: <0.32.0>  1 0 0 0 'Hello' '' '' ''
```

whenever the program execution passes the probe. The first argument is always the Erlang PID (process identifier) of the process that fires the probe, the second argument is the dynamic trace tag (which will be discussed later) and the rest of the arguments are the parameters supplied when defining the probe. If an integer value is not defined, it will be `0` and if a string value is not defined, it will be an empty string (or `NULL`, depending on the underlying DTrace implementation).

There is also an interface where one does not supply a "probe number". All those probes end up in the `user_trace-i4s4` probe, which will be heavily fired in a system where a lot of Erlang code uses this probe. The use of `dyntrace:p()` and the underlying `user_trace-i4s4` probe is therefore strongly discouraged; it is retained purely for backwards compatibility.

The implementation of the user probes is conceptually put in a NIF (Native Interface Function) library which is loaded whenever the `dyntrace` module is loaded. When the execution of an Erlang program passes the "probe" (i.e., the call to `dyntrace:pn`, a function in the NIF code is called, which determines if the probe is activated by `.d` scripts, and in that case it marshals the Erlang terms and fires the appropriate probe. All the probes are available in the function in a huge switch clause, as they need to be statically compiled into the code, all 951 of them.

This implementation makes probes in Erlang code somewhat costly even if they are not enabled; they always imply a call to the `dyntraced` module and a jump to the NIF library before we can see if the probe is enabled at all. Instrumenting an Erlang application therefore needs to be done carefully to avoid performance penalties. It is however still a very useful lightweight and dynamic tracing tool.

The current implementation of dynamic tracing frameworks unfortunately requires the actual probes to be in the virtual machine executable, so the probes are currently only conceptually in the NIF library. It then uses an undocumented callback into the virtual machine to actually fire the probe. This somewhat clutters the VM implementation, but it is the only portable solution today.

### 4.6 Support for DTrace Tags in Erlang Code and in Probes

As mentioned earlier, one may want to identify which part of the system fired a probe by other means than the process ID. This may be the case if, for example, we have many processes running the same library code and we do not want to distinguish between them, or if we have no knowledge about which process originally initiated an operation. The later is the usual scenario when it comes to file I/O.

To address these problems, the concept of dynamic trace tags is added. In its simplest form, it behaves more or less as an entry in the process dictionary of the process, a string that is sent to the tracing party in certain probes, most notably the user defined ones described in the previous chapter. If we go back to the example:

```
dyntrace:pn(42, 1, "Hello"),
```

we could add a dynamic trace tag to the process to distinguish it from other occurrences of the same probe, either prior to calling a library function that contains the probe or at some either stage of the processing:

```
dyntrace:put_tag(<<"my_tag">>)
```

The tag is a binary, but automatic conversion will take place if it is given as a mixed list of characters and binaries. Characters in lists beyond code point 127 will be encoded in UTF-8. If we then pass the probe in our program, the current trace tag of the process will be supplied as the second argument to the `.d` script that has enabled the probe, so that the output from running a trace will now be:

```
Probe 42 fired: <0.32.0> my_tag 1 0 0 0 'Hello' '' '' ''
```

While this is useful for user defined probes, it is more or less useless when operating on files. For file I/O, the tag of the calling process needs to be spread along with the requests sent to other Erlang processes in the system for it to end up in the file driver and the actual probes. This is implemented using a mechanism similar to sequential tracing. If there is a trace tag present in the

current Erlang process, a call to `dyntrace:spread_tag(true)` will contaminate the next message sent from the process with the tag, so that the receiving process will continue to spread the tag until it receives a message without the tag. The tag will then spread between processes all the way down to the actual file driver, where it is picked up by the probes and it ultimately gets sent to the tracing part.

Spreading tags is automatically done by all file operations in Erlang if dynamic tracing is enabled. Some operations also need to save and restore the tag to avoid sending it to the wrong processes or to propagate a tag to a newly created process. The `spread_tag` interface returns an opaque state that was the complete previous state of tags and their spreading. The opaque state can later be used to restore this state using `dyntrace:restore_state/1`. Combining these two interfaces allows for library functions to control to which processes tags are spread without permanently altering the state of the process.

The use of `dyntrace:spread_tag/1` and `dyntrace:restore_tag/1` is described in the reference manual for those who need to use them, but for most Erlang programmers, they will just automatically be used by the `file` interface and any tag stored in the process doing the call to `file` will end up in the probe. The only thing the Erlang programmer has to do is to set the tag:

```
dyntrace:put_tag("my_tag"),
...
file:read_file("port1.d"),
```

If we then use a `.d` script to view file operations (as in the example file `efile_drv.d`), we will see output on the lines of:

```
efile_drv enter tag={0,655} user tag my_tag | READ_FILE (15) | args: port1.d,
  0 0 (port #Port<0.586>)
```

The part `tag={0,655}` is a request identifier making it possible to follow the request to the asynchronous threads doing the actual I/O operation and has nothing to do with the user defined tag. The actual tag we set however also appears as `my_tag` and we can therefore identify which process initiated the I/O. The beauty of it is that the tag automatically spreads with the Erlang messages to the driver without concerning the intermediate processes in the operation.

To limit the impact on systems that do not need dynamic tracing, all code handling dynamic trace tags is automatically removed when loaded into a system where dynamic tracing is not enabled, so every call to the built-in functions is simply replaced with a no-op (or possibly with the value `false`). Handling of dynamic trace tags can therefore be placed in any Erlang library code of the system without the risk of affecting performance-critical systems without support for dynamic tracing. To get the code completely removed, one can use special functions in the Erlang module which correspond to the functions in the `dyntrace` module for handling tags; this approach even avoids the external call to `dyntrace` so that the loaded virtual machine code contains no trace of the original tag-handling code. This scheme has allowed for adding dynamic trace tag handling to the very kernel of Erlang/OTP, with no performance impact whatsoever for regular users of Erlang.

## 4.7 Profiling in Erlang using DTrace

Profilers for Erlang applications so far (e.g., `percept`) have been based on Erlang's built-in tracing mechanism. Using the dynamic tracing offered by DTrace presents several advantages, as we have already discussed, the principal one being a much smaller performance penalty when profiling. DTrace can be used for profiling Erlang applications in a quite simple way. Instead of using Erlang's built-in mechanism which creates and sends trace messages every time that a traced event occurs, a dynamic DTrace probe is fired instead.

For this purpose, a number of additions were necessary to the DTrace probes that were already present in the Erlang virtual machine. Essentially, a DTrace probe had to be associated with each and every Erlang trace and profiling message. We needed to ensure that:

- each DTrace probe carries the same information with the corresponding trace (or profiling) message, and

| Erlang trace message | DTrace probe |
|---|---|
| receive | message-queued |
| send | message-send |
| send_to_non_existing_process | - |
| call | bif-entry, global-function-entry, local-function-entry, nif-entry |
| return_to | - |
| return_from | function-return, bif-return, nif-return |
| exception_from | - |
| spawn | process-spawn |
| exit | process-exit |
| link | process-link, port-link |
| unlink | process-unlink, port-unlink |
| getting_linked | process-getting_linked, port-getting_linked |
| getting_unlinked | process-getting_unlinked, port-getting_unlinked |
| register | process-registered, port-registered |
| unregister | process-unregistered, port-unregistered |
| in | process-scheduled |
| out | process-unscheduled |
| gc_start | gc_major-start, gc_minor-start |
| gc_end | gc_major-end, gc_minor-end |

Table 1: Erlang trace message to DTrace probe correspondence.

| Erlang profiling message | DTrace probe |
|---|---|
| profile_start | - |
| profile_stop | - |
| profile (processes) | process-active, process-inactive |
| profile (ports) | port-active, port-inactive |
| profile (schedulers) | scheduler-active, scheduler-inactive |

Table 2: Erlang profiling message to DTrace probe correspondence.

- each DTrace probe is fired whenever the corresponding trace (or profiling) message is sent.

As far as the first task is concerned:

- we specified several new DTrace probes (e.g. process-registered, process-link),
- we modified some of the existing DTrace probes, in order to ensure they contain all the required information (e.g., we had to add the MFA in process-unscheduled), and
- we added timestamps to all new and existing DTrace probes that we wanted to associate with some trace (or profiling) message.

The correspondence between Erlang trace messages and DTrace probes is shown in Table 1, and the correspondence between Erlang profiling messages and DTrace probes is shown in Table 2.

Note that, although our initial goal was to make all DTrace probes carry the same information with the corresponding trace (or profiling) messages, there were cases that we decided not to do that. The pieces of information that we decided not to include in the DTrace probes were the contents of messages sent and received (we included the size instead) and the arguments of the function calls (we included the arity). This was a deliberate choice, to reduce the number of bytes communicated through DTrace probes and therefore to reduce the performance penalty incurred by DTrace when probes are enabled.

A summary of the probes used for profiling purposes is given below. For each probe, the description contains the event that causes it, the names and types of parameters, and the kind of information that is passed in them.

**Probe:** message-queued

**Fired:** whenever a message is queued to a process

**Header:**

```
probe message__queued(char *receiver, uint32_t size, uint32_t queue_len,
                      int token_label, int token_previous, int token_current,
                      uint64_t ts)
```

**Parameters:**

- `receiver`: the PID of the receiver
- `size`: the size of the message (in words)
- `queue_len`: the size of the queue of the receiver
- `token_label`: the label of the sender's sequential trace token
- `token_previous`: the previous count of the sender's sequential trace token
- `token_current`: the current count of the sender's sequential trace token
- `ts`: the timestamp (in microseconds)

**Probe:** message-send

**Fired:** whenever a message is sent

**Header:**

```
probe message__send(char *sender, char *receiver, uint32_t size,
                    int token_label, int token_previous, int token_current,
                    uint64_t ts)
```

**Parameters:**

- `sender`: the PID of the sender
- `receiver`: the PID of the receiver
- `size`: the size of the message (in words)
- `queue_len`: the size of the queue of the receiver
- `token_label`: the label of the sender's sequential trace token
- `token_previous`: the previous count of the sender's sequential trace token
- `token_current`: the current count of the sender's sequential trace token
- `ts`: the timestamp (in microseconds)

**Probe:** bif-entry

**Fired:** whenever a Built-In Function (BIF) is called

**Header:**

```
probe bif__entry(char *p, char *mfa, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the caller
- `mfa`: the MFA for the called BIF

- `ts`: the timestamp (in microseconds)

**Probe:** global-function-entry

**Fired:** whenever an external function is called

**Header:**

```
probe global__function__entry(char *p, char *mfa, int depth, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the caller
- `mfa`: the MFA for the called function
- `depth`: the stack depth
- `ts`: the timestamp (in microseconds)

**Probe:** local-function-entry

**Fired:** whenever a local function is called

**Header:**

```
probe local__function__entry(char *p, char *mfa, int depth, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the caller
- `mfa`: the MFA for the called function
- `depth`: the stack depth
- `ts`: the timestamp (in microseconds)

**Probe:** nif-entry

**Fired:** whenever a Native Implemented Function (NIF) is called

**Header:**

```
probe nif__entry(char *p, char *mfa, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the caller
- `mfa`: the MFA for the called NIF
- `ts`: the timestamp (in microseconds)

**Probe:** function-return

**Fired:** whenever a user function returns

**Header:**

```
probe function__return(char *p, char *mfa, int depth, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the caller
- `mfa`: the MFA for the called function
- `depth`: the stack depth
- `ts`: the timestamp (in microseconds)

**Probe:** bif-return

**Fired:** whenever a Built-In Function (BIF) returns

**Header:**

```
probe bif__return(char *p, char *mfa, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the caller
- `mfa`: the MFA for the called BIF
- `ts`: the timestamp (in microseconds)

**Probe:** nif-return

**Fired:** whenever a Native Implemented Function (NIF) returns

**Header:**

```
probe nif__return(char *p, char *mfa, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the caller
- `mfa`: the MFA for the called NIF
- `ts`: the timestamp (in microseconds)

**Probe:** process-spawn

**Fired:** whenever a new process is spawned

**Header:**

```
probe process__spawn(char *p, char *p2, char *mfa, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the new process
- `p2`: the PID of the parent process
- `mfa`: the MFA for the entry point of the new process
- `ts`: the timestamp (in microseconds)

**Probe:** process-exit

**Fired:** whenever a process exits

**Header:**

```
probe process__exit(char *p, char *reason, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process that exited
- `reason`: the exit reason
- `ts`: the timestamp (in microseconds)

**Probe:** process-link

**Fired:** whenever one process links to another

**Header:**

```
probe process__link(char *p, char *p2, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process
- `p2`: the PID of the process that p links to
- `ts`: the timestamp (in microseconds)

**Probe:** port-link

**Fired:** whenever a process links to a port

**Header:**

```
probe port__link(char *p, char *port, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process
- `port`: the port ID of the port that p links to
- `ts`: the timestamp (in microseconds)

**Probe:** process-unlink

**Fired:** whenever a process removes its link to another process

**Header:**

```
probe process__unlink(char *p, char *p2, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process
- `p2`: the PID of the process that p unlinks from
- `ts`: the timestamp (in microseconds)

**Probe:** port-unlink

**Fired:** whenever a process removes its link to a port

**Header:**

```
probe port__unlink(char *p, char *port, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process
- `port`: the port ID of the port that p unlinks from
- `ts`: the timestamp (in microseconds)

**Probe:** process-getting_linked

**Fired:** whenever a process gets linked to another process

**Header:**

```
probe process__getting_linked(char *p, char *p2, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process
- `p2`: the PID of the process that p gets linked to
- `ts`: the timestamp (in microseconds)

**Probe:** port-getting_linked

**Fired:** whenever a port gets linked to a process

**Header:**

```
probe port__getting_linked(char *p, char *port, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process
- `port`: the port ID of the port that gets linked to p
- `ts`: the timestamp (in microseconds)

**Probe:** process-getting_unlinked

**Fired:** whenever one process gets unlinked from another process

**Header:**

```
probe process__getting_unlinked(char *p, char *p2, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process
- `p2`: the PID of the process that p gets unlinked from
- `ts`: the timestamp (in microseconds)

**Probe:** port-getting_unlinked

**Fired:** whenever a port gets unlinked from a process

**Header:**

```
probe port__getting_unlinked(char *p, char *port, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process
- `port`: the port ID of the port that gets unlinked from p
- `ts`: the timestamp (in microseconds)

**Probe:** process-registered

**Fired:** whenever a process is registered with a name

**Header:**

```
probe process__registered(char *p, char *name, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the registered process
- `name`: the name that is associated with the process
- `ts`: the timestamp (in microseconds)

**Probe:** port-registered

**Fired:** whenever a port is registered with a name

**Header:**

```
probe port__registered(char *port, char *name, uint64_t ts)
```

**Parameters:**

- `port`: the port ID of the registered port
- `name`: the name that is associated with the port
- `ts`: the timestamp (in microseconds)

**Probe:** process-unregistered

**Fired:** whenever a process is unregistered

**Header:**

```
probe process__unregistered(char *p, char *name, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the unregistered process
- `name`: the name that was associated with the process
- `ts`: the timestamp (in microseconds)

**Probe:** port-unregistered

**Fired:** whenever a port is unregistered

**Header:**

```
probe port__unregistered(char *port, char *name, uint64_t ts)
```

**Parameters:**

- `port`: the port ID of the unregistered port
- `name`: the name that was associated with the port
- `ts`: the timestamp (in microseconds)

**Probe:** process-scheduled

**Fired:** whenever a process is scheduled

**Header:**

```
probe process__scheduled(char *p, char *mfa, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the scheduled process
- `mfa`: the MFA for the function that will be executed next
- `ts`: the timestamp (in microseconds)

**Probe:** process-unscheduled

**Fired:** whenever a process is unscheduled

**Header:**

```
probe process__unscheduled(char *p, char *mfa, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the unscheduled process
- `mfa`: the MFA for the function that was being executed
- `ts`: the timestamp (in microseconds)

**Probe:** gc_major-start

**Fired:** whenever a major garbage collection starts

**Header:**

```
probe gc_major__start(char *p, int need, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process, for which the garbage collection takes place
- `need`: the number of words that p needs
- `ts`: the timestamp (in microseconds)

**Probe:** gc_minor-start

**Fired:** whenever a minor garbage collection starts

**Header:**

```
probe gc_minor__start(char *p, int need, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process, for which the garbage collection takes place
- `need`: the number of words that p needs
- `ts`: the timestamp (in microseconds)

**Probe:** gc_major-end

**Fired:** whenever a major garbage collection completes

**Header:**

```
probe gc_major__end(char *p, int reclaimed, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process, for which the garbage collection took place
- `reclaimed`: the number of words that were reclaimed
- `ts`: the timestamp (in microseconds)

**Probe:** gc_minor-end

**Fired:** whenever a minor garbage collection completes

**Header:**

```
probe gc_minor__end(char *p, int reclaimed, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the process, for which the garbage collection took place
- `reclaimed`: the number of words that were reclaimed
- `ts`: the timestamp (in microseconds)

**Probe:** process-active

**Fired:** whenever a process becomes active

**Header:**

```
probe process__active(char *p, char *mfa, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the active process
- `mfa`: the MFA for the function that will be executed next
- `ts`: the timestamp (in microseconds)

**Probe:** process-inactive

**Fired:** whenever a process becomes inactive

**Header:**

```
probe process__inactive(char *p, char *mfa, uint64_t ts)
```

**Parameters:**

- `p`: the PID of the inactive process
- `mfa`: the MFA for the function that was being executed
- `ts`: the timestamp (in microseconds)

**Probe:** port-active

**Fired:** whenever a port becomes active

**Header:**

```
probe port__active(char *port, char *mfa, uint64_t ts)
```

**Parameters:**

- `port`: the port ID of the active port
- `mfa`: the MFA for the function that will be executed next
- `ts`: the timestamp (in microseconds)

**Probe:** port-inactive

**Fired:** whenever a port becomes inactive

**Header:**

```
probe port__inactive(char *port, char *mfa, uint64_t ts)
```

**Parameters:**

- `port`: the port ID of the inactive port
- `mfa`: the MFA for the function that was being executed
- `ts`: the timestamp (in microseconds)

**Probe:** scheduler-active

**Fired:** whenever a scheduler becomes active

**Header:**

```
probe scheduler__active(char *s, int active, uint64_t ts)
```

**Parameters:**

- `s`: the ID of the active scheduler
- `active`: the number of currently active schedulers
- `ts`: the timestamp (in microseconds)

**Probe:** scheduler-inactive

**Fired:** whenever a scheduler becomes inactive

**Header:**

```
probe scheduler__inactive(char *s, int active, uint64_t ts)
```

**Parameters:**

- `s`: the ID of the inactive scheduler
- `active`: the number of currently active schedulers
- `ts`: the timestamp (in microseconds)

### 4.8 Future Work

A few areas of improvement have been mentioned earlier in the text of this section. The user-defined probes could be made purely dynamic if we had `libsdt` on all supported platforms, the user probes could be placed in the NIF library proper if all the frameworks supported it, and we should definitely look into extending the support to the LTTng-UST framework, possibly allowing for purely dynamic probes at least for that framework.

The handling of dynamic trace tags should be extended to more interfaces in Erlang/OTP, and the trace tag could be used by more process related probes.

More probes are also definitely possible to add to the Virtual Machine, with accompanying example scripts. More examples showing how to do more elaborate things in `.d` scripts would also be useful.

On platforms with a stable implementation of DTrace, the instrumented build can also be made the default one. In the near future that mainly concerns MacOS X, FreeBSD and Oracle Solaris, but when future Linux kernels contain support for SystemTap by default, even Linux builds can have dynamic tracing enabled by default.

On the side of profiling and monitoring tools for Erlang/OTP, we have the opportunity to utilize DTrace to monitor a running system with DTrace enabled, with minimal impact on normal performance. One can imagine both process monitoring tools, I/O performance measurement, and other profiling tools based on the output of tailored `.d` scripts. Along this line, we are currently using our DTrace-based tracing and profiling infrastructure, in order to build an alternative back-end for the `percept2` tool, developed as part of WP5 of RELEASE. We expect to gather enough data from this experiment that will help us compare our infrastructure with the existing one.

## 5 Preliminary Port of Erlang/OTP to Blue Gene/Q

### 5.1 Blue Gene/Q Architecture

The Blue Gene/Q system, shown in Figure 2, is a massively parallel computer system from the Blue Gene computer architecture series by IBM. It is divided into racks, in which nodes are specialized for either computation or handling I/O. Additional hardware is used for the storage subsystem, and certain specialized nodes. These are the *service nodes* and the *front end nodes*. A service node is used by on-site administrators to control and configure the Blue Gene/Q system. To improve scalability it is possible to use more than one service node in a system. Front end nodes are interactive login nodes for system users. Only from these nodes a user can get access to the rest of the Blue Gene/Q system. The front end nodes also provide the necessary tools (e.g., compilers) and allow access to the job control system. Additionally they can be used for pre- and post-processing of data like any normal Linux computer system.

Figure 3 contains a depiction of the individual building blocks of a Blue Gene/Q system. The Blue Gene/Q Application Development handbook sums this up as follows:

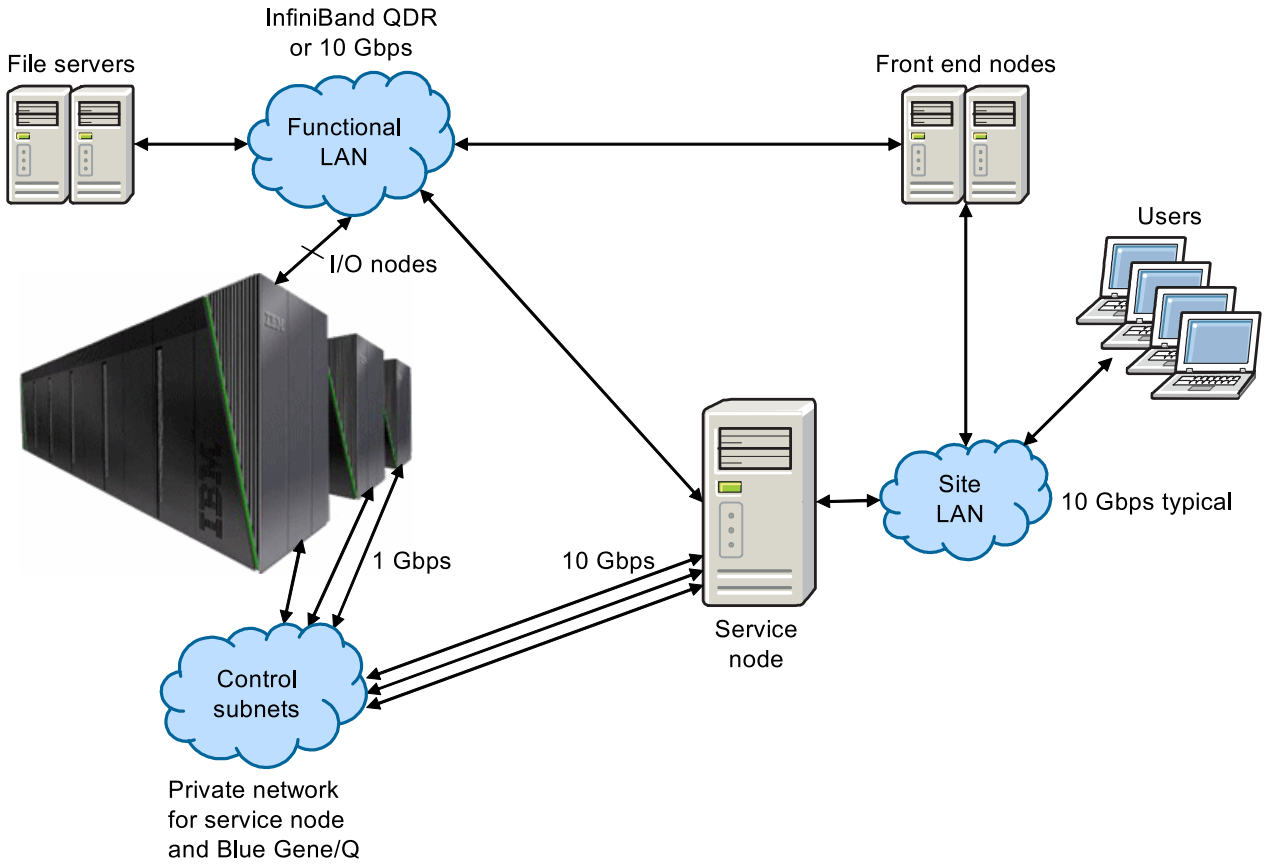


Figure 2: Blue Gene/Q system architecture (from Blue Gene/Q Application Development).

> Compute cards contain 16 IBM Blue Gene/Q PowerPC(R) A2 core processors and 16 GB of memory. Thirty-two such cards plug into a node board and 16 node boards are contained in a midplane. A Blue Gene/Q compute rack has either one (half rack configuration) or two fully populated midplanes. The system can be scaled to 512 compute racks. Compute racks components are cooled either by water or air. Water is used for the processing nodes. Air is used for the power supplies and the I/O drawers mounted in the Blue Gene/Q rack. Eight I/O nodes are housed in each I/O drawer. In the compute rack, up to four I/O drawers, two per midplane, can be configured using the I/O enclosure.

The front end nodes and I/O nodes are running a normal Linux kernel and system, while compute nodes are running a specialized Compute Node Kernel.

## 5.2 Porting Challenges

The first challenge in porting a complex computer program, like the full Erlang/OTP system, to the Blue Gene/Q architecture is related to the fact that different processors are used in front end nodes and compute nodes. While their ABI is somewhat compatible, the job control system denies running code that has not been specifically compiled for the compute nodes. This means one is effectively restricted to the compilers provided by IBM, which are IBM's own XL compiler, and a specialized version of the GNU Compiler Collection (GCC). While IBM recommends not using the GNU compilers for performance reasons, for this preliminary port we decided it was a better option as this compiler is the preferred compiler for the Erlang/OTP distribution. So, for the preliminary port of Erlang/OTP we used GCC.

The second, and bigger, challenge is related to the fact that there are some major restrictions on the Compute Node Kernel (CNK) in comparison to a normal Linux kernel system. These restrictions are because compute nodes specialize in performing (in-memory) computation as opposed to I/O. This means that a compute node kernel requires the help of I/O nodes for I/O operations, effectively

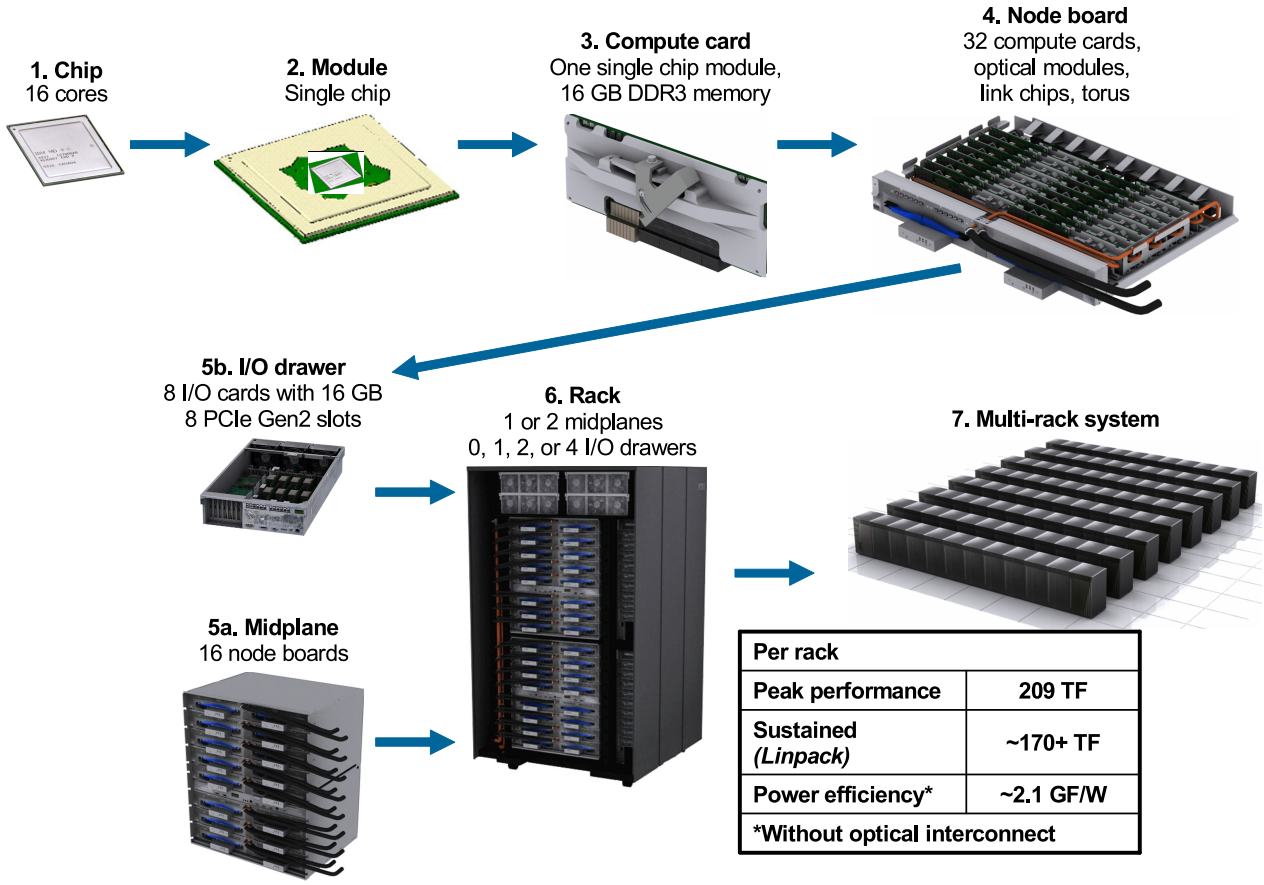


Figure 3: Blue Gene/Q hardware overview (from Blue Gene/Q Application Development).

transferring I/O kernel calls over the network to an I/O node for execution. This in turn means that I/O operations are more expensive than on a Linux kernel. Additionally, some kernel calls are unavailable on the CNK. Unfortunately, these include common calls like `fork()` and `pipe()`, which are heavily used by the Erlang/OTP Runtime System. It is therefore necessary to change the Erlang/OTP system to not use these calls when it is compiled for the CNK.

### 5.3 Current Port Status

For porting Erlang/OTP to the Blue Gene/Q we needed access to EDF's machine ("ZUMBROTA"). Due to administrative reasons this access was granted to the UU and ICCS team members only in late January, so the porting effort is relatively early in its progress at the time of this writing. Still, some non-trivial progress towards a fully working port has been achieved:

- We have verified that the Erlang/OTP system compiles and runs successfully on the front end nodes of the machine. This includes the complete Erlang RunTime System (ERTS), the BEAM bytecode compiler, the HiPE native code compiler (generating PPC64 code), and the complete set of Erlang/OTP libraries. This part of the porting effort required very few adjustments to Erlang/OTP's code base.

- On the compute nodes, the compilation of Erlang/OTP requires the use of a specialized compiler. For this reason, we developed a cross-compiling (`xcomp`) configuration file, which allows the build system to issue the correct commands automatically. This file, whose code is shown in Appendix A, has been included in Erlang/OTP R16B (release 16B).

### 5.4 Future Work

The main obstacle for a fully working Erlang/OTP system on the compute nodes remains the missing kernel calls. The `fork()` call is only used to detect certain hardware features in the threaded version of the VM, so it is possible to disable these. Similarly, `clock_gettime()` can be replaced by a less precise way of measuring time.

Unfortunately, this is not as easy for `pipe()` calls, as these are used extensively for internal communication in the Erlang VM, to allow waking threads that are polling file descriptors. It will require additional effort to rewrite the internal communication to avoid the requirement for pipes.

## 6 Concluding Remarks

We have described the design and implementation of two key components of the prototype Erlang Virtual Machine release. The first is a scalable implementation of ETS, the Erlang Term Storage, which allows Erlang processes to efficiently share data in parallel applications. The second is its efficient tracing support based on DTrace/SystemTap for profiling and monitoring Erlang applications, which is expected to play a significant role in the profiling tools developed as part of WP5 of RELEASE. In addition, we reported on the status of a preliminary port of Erlang/OTP on the Blue Gene/Q architecture. All these parts of this deliverable are currently released as open source and available to the Erlang programming community as part of Erlang/OTP R16B.

Although we have used the word "prototype" to describe this release of the scalable Erlang VM, its implementation in Erlang/OTP R16B is actually quite robust and is being used in (commercial) Erlang applications. Still, there is plenty of room for further scalability improvements and extensions as described in the future work sections of this document. We are currently working on them. Their realization will be part of D2.4 ("Robust Scalable VM").

## Acknowledgments

Scott Lystig Fritchie cannot be thanked enough for his work on DTrace support for Erlang/OTP, which provided the basics for the current implementation and was finally merged into the official Erlang/OTP source code starting with release R15B01.

## Change Log

| Version | Date | Comments |
|---|---|---|
| 0.1 | 8/4/2013 | First Version Submitted to the Commission Services |

## References

[1] S. L. Fritchie. DTrace and Erlang: A new beginning. Presented at the Erlang User Conference, Nov. 2011.

[2] B. Gregg and J. Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall, 2011.

[3] `cprof`. A simple Call Count Profiling Tool. URL http://erlang.org/doc/man/cprof.html.

[4] `dbg`. The Text Based Trace Facility. URL http://erlang.org/doc/man/dbg.html.

[5] `eprof`. A Time Profiling Tool. URL http://erlang.org/doc/man/eprof.html.

[6] `et`. Event Tracer Reference Manual, Version 1.4.4.3. URL http://erlang.org/doc/apps/et.

[7] `etop`. Erlang Top Tool. URL http://erlang.org/doc/man/etop.html.

[8] `ets`. Erlang Term Storage. URL http://erlang.org/doc/man/ets.html.

[9] `fprof`. A Time Profiling Tool Using Trace to File. URL http://erlang.org/doc/man/fprof.html.

[10] `lcnt`. A runtime system Lock Profiling tool. URL http://erlang.org/doc/man/lcnt.html.

[11] `mnesia`. Mnesia Reference Manual, Version 4.8. URL http://erlang.org/doc/apps/mnesia.

[12] `percept`. Erlang Concurrency Profiling Tool Reference Manual, Version 0.8.8. URL http://erlang.org/doc/apps/percept.

[13] `pman`. Process Manager Reference Manual, Version 2.7.1.4. URL http://erlang.org/doc/apps/pman.

[14] `ttb`. A base for building trace tools for distributed systems. URL http://erlang.org/doc/man/ttb.html.

# A The `erl-xcomp-powerpc64-bgq-linux.conf` Configuration File

The following file, which is part of Erlang/OTP R16B, enables cross compilation of the Erlang/OTP system on the compute nodes of Blue Gene/Q.

```
## -*-shell-script-*-
##
## ... Copyright Statement deleted ...
##
## ----------------------------------------------------------------------
## When cross compiling Erlang/OTP using 'otp_build', copy this file and set
## the variables needed below. Then pass the path to the copy of this file as
## an argument to 'otp_build' in the configure stage:
##   'otp_build configure --xcomp-conf=<FILE>'
## ----------------------------------------------------------------------

## Note that you cannot define arbitrary variables in a cross compilation
## configuration file. Only the ones listed below will be guaranteed to be
## visible throughout the whole execution of all 'configure' scripts. Other
## variables needs to be defined as arguments to 'configure' or exported in
## the environment.

## -- Variables for 'otp_build' Only ---------------------------------------

## Variables in this section are only used, when configuring Erlang/OTP for
## cross compilation using '$ERL_TOP/otp_build configure'.

## *NOTE*! These variables currently have *no* effect if you configure using
## the 'configure' script directly.

# * 'erl_xcomp_build' - The build system used. This value will be passed as
#   '--build=$erl_xcomp_build' argument to the 'configure' script. It does
#   not have to be a full 'CPU-VENDOR-OS' triplet, but can be. The full
#   'CPU-VENDOR-OS' triplet will be created by
#   '$ERL_TOP/erts/autoconf/config.sub $erl_xcomp_build'. If set to 'guess',
#   the build system will be guessed using
#   '$ERL_TOP/erts/autoconf/config.guess'.
erl_xcomp_build=guess

# * 'erl_xcomp_host' - Cross host/target system to build for. This value will
#   be passed as '--host=$erl_xcomp_host' argument to the 'configure' script.
#   It does not have to be a full 'CPU-VENDOR-OS' triplet, but can be. The
#   full 'CPU-VENDOR-OS' triplet will be created by
#   '$ERL_TOP/erts/autoconf/config.sub $erl_xcomp_host'.
erl_xcomp_host=powerpc64-bgq-linux

# * 'erl_xcomp_configure_flags' - Extra configure flags to pass to the
#   'configure' script.
erl_xcomp_configure_flags="--without-termcap"

## -- Cross Compiler and Other Tools ---------------------------------------

## If the cross compilation tools are prefixed by '<HOST>-' you probably do
## not need to set these variables (where '<HOST>' is what has been passed as
## '--host=<HOST>' argument to 'configure').

## This path should really be part of the user's PATH environment, but
## since it is highly unlikely that it will differ between Blue Gene/Q
## installations, the path is hard-coded here for convenience.
TOP_BIN=/bgsys/drivers/ppcfloor/gnu-linux/bin

## All variables in this section can also be used when native compiling.

# * 'CC' - C compiler.
CC=${TOP_BIN}/${erl_xcomp_host}-gcc
```

```
# * 'CFLAGS' - C compiler flags.
#CFLAGS=

# * 'STATIC_CFLAGS' - Static C compiler flags.
#STATIC_CFLAGS=

# * 'CFLAG_RUNTIME_LIBRARY_PATH' - This flag should set runtime library
#   search path for the shared libraries. Note that this actually is a
#   linker flag, but it needs to be passed via the compiler.
#CFLAG_RUNTIME_LIBRARY_PATH=

# * 'CPP' - C pre-processor.
#CPP=

# * 'CPPFLAGS' - C pre-processor flags.
#CPPFLAGS=

# * 'CXX' - C++ compiler.
CXX=${TOP_BIN}/${erl_xcomp_host}-g++

# * 'CXXFLAGS' - C++ compiler flags.
#CXXFLAGS=

# * 'LD' - Linker.
LD=${TOP_BIN}/${erl_xcomp_host}-ld

# * 'LDFLAGS' - Linker flags.
#LDFLAGS=

# * 'LIBS' - Libraries.
#LIBS=

## -- *D*ynamic *E*rlang *D*river Linking --

## *NOTE*! Either set all or none of the 'DED_LD*' variables.

# * 'DED_LD' - Linker for Dynamically loaded Erlang Drivers.
#DED_LD=

# * 'DED_LDFLAGS' - Linker flags to use with 'DED_LD'.
#DED_LDFLAGS=

# * 'DED_LD_FLAG_RUNTIME_LIBRARY_PATH' - This flag should set runtime library
#   search path for shared libraries when linking with 'DED_LD'.
#DED_LD_FLAG_RUNTIME_LIBRARY_PATH=

## -- Large File Support --

## *NOTE*! Either set all or none of the 'LFS_*' variables.

# * 'LFS_CFLAGS' - Large file support C compiler flags.
#LFS_CFLAGS=

# * 'LFS_LDFLAGS' - Large file support linker flags.
#LFS_LDFLAGS=

# * 'LFS_LIBS' - Large file support libraries.
#LFS_LIBS=

## -- Other Tools --

# * 'RANLIB' - 'ranlib' archive index tool.
RANLIB=${TOP_BIN}/${erl_xcomp_host}-ranlib
```

```
# * 'AR' - 'ar' archiving tool.
AR=${TOP_BIN}/${erl_xcomp_host}-ar

# * 'GETCONF' - 'getconf' system configuration inspection tool. 'getconf' is
#   currently used for finding out large file support flags to use, and
#   on Linux systems for finding out if we have an NPTL thread library or
#   not.
#GETCONF=

## -- Cross System Root Locations --------------------------------------------

# * 'erl_xcomp_sysroot' - The absolute path to the system root of the cross
#   compilation environment. Currently, the 'crypto', 'odbc', 'ssh' and
#   'ssl' applications need the system root. These applications will be
#   skipped if the system root has not been set. The system root might be
#   needed for other things too. If this is the case and the system root
#   has not been set, 'configure' will fail and request you to set it.
#erl_xcomp_sysroot=

# * 'erl_xcomp_isysroot' - The absolute path to the system root for includes
#   of the cross compilation environment. If not set, this value defaults
#   to '$erl_xcomp_sysroot', i.e., only set this value if the include system
#   root path is not the same as the system root path.
#erl_xcomp_isysroot=

## -- Optional Feature, and Bug Tests -----------------------------------------

## These tests cannot (always) be done automatically when cross compiling. You
## usually do not need to set these variables. Only set these if you really
## know what you are doing.

## Note that some of these values will override results of tests performed
## by 'configure', and some will not be used until 'configure' is sure that
## it cannot figure the result out.

## The 'configure' script will issue a warning when a default value is used.
## When a variable has been set, no warning will be issued.

# * 'erl_xcomp_after_morecore_hook' - 'yes|no'. Defaults to 'no'. If 'yes',
#   the target system must have a working '__after_morecore_hook' that can be
#   used for tracking used 'malloc()' implementations core memory usage.
#   This is currently only used by unsupported features.
#erl_xcomp_after_morecore_hook=

# * 'erl_xcomp_bigendian' - 'yes|no'. No default. If 'yes', the target system
#   must be big endian. If 'no', little endian. This can often be
#   automatically detected, but not always. If not automatically detected,
#   'configure' will fail unless this variable is set. Since no default
#   value is used, 'configure' will try to figure this out automatically.
#erl_xcomp_bigendian=

# * 'erl_xcomp_double_middle' - 'yes|no'. No default. If 'yes', the
#   target system must have doubles in "middle-endian" format. If
#   'no', it has "regular" endianness. This can often be automatically
#   detected, but not always. If not automatically detected,
#   'configure' will fail unless this variable is set. Since no
#   default value is used, 'configure' will try to figure this out
#   automatically.
#erl_xcomp_double_middle_endian

# * 'erl_xcomp_clock_gettime_cpu_time' - 'yes|no'. Defaults to 'no'. If 'yes',
#   the target system must have a working 'clock_gettime()' implementation
#   that can be used for retrieving process CPU time.
#erl_xcomp_clock_gettime_cpu_time=
```

```
# * 'erl_xcomp_getaddrinfo' - 'yes|no'. Defaults to 'no'. If 'yes', the target
#   system must have a working 'getaddrinfo()' implementation that can
#   handle both IPv4 and IPv6.
#erl_xcomp_getaddrinfo=

# * 'erl_xcomp_gethrvtime_procfs_ioctl' - 'yes|no'. Defaults to 'no'. If 'yes',
#   the target system must have a working 'gethrvtime()' implementation and
#   is used with procfs 'ioctl()'.
#erl_xcomp_gethrvtime_procfs_ioctl=

# * 'erl_xcomp_dlsym_brk_wrappers' - 'yes|no'. Defaults to 'no'. If 'yes', the
#   target system must have a working 'dlsym(RTLD_NEXT, <S>)' implementation
#   that can be used on 'brk' and 'sbrk' symbols used by the 'malloc()'
#   implementation in use, and by this track the 'malloc()' implementations
#   core memory usage. This is currently only used by unsupported features.
#erl_xcomp_dlsym_brk_wrappers=

# * 'erl_xcomp_kqueue' - 'yes|no'. Defaults to 'no'. If 'yes', the target
#   system must have a working 'kqueue()' implementation that returns a file
#   descriptor which can be used by 'poll()' and/or 'select()'. If 'no' and
#   the target system has not got 'epoll()' or '/dev/poll', the kernel-poll
#   feature will be disabled.
#erl_xcomp_kqueue=

# * 'erl_xcomp_linux_clock_gettime_correction' - 'yes|no'. Defaults to 'yes' on
#   Linux; otherwise, 'no'. If 'yes', 'clock_gettime(CLOCK_MONOTONIC, _)' on
#   the target system must work. This variable is recommended to be set to
#   'no' on Linux systems with kernel versions less than 2.6.
#erl_xcomp_linux_clock_gettime_correction=

# * 'erl_xcomp_linux_nptl' - 'yes|no'. Defaults to 'yes' on Linux; otherwise,
#   'no'. If 'yes', the target system must have NPTL (Native POSIX Thread
#   Library). Older Linux systems have LinuxThreads instead of NPTL (Linux
#   kernel versions typically less than 2.6).
#erl_xcomp_linux_nptl=

# * 'erl_xcomp_linux_usable_sigaltstack' - 'yes|no'. Defaults to 'yes' on Linux;
#   otherwise, 'no'. If 'yes', 'sigaltstack()' must be usable on the target
#   system. 'sigaltstack()' on Linux kernel versions less than 2.4 are
#   broken.
#erl_xcomp_linux_usable_sigaltstack=

# * 'erl_xcomp_linux_usable_sigusrx' - 'yes|no'. Defaults to 'yes'. If 'yes',
#   the 'SIGUSR1' and 'SIGUSR2' signals must be usable by the ERTS. Old
#   LinuxThreads thread libraries (Linux kernel versions typically less than
#   2.2) used these signals and made them unusable by the ERTS.
#erl_xcomp_linux_usable_sigusrx=

# * 'erl_xcomp_poll' - 'yes|no'. Defaults to 'no' on Darwin/MacOSX; otherwise,
#   'yes'. If 'yes', the target system must have a working 'poll()'
#   implementation that also can handle devices. If 'no', 'select()' will be
#   used instead of 'poll()'.
#erl_xcomp_poll=

# * 'erl_xcomp_putenv_copy' - 'yes|no'. Defaults to 'no'. If 'yes', the target
#   system must have a 'putenv()' implementation that stores a copy of the
#   key/value pair.
#erl_xcomp_putenv_copy=

# * 'erl_xcomp_reliable_fpe' - 'yes|no'. Defaults to 'no'. If 'yes', the target
#   system must have reliable floating point exceptions.
#erl_xcomp_reliable_fpe=

## ------------------------------------------------------------------------
```