IT 11 036
Examensarbete 30 hp
Juni 2011

Development of an Erlang System Adapted to Embedded Devices

Fredrik Andersson
Fabian Bergström

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Development of an Erlang System Adapted to Embedded Devices

Fredrik Andersson, Fabian Bergström

Erlang is a powerful and robust language for writing massively parallel and distributed applications. With the introduction of multi-core ARM processors, the embedded market will be looking for ways of taking advantage of the newfound opportunities for parallelism.

To support the development of embedded applications using Erlang we want to provide Erlang and Embedded developers with a run-time system suited for embedded devices. We have managed to shrink the disk size of the Erlang runtime system by a factor of 42 without affecting the stability of the Erlang virtual machine.

It is our firm belief that Erlang can play a vital role in the future of embedded systems.

Handledare: Jan Nyström
Ämnesgranskare: Mikael Pettersson
Examinator: Anders Jansson
IT 11 036
Sponsor: Erlang Solutions

Tryckt av: Reprocentralen ITC

Contents

Acknowledgements

Erlang Solutions For giving us the opportunity to do this thesis work and supplying us with hardware and expert domain knowledge.

Henry Nyström For helping out with the initial planning, guiding us towards our goal, and answering questions that arose during our work.

The OTP team For answering technical questions and pointing us to the interesting parts of the Erlang documentation.

Mikael Pettersson For guidance and help with setting goals.

Ulf Wiger For Erlang expertise and introducing us to bstone.

Henrik Nordh & Gustav Simonsson For general help regarding Linux configuration on the devices.

1 Introduction

1.1 Motivation

With the upcoming release [ARM 2009] of dual core ARM processors many embedded systems will see a potential boost in performance which is easier to make use of in an Erlang environment.

Software development for embedded systems is still done in low level programming languages with unmanaged memory, such as C and C++, for performance reasons. Some parts of these systems are not performance critical, and could be written in more high level languages.

We believe Erlang would also be a good fit for parts of embedded systems that have to communicate with other computers/CPU's, due to the distributed nature of Erlang. A high level language could also make maintenance easier and development time shorter.

When creating an Erlang release, you specify which applications it uses and the release tools will package the needed applications only. Unused libraries will not be included so the release will be kept somewhat small. Other high level languages often have a modified virtual machine e.g. Java, which has embedded versions, that is not bytecode compatible with the standard virtual machine, and a limited number of libraries available to the developer.

Erlang/OTP has been shown to be well suited for application scenarios which are similar to the ones in the telecommunication space [Mattsson et al. 1999, Arts et al. 2004, Armstrong 1996, Däcker 2000, Armstrong 2002, Blau et al. 1999]. It has been developed with requirements in mind which do not necessarily apply to embedded systems with limited resources [Armstrong 2007]. Requirements such as concurrency, garbage collection, networking. These may lead to an increased memory usage and more work to do for the processor.

It would therefore be very beneficial to have an Erlang distribution adapted to embedded systems which can better utilize the limited resources available.

1.2 Embedded systems

"Embedded system" can mean many different things, in this paper when we mention embedded systems we mean embedded systems with resources more limited than PCs or server computers. Processors with clock frequencies of a few hundred MHz, a few hundred megabytes or less of primary memory.

Among the daily uses of embedded systems are control units inside of electronics, entertainment systems in cars, mobile phone processors and such.

Our focus in this thesis has been computers on the more powerful side

of the embedded spectrum, such as the BeagleBoard[1] and Gumstix[2]. One could imagine these being integrated into cars, routers or any other area where a more powerful device is needed.

1.3 Erlang

Erlang is a general-purpose programming language and runtime environment developed by Ericsson. It has built-in support for concurrency, distribution and fault tolerance[Armstrong 2002]. In 1998 Ericsson decided to release Erlang as open source and today it is available online[3].

1.4 OTP

OTP (Open Telecom Platform) is a large collection of libraries for Erlang to do everything from compiling ASN.1[4] to providing an HTTP server. It supports supervisor structures making it easier to write fault tolerant systems. In addition a design pattern called behaviours is also supported. Behaviours generalise common tasks in an Erlang system and are used by supplying a number of callback functions that implements the specific parts of the task, for example `gen_server` that implements a server interface.

As is the case with other high level languages such as Java or C#, Erlang is most often used in conjunction with its platform libraries.

1.5 BeagleBoard

The BeagleBoard is a small, open source computer, meant to be extended by the community. It is purely a development board by Texas Instruments, meant to give quick access to the OMAP (Open Multimedia Applications Platform^{TM}) [Chaoui 2001] system-on-chip for developers. Because it is meant to be extended, it has few integrated peripherals and several standard interfaces. It has hardware for OpenGL and Digital Signal Processing. The available interfaces are DVI-D and S-Video ports for video output, there are standard 3.5 mm audio in and out jacks. The BeagleBoard also has USB ports and an RS232 serial port [Electronics Industries Association]. For more information see the BeagleBoard reference manual[bea 2009] and the schematics[BeagleBoard.org 2010].

The BeagleBoard has been used in e.g. teaching[Hofmann 2009] and for wearable computers for rescue dogs[J. Tran 2010].

[1] `www.beagleboard.org`

[2] `www.gumstix.net`

[3] `http://www.erlang.org`

[4] Abstract Syntax Notation One, a formal notation that allows specifications of information handled by high-level Telecom protocols with no loss of generality, regardless of software or hardware systems [Dubuisson 2001]

1.6 Gumstix

Gumstix are commercial, much smaller computers, available in several flavours and with different expansion boards to provide various interfaces. The Gumstix Overo series is built on the same OMAP platform as the BeagleBoard, meaning they are also available with OpenGL and DSP support. There are also models with WiFi and Bluetooth hardware.

Gumstix have been used e.g. for controlling a swarm of miniature helicopters [Nardi and Holland 2006] and in wireless sensor networks Hughes et al. [2006].

2 Problem Description

We believe embedded systems development could be made simpler and faster, and more easily take advantage of the new multicore embedded processors by using Erlang instead of low level languages such as C. Erlang programs however, because of the Erlang's high level nature, are often less performant than C programs, and use more memory.

Even though some parts of an embedded system might not be performance critical, we still want to make sure not too much memory or disk space is used. The systems might be up for long periods of time and we don't want them to fail from running out of memory. Therefore, we will investigate the extent to which memory and disk space can be saved when making an Erlang release.

We will create an Erlang release tailored for embedded systems, cross compiled, and set up to use less memory and disk space.

2.1 Deliverables

A minimal Erlang system containing the virtual machine and some standard library modules will be made available as a package that can be installed on an embedded system. Furthermore, the differences between the original distribution and the adapted version will be described. Any drivers needed for accessing hardware specific to the embedded systems should be provided as linked in drivers.

We would also like to develop a tool to allow embedded Erlang developers to better minimize their memory usage. We believe that it should be possible to remove unused modules and functions from the application release to better fit the small memory requirements of an embedded system.

2.2 Scope

The release will be adapted to the BeagleBoard and Gumstix platforms. It will include, at a minimum, the Erlang virtual machine with its runtime system, and the standard libraries `stdlib` and `kernel`. None of the other OTP libraries will be adapted or packaged with the runtime system. We will configure the system according to our results from examining memory and disk space usage.

3 Methodology

3.1 Approach

Erlang/OTP will be stripped down to just the virtual machine with runtime system and a small set of libraries deemed necessary. The virtual machine must be adapted to work on the limited resources of embedded systems, drivers for any devices not readily available for Erlang must be produced. The system should run on a Linux distribution.

3.2 Reducing disk usage

An out of the box installation of Erlang/OTP is 118 megabytes by default. When deploying an Erlang system, one usually removes the source code, documentation, and all libraries except the ones needed. If you remove all the source code and documentation, and only keep `stdlib` and `kernel`, the release is around 6.3 megabytes.

The Erlang application `reltool` can be used to create a release, it has experimental support for deriving which applications and modules are used and include only those when building the release. Not including unused applications and modules will save disk space. See the `reltool` reference manual for instructions on how to use it [Ericsson 2010a]. We have chosen to include all modules of the `stdlib` and `kernel` applications for our comparisons.

In addition to removing applications and modules, there are other things one can do to save space. GCC has an optimization level that optimizes for smaller size of the object files. The files produced from the C code can further be made smaller by stripping all symbol information with the `strip` program.

OTP has a feature similar to the `strip` program, in the module `beam_lib`, there is a function called `strip_release`. `strip_release` takes as parameter the path to the root of an Erlang release, and removes all chunks except those needed by the loader from the BEAM files.

The Erlang runtime system also supports loading compressed modules or compressed applications. The compression format in this case is zip. To create a compressed BEAM file, one can use a compression program to create a zip file and change the file ending to `ez`, the filename before the ending must still be the module name. Another way to create compressed modules is to pass the flag `compressed` to the Erlang compiler, and it will compress the modules automatically[5].

In order to save on the disk usage by Erlang modules, one could create a program to statically inspect the code for a release and removing unused functions before compiling, much like `reltool` can create a smaller release

[5] BEAM files compressed by the Erlang compiler will have the regular `beam` file ending

by not including unused modules. We chose not to focus on this in order to have more time to do measurements, and because we think the savings would not be great since compiled Erlang bytecode is generally small[6]. One would also need to maintain some sort of white list of functions that may be called dynamically since discovering them statically might be impossible.

Our expectation is that compressing applications and modules will have the greatest impact on performance, due to the fact that the code loader will have to uncompress them to load the code. Stripping binaries and BEAM files should mostly affect the disk size because it just removes parts of the binary files that we don't use. Compressing modules files that have been stripped will probably not have as much effect as compressing modules that have not been stripped, since bytecode may not lend itself to compression as well as the textual meta-data will. Since we only include the `stdlib` and `kernel` applications, compressing modules or applications might not make as much difference as they would if more applications are used in a release. We also assume that compiling with the different optimizations for GCC should give some differences, i.e. that the `os` option will decrease the size of the emulator binary because that is its purpose, and that `o3` will increase the size of the binary because it's designed to produce better performing programs, instead of smaller ones.

3.3 Reducing memory usage

3.3.1 Memory allocation

Erlang is garbage collected and memory of different types are allocated in different memory areas. Several different allocators in the runtime system are responsible for one type of memory each (for example, `binary_alloc` allocates binaries). When starting the Erlang runtime, one can choose to deactivate certain allocators and let the default allocator (`sys_alloc` an malloc based allocator) handle their data instead. The different allocators also have different settings for e.g. how much memory to allocate every time they need more. These settings can also be set when starting the emulator.

Besides manually disabling individual allocators, there are some pre-configured allocator profiles available to choose from as well. These allocator profiles are chosen by giving the flag `+Mea` when starting the runtime. The flag takes the following options: `min`, `max`, `r9c`, `r10b`, `r11b`, `config`. `min` will disable all the allocators except for `fix_alloc`, `sys_alloc` and `mseg_alloc`. `max` enables all allocators and is the default setting as of now. `r9c`, `r10b`, `r11b` configure the allocators to the default settings of the R9C, R10B, and R11B releases respectively. `config` disables all the features that have to be disabled when creating an allocator configuration file using `erts_alloc_config`.

[6]The average size of the compressed and stripped modules in the applications erts, stdlib, and kernel is 8 kilobytes

For more indepth description of the different allocators see the `erts_alloc` documentation.

`erts_alloc_config` is an Erlang module that can be used to record the memory usage of a running Erlang program and produce a file of command line options for the emulator that sets up the allocators to fit the memory usage in the program [Ericsson 2010a].

When building OTP, one can define the variable `SMALL_MEMORY`, and the C preprocessor will disable all allocators so the runtime system always behaves as if `+Mea min` was passed on startup. Some constants affecting the size of memory areas in the allocator will also be smaller. `SMALL_MEMORY` also makes Erlang processes more prone to garbage collection.

For our comparison, we will only use the default allocator settings, the minimal profile, and a profile created with `erts_alloc_config`, this is to limit the combinatorial explosion of that would result from comparing releases with all the different allocator settings combinations.

Our expectation is that using the default memory allocator profile will lead to the most memory usage, that the minimal profile should save some memory because of less overhead when each type of allocator keeps a memory area of it own, and that using `erts_alloc_config` should lead to more memory saved due to better tailoring the allocators for the benchmark program.

3.3.2 Garbage collection

Erlang has a generational garbage collector which separates old and new memory. The reasoning behind this is that garbage collection only needs to be done on the new part of the memory frequently and less frequent on the old part because it is exptected that younger data can be reclaimed more often. The new part of the memory is often small and a full stop is going to be short since the garbage collector in general only will look at the new part. When not enough memory is found in the new part it will perform a full sweep which includes the old part of the memory as well.

The garbage collector in Erlang has a setting that lets you tell it how often to do a full sweep collecting garbage. This setting is controlled by setting the environment variable `ERL_FULLSWEEP_AFTER` before starting the virtual machine. The variable is an integer representing the number of generations you want to pass between each full sweep.

We will examine how this variable affects the memory usage and performance by running our benchmark with the variable set to 0, which will trigger a full sweep at every generation, and not setting it, using the default setting[7]. Comparing other settings could be interesting as well, but we want to limit the amount of releases to benchmark.

[7]For a description of how the garbage collector works see the Erlang FAQ and the answer to "How does the Garbage Collector work?"

Our expectation is that by doing full sweeps every generation, less memory would be used by freeing more often, but performance will decrease due to the garbage collector overhead when traversing the graph of values to see which ones can be freed.

3.4 Evaluation

We will compare the disk usage of releases with the different combinations of compiler options, stripping, and compressing that we discussed above.

A demo application to exercise the virtual machine and included libraries will use a simple protocol to communicate between the Gumstix and BeagleBoard computers over all suitable interfaces. In addition to this the release must still pass the whole OTP test suit before packaging, to ensure that we have not affected the stability of the runtime environment with different compiler options.

The memory usage of the different releases will be measured using a benchmark program. To ensure that saving memory does not significantly reduce the performance, we will also compare the performance metrics from the benchmark program's reports.

3.4.1 Bstone

We will perform our benchmarks using a program called `bstone`, available from the Erlang download page. `bstone` is meant to be representative of a fairly typical Erlang program, it was used at Ericsson for evaluating new hardware to use in products [Wiger 2010]. The benchmark runs a number of iterations and calculates a score depending on how fast the iterations are. We modified the program to change the reporting and added a process to monitor the memory usage of the Erlang emulator.

3.4.2 Benchmark configurations

Considering the number of different allocators, and the amount of options available for them, the combinations of different settings affecting the memory usage are many. We have chosen to compare just a few of these combinations. The allocator settings we will compare are the default profile, the minimal profile, and one which was set up using `erts_alloc_config` while running the benchmark program[8]. We will compare these allocator profiles using the default garbage collector settings, and doing a full sweep every garbage collection cycle. For the benchmarks, we will use three releases with the three different optimisation settings for GCC, as well as five different releases all built with `-o2` but different settings for compression and stripping of the module files.

[8]We ran `bstone` with calls to `erts_alloc_config` for the same duration as our benchmarks will run to gather statistics to be able to generate a configuration for the allocators

Furthermore we will run bstone for 10 minutes for each configuration we are testing.

4 Related Work

4.1 Erlang's official support for embedded devices

The embedded systems mentioned in this paragraph does not fit the description of embedded systems we use in this document. The systems mentioned here performs much better than the small ones we have focused on.

Although Ericsson has used Erlang in embedded systems, more specifically the VxWorks based Switchboard[Ericsson 2010a], Erlang is not tailored to embedded systems outside of Ericsson's needs. The Embedded Systems User's Guide [Ericsson 2010a] concerns only embedded Solaris, windows NT and VxWorks and instructs only on how to setup an embedded system, not how to tailor the virtual machine or the application to lower the ram and disk usage.

4.2 Stand Alone Erlang

Joe Armstrong has developed a version of Erlang called Stand Alone Erlang[9] which is made to have a smaller disk footprint than the ordinary Erlang release. Stand Alone Erlang is based on Erlang R9, which was released 2002. We have not compared our work with Armstrong's Stand Alone Erlang because we did not make any changes to the virtual machine and the difference between R13B04 and R9 are quite substantial (multicore support was implemented in R11B[Ericsson 2006] for example).

[9] `http://www.sics.se/~joe/sae.html`

5 Results

5.1 Reducing disk usage.

With the different stripping and compression settings, and compiler optimization options, we managed to shrink the release to 2.8 Megabytes. Our release consists of the Erlang runtime system and the libraries `kernel` and `stdlib`.

Stripping and compressing Erlang modules will affect the `lib` subdirectory in a release, since that's where the Erlang code is. The different GCC optimization options and stripping binaries will affect the `erts` subdirectory in a release, since that is where the binaries for the virtual machine and runtime system are.

For the above reasons, we present the results for disk usage in two tables, one for settings changing the size of the runtime system, and one for settings changing the size of the Erlang libraries.

Table 1 shows the size of the runtime system. The column "O-level" shows the optimization level used when building the binaries with GCC. "Stripped" shows whether a release has had its binaries stripped by the `strip` program. "Size" presents the size in kilobytes of the `erts` subdirectory.

Table 2 shows the size of the Erlang libraries. The column "Compressed applications" shows if the applications in the release have been compressed with `zip` into `ez` archives. "Compressed modules" shows whether the Erlang modules in the release were compiled with the `compressed` flag. "Stripped" shows if the BEAM files in the release were stripped using `strip_release` in `beam_lib`. "Size" presents the size in kilobytes of the `lib` subdirectory.

O-level	Stripped	Size (kb)
2	No	2068
3	No	2092
s	No	1828
2	Yes	1792
3	Yes	1820
s	Yes	1544

Table 1: Disk usage of `erts`

Compressed applications	Compressed modules	Stripped	Size (kb)
No	No	No	2296
No	No	Yes	1284
No	Yes	No	1524
No	Yes	Yes	1284
Yes	No	No	1172
Yes	No	Yes	972
Yes	Yes	No	1172
Yes	Yes	Yes	972

Table 2: Disk usage of `lib`

The release with the smallest disk usage is thus achieved by building with the `-os` flag for GCC, stripping the binaries, stripping the BEAM files, and compressing the Erlang applications into `ez` archives. With these settings, the release is 2796 kilobytes in total.

One interesting thing to note is that the size differences between `o3` and `o2` compilations are not that great, while `os` is much smaller.

Compressing BEAM files does not change the size at all when coupled with stripping the BEAM files or compressing whole applications. When compressing applications, it's not surprising that compressing modules individually first does not save more space, data that is already compressed cannot be made smaller by applying the same compression again. One could have imagined the combination of stripping and compressing BEAM files could save more space than just stripping them. Perhaps the bytecode left in modules after stripping does not compress well.

5.2 Reducing primary memory usage.

The diagram below shows the results of our memory usage benchmarks.

In the diagram, the Y-axis describes the allocator and garbage collection configuration used, "minimal" and "default" means the minimal or default allocator configuration was used. "config" means the allocator settings recommended by `erts_alloc_config` were used. "fson" and "fsoff" means that full sweep every generation was on or off, respectively. The X-axis shows the minimum, maximum, and mean amount of memory that was in use every second by the virtual machine.

The reasons for the big difference between min and max usage in the diagram is that our modified Bstone benchmark does a series of tests. Some are larger and some are smaller in size.

From the diagram we can see that the minimal allocator configuration uses significantly less memory.

More detailed graphs can be seen in A.

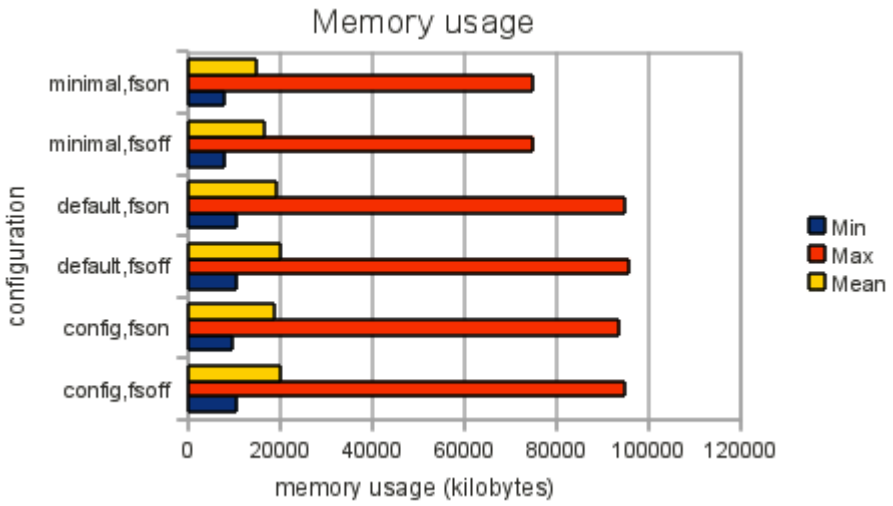


Figure 1: Memory usage with different allocators and GC settings

5.3 Performance implications

We also measured the performance of our different releases, to see how it was affected by the different memory saving methods.

The diagrams below show the results of our performance benchmarks for the releases built with different ways of saving memory on the BEAM files and binaries from GCC.

In the diagrams, the Y-axis describes the allocator configuration used, "minimal" and "default" means the minimal or default allocator configuration was used. "config" means the allocator settings recommended by `erts_alloc_config` were used. "fson" and "fsoff" means that full sweep every generation was on or off, respectively. The X-axis shows the minimum, maximum, and mean amount of "stones" that was attained for the benchmark iterations of a given configuration.

More detailed graphs can be seen in A.

The different allocator configurations do not affect performance that much, but the `default, fsoff` configuration performs the best, while one could have expected that `config, fsoff` would have been better. This indicates that the default configuration is well suited for bstone. In most cases `config, fsoff` is coming in at second place followed by the `minimal, fsoff` at third.

Unsurprisingly, turning on full sweep for every generation performs worse that the default garbage collection scheme, although not by a lot.

The release built with `-os` performs significantly worse than the others.

Comparing these performance results with the memory usage results, one can see that using the minimal allocator profile does not come with a

significant performance penalty, even though it greatly saves on the memory usage.

While building the runtime system with `-os` can save disk usage, it has a significant performance cost that should be considered.

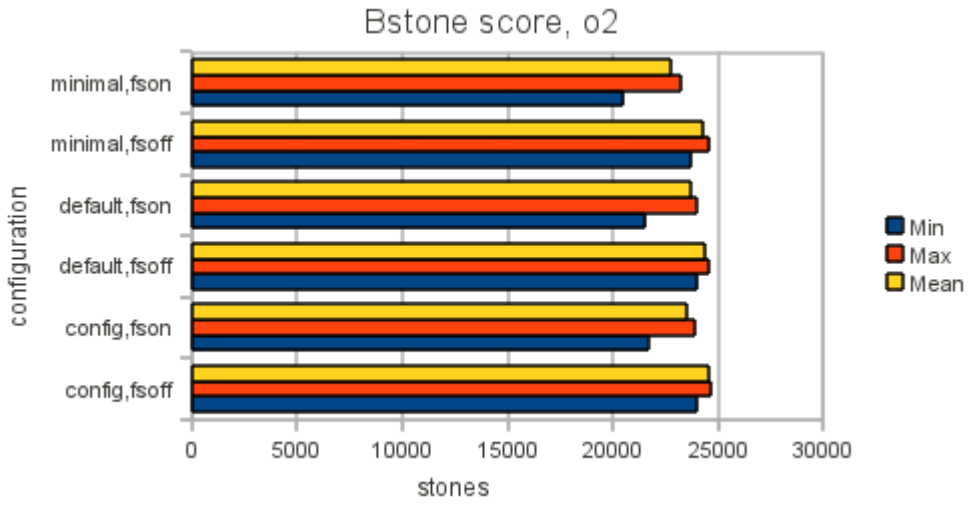


Figure 2: `-o2`, no compression or stripping

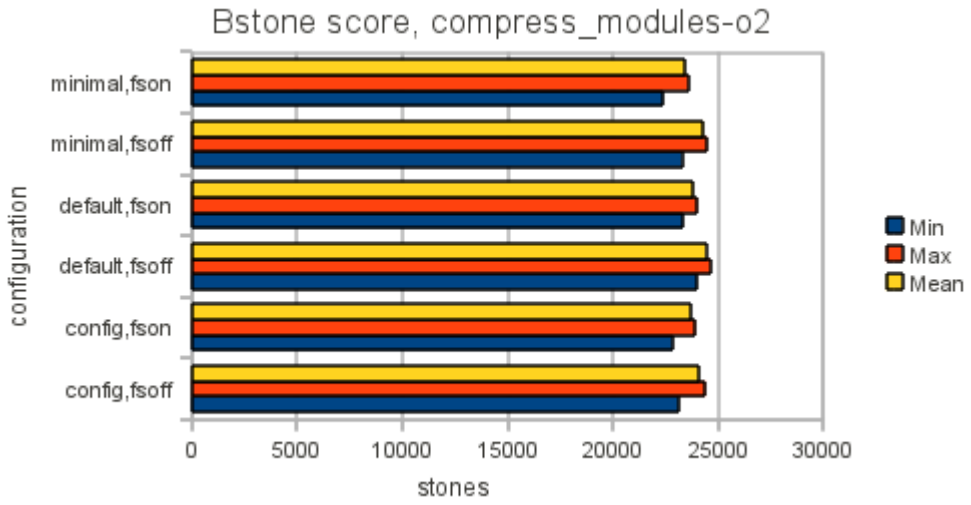


Figure 3: `-o2`, Compressed BEAM files

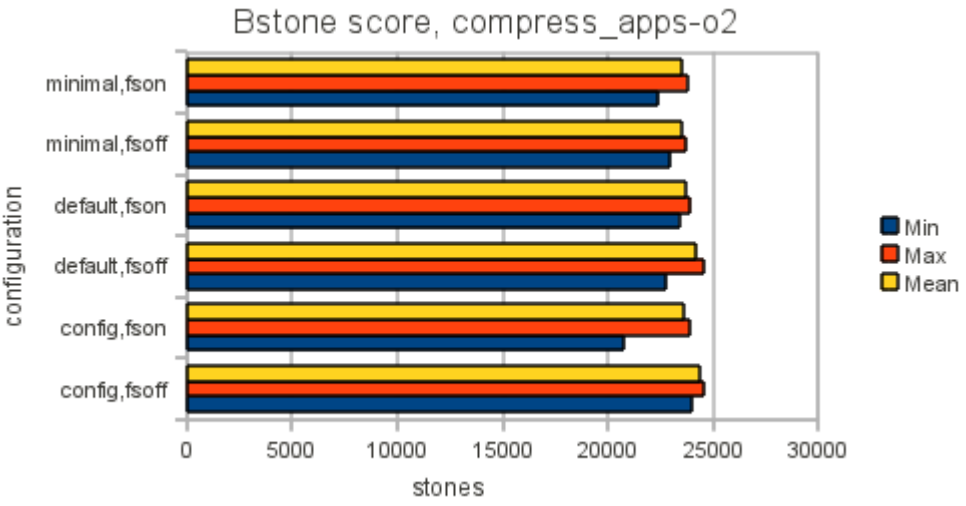


Figure 4: `-o2`, Compressed applications

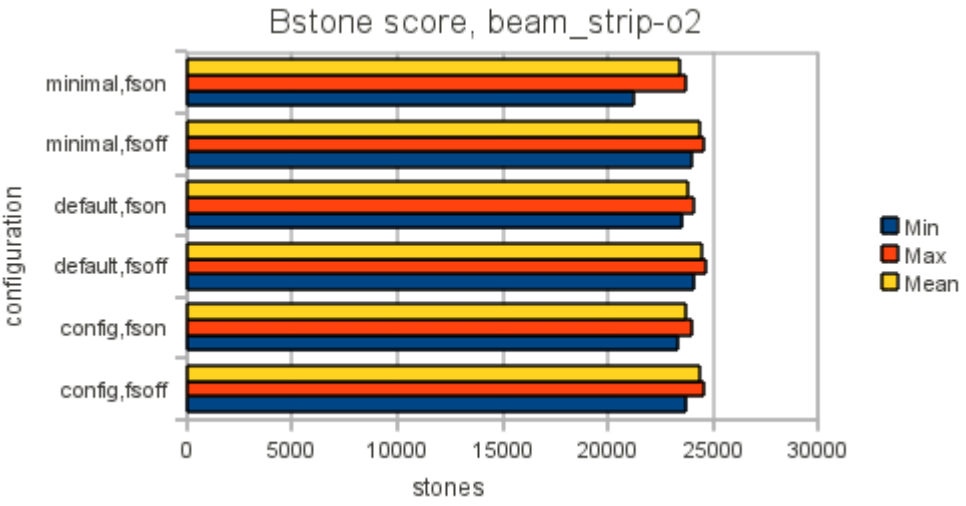


Figure 5: `-o2`, Stripped BEAM files

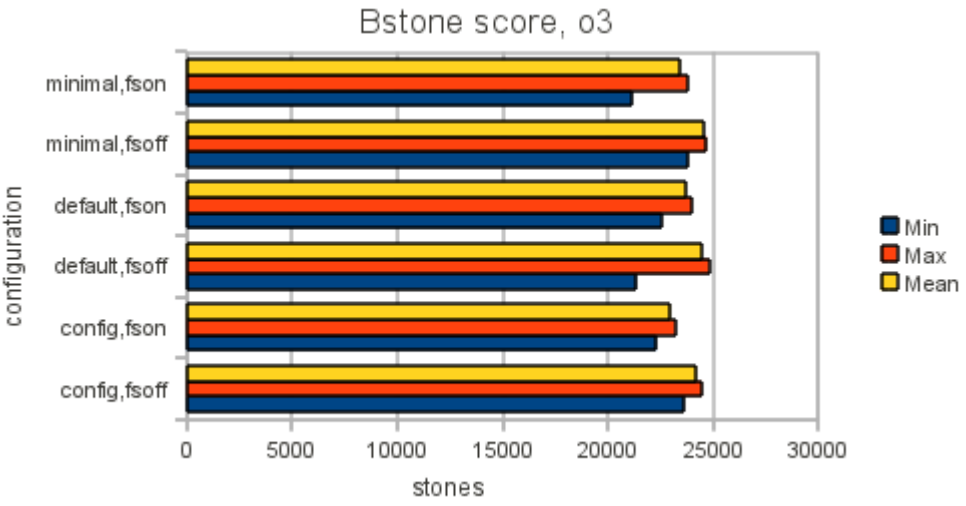


Figure 6: `-o3`, no compression or stripping

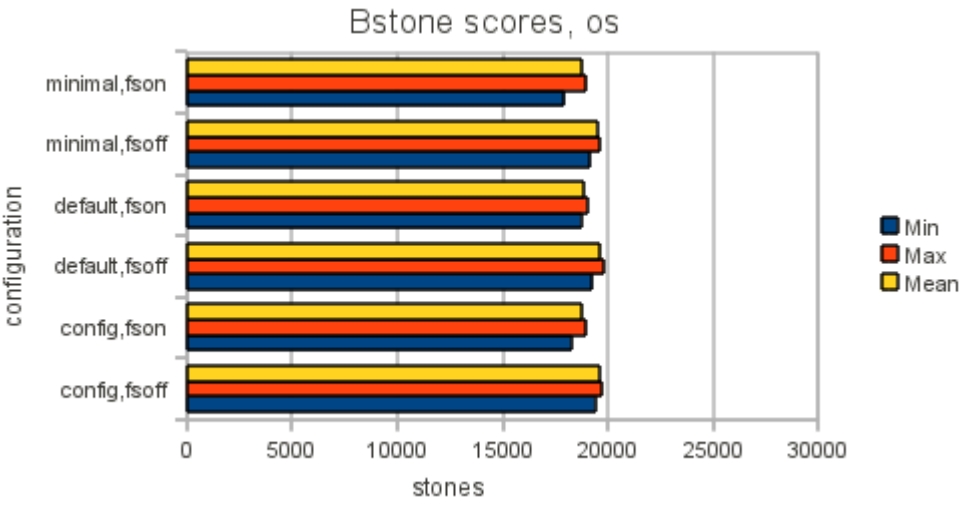


Figure 7: `-os`, no compression or stripping

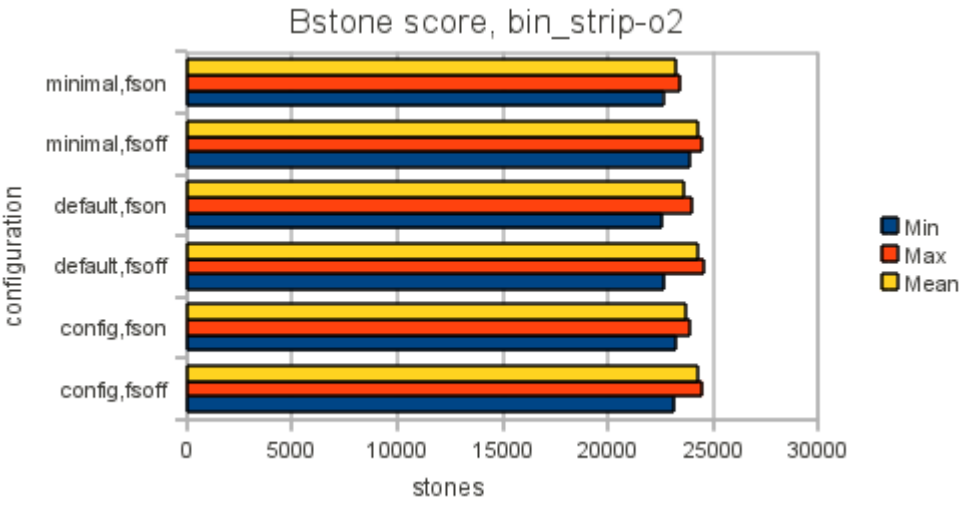


Figure 8: `-o2`, stripped binaries from GCC

5.4 The demo application

The serial port was the only interface that Erlang did not have native support for and because of this it is the only interface we have used in the demo application.

We added support for a serial communication library called erlang-serial[10], by Tony Garnock-Jones based on work by Johan Bevemyr, which allows us to communicate via the RS-232 serial port. On top of this serial library, we wrote a program that reads input from a button on the BeagleBoard and signals over the serial line to another BeagleBoard running the same program. The signal results in an LED on the receiving BeagleBoard lighting up.

We were unable to test the demo application between the Gumstix and the BeagleBoard because of missing hardware. We did not have the parts needed to use the serial ports on the Gumstix.

5.5 Development tools

Thanks to the built in support for cross compiling Erlang releases[Ericsson 2010b] in the Erlang/OTP source we created a cross compiled release for the arm platform in a simple way. We used the OpenEmbedded[11] toolchain for compiling erlang.

[10] `http://github.com/tonyg/erlang-serial`

[11] `http://www.openembedded.org/`, a framework for -building Linux distributions for embedded devices

5.6 The OTP test suite

We successfully passed the whole test suite except for the tests that try to manipulate file permissions and those who where designed to consume a lot of memory. We failed to pass the file permission tests since the file system being used on the SD cards (FAT 32) did not support changing file permissions and since we did not have a lot of memory available on the boards we naturally failed the tests trying to allocate allot of memory.

6 Conclusion and discussion

We managed to cut down on the disk usage for our Erlang release quite significantly by letting the C compiler optimize binaries for size, stripping the binaries, stripping the Erlang modules and compressing Erlang applications. More space could be saved if one knows what parts of the libraries will be used and remove unused code, but the Erlang modules are not very big to begin with.

The results from our attempts to decrease the primary memory usage were less interesting. This could be an indication that the current virtual machine implementation is quite memory efficient. The language is almost 25 years old, and has gone through several virtual machine iterations. Another likely reason for this might be that memory usage is extremely application dependent or that Bstone is not a great benchmark tool for memory usage.

Getting Erlang up and running on the boards was easier than what we expected it to be. Erlang was portable enough to run on arm without any changes to the source code. Making it small and usable is another topic that can be improved further. One would have to take a dive down into the virtual machine source code for this to happen and that has been considered to be out of scope for this project.

The fact that the virtual machine has not changed from the standard implementation enables a embedded device to be deployed as a part of a larger system, maybe performing one specific task such as monitoring and diagnostics. The distributed nature of Erlang makes a design like this transparent.

6.1 Challenges

6.1.1 The current state of the OTP documentation

While the documentation of Erlang/OTP is very good it is mostly good on a module level basis. You need to know what to look for to be able to find it and that is not always the case. This is one of the reason why sites such as `http://www.erldocs.com/` exists.

6.2 Known issues

6.3 No linked in driver for serial port communication

A linked in driver was never developed for the serial port communication. Instead the serial port can be accessed through the `erlang-serial` library, which uses a port program to communicate with the serial port.

7 Future Work

7.1 Improving performance

While Erlang is not the language best suited for high performance applications we still want to enable it to work as fast as possible for the tasks that are given, for example being able to process as many messages per second as possible. In the following sections we will try to map out areas in which we think that the embedded Erlang release could be improved.

7.1.1 Hipe

Hipe is a native compiler for Erlang code. It is a part of the official OTP release since R9B and documentation is supplied with OTP. Hipe allows a function to be compiled to native machine code which greatly improves the performance of that code [Johansson et al. 2002]. Since Hipe has support for the ARM processor a possible approach would be to allow Erlang to automatically Hipe compile (native compile) the most suitable code in a users application. It would be interesting to explore the performance impact of Hipe as well as the changes to memory and disk usage.

7.1.2 Allow for automatic function removal

Since OTP already has support for automatic module removing it would be beneficial to be able to exclude functions not needed in the system, thereby saving both primary and secondary memory. This could be done by processing beam files and stripping them of the unneeded functions. This would have to be done in conjunction with the developer since it can be impossible to derive dynamic function calls. A function white list would be one way of implementing this, in order to let the developer to tell the system which functions will be used so that the rest can be removed.

7.1.3 Strip the erlang virtual machine of features

It would be nice to be able to be able to deactivate features of the erlang virtual machine to cut down on memory consumption depending on the needs of the application. For example removing ETS tables completley or removing file system drivers that it does not use.

7.1.4 Explore the upcoming support for compressing ETS tables

With the recent release of OTP R14B01 support has been added for compressing ETS tables [Ericsson 2010c], this will save memory in exchange for a performance hit when decompressing and could prove useful on embedded systems where the memory is limited. The Erlang runtime system uses ETS tables internally so there could be great benefits for doing this.

7.2 virtual machine tweaks for embedded systems

We have only looked at the possibilities available at build time and runtime, but it would be good to also explore the possibility of changing the Erlang virtual machine implementation to better suit embedded systems. Different parts of the virtual machine could probably be rewritten to consume less memory, for example changing the optimizations done on BEAM loading towards low memory consumption rather than performance optimizations or using smaller data structures. Another possibility would be to extend the virtual machine with support for special purpose hardware should increase the performance of it[Ferm 2011].

7.3 Better platform support

It would be good to support other kinds of open embedded systems e.g. Linksys routers, Arduino boards or the Pandora gaming platform to mention a few of them. Recently an Android port of Erlang was released[Axelsson 2011], and bringing our results to the Android port could prove interesting as well.

7.4 Continuous integration with new Erlang releases

As of now we have done much of our work by hand and it would be good to automatically build, test and deploy new releases of the Erlang virtual machine as they are released by the OTP team. It might even be beneficial to automatically create the benchmarks for each build to allow for comparison when a user tries to decide which version to choose. A natural choice for us would be to integrate the embedded Erlang build system with Swarm[Breddin 2010] developed by Erlang Solutions.

7.5 Deeper integration into Erlang/OTP

It would be good to have a system more integrated than the script we have now. The current build system for Erlang/OTP could be extended to also support building an embedded release.

7.6 Test with mmap enabled

The memory management in Erlang is built on top of the C standard libraries `malloc` for portability reasons. There is, however, support in the build system for using a memory manager bypassing `malloc` and instead using the `mmap` system call available in e.g. Linux. Since this memmory manager has been implemented by the OTP team using mmap instead of malloc it would be interesting to see if it is faster since it knows more of how Erlang data is used.

7.7 Compile individual parts of the virtual machine with different GCC optimizations

Most likely only a small part of the virtual machine (the emulator and the garbage collector) benefits from heavy optimization and the other parts of it can be compiled with size optimization to save more disk space while still benefiting from the optimization.

7.8 Direct threading evaluation

The Erlang virtual machine has support for direct threading. Exploring how memory and performance would be affected by this feature could be interesting.

A Graphs

A.1 Performance and memory usage graphs

Below are the graphs showing the results of our performance and memory usage benchmarks.

In the diagrams, the different colored series describe the allocator configuration used, “minimal” and “default” means the minimal or default allocator configuration was used. “config” means the allocator settings recommended by `erts_alloc_config` were used. “fson” and “fsoff” means that full sweep every generation was on or off, respectively.

In the memory usage graphs, the Y-axis shows the number of bytes allocated by the runtime system and the X-axis shows the number of seconds passed since the start of the benchmark. Each peak in the memory usage is because of a section in the benchmark designed to allocate a lot of memory. We can see that the memory is recovered quite fast even without forcing full sweeps.

The Bstone graphs are scatterplots, each point is one benchmark iteration. The Y-axis shows how many “stones” were achieved during the run, and the X-axis shows how many seconds the run took. Since all the benchmarks were run for ten minutes, the series with better performance have more points than others because they had time to finish more iterations. Please note that the origin for the scatterplots is not (0,0).

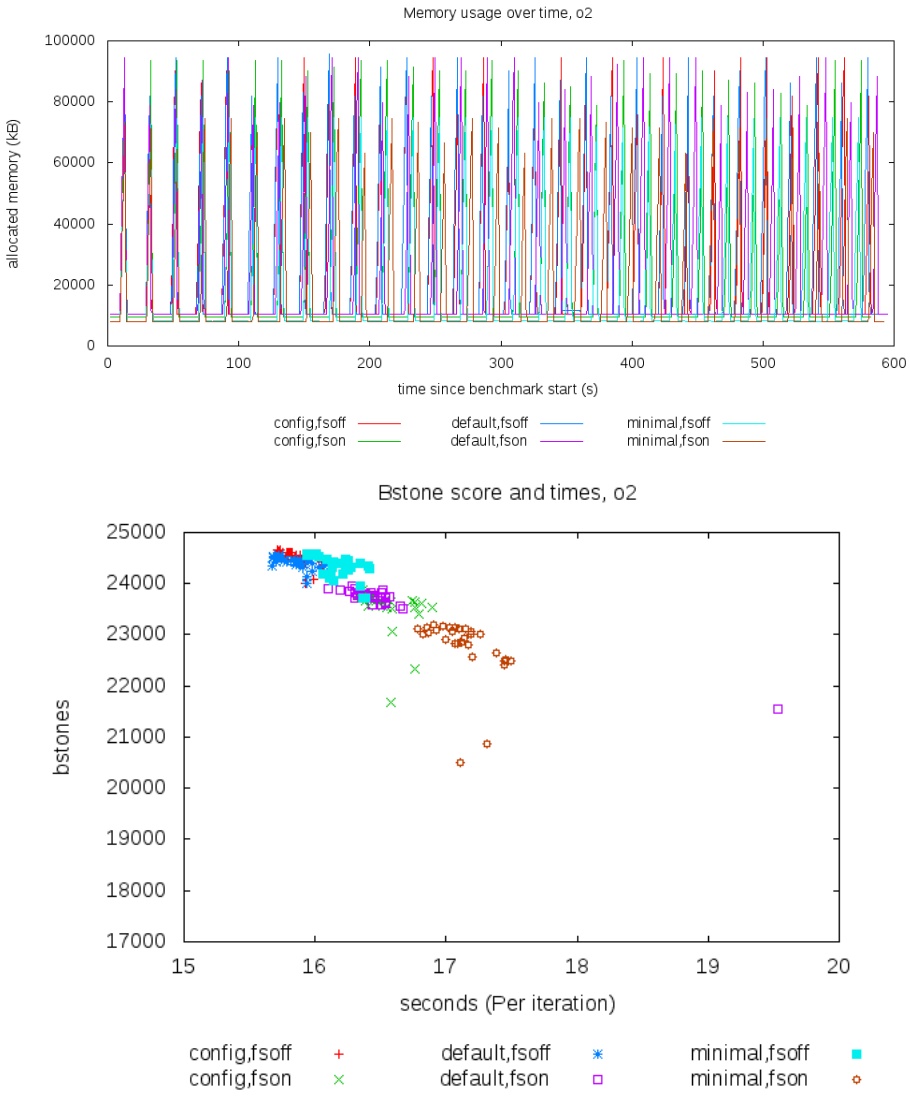


Figure 9: Built with `-o2`, no stripping or compression

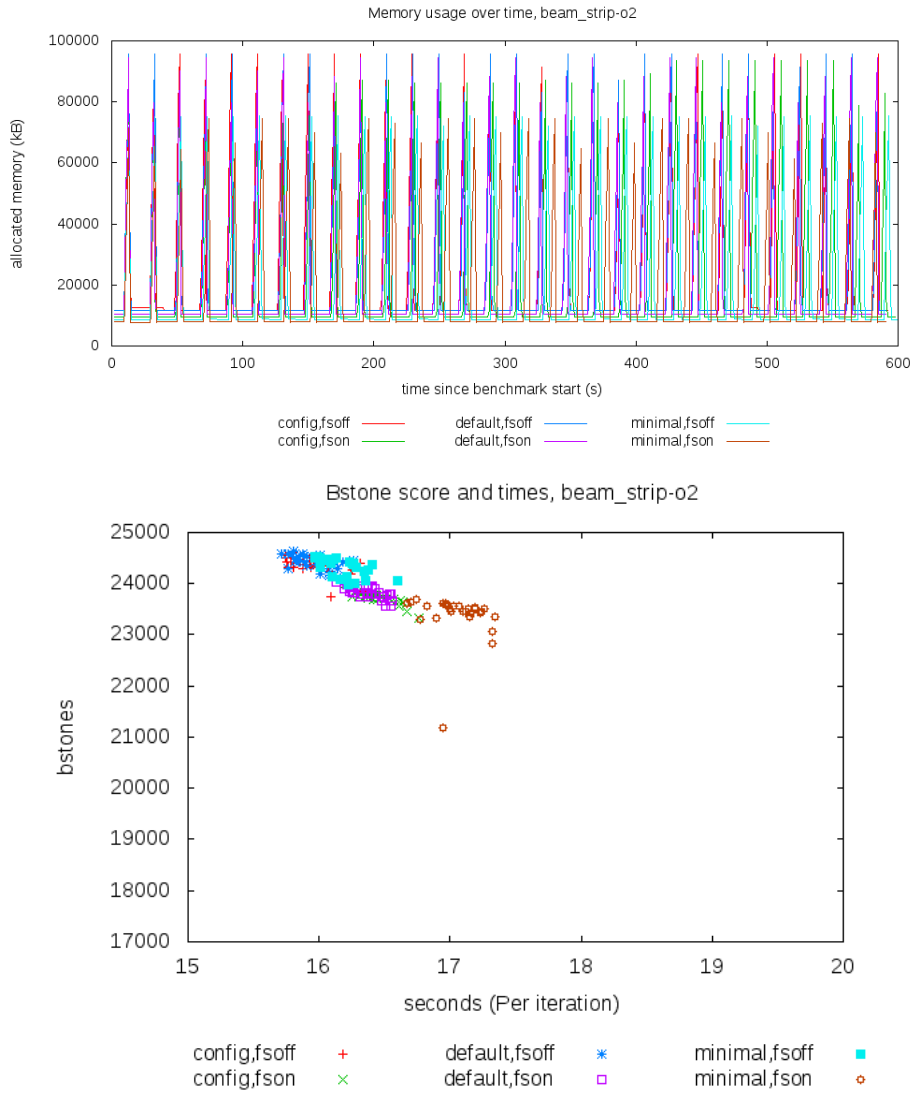


Figure 10: Built with `-o2`, stripped BEAM files

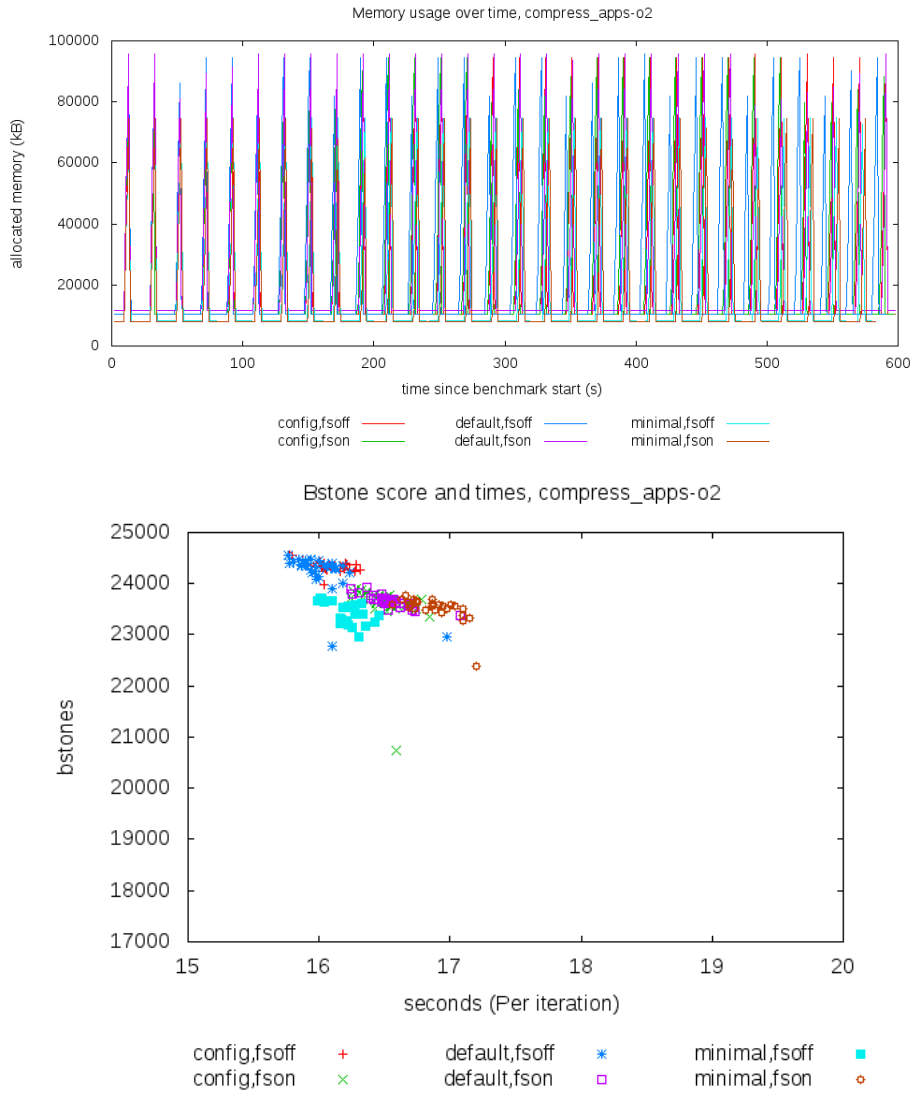


Figure 11: Built with `-o2`, compressed applications

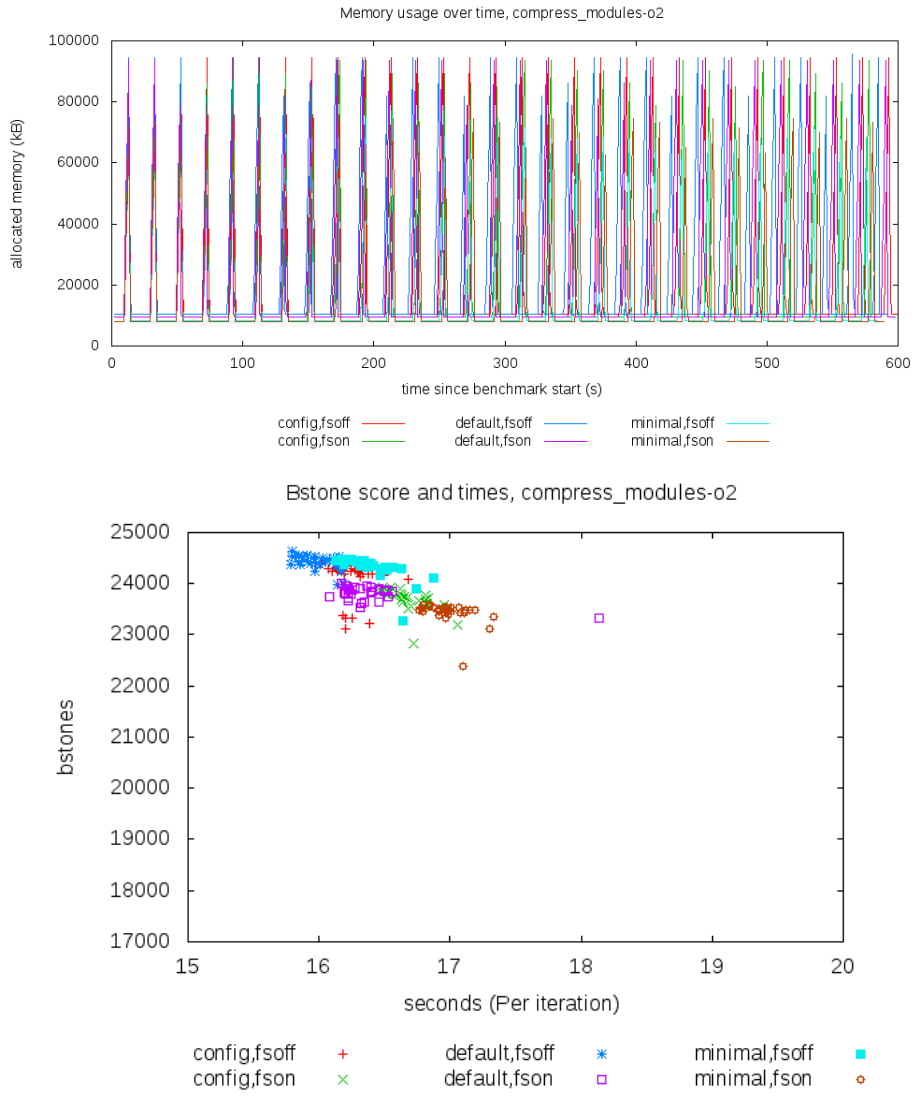
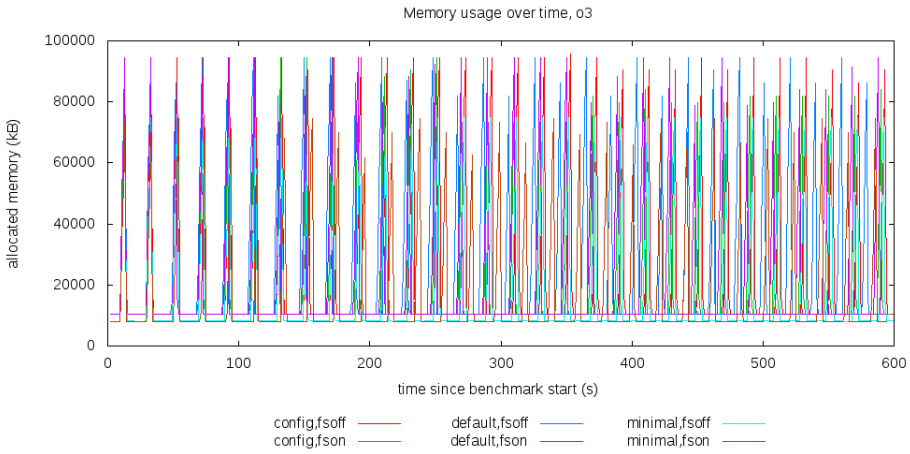


Figure 12: Built with o2, compressed modules



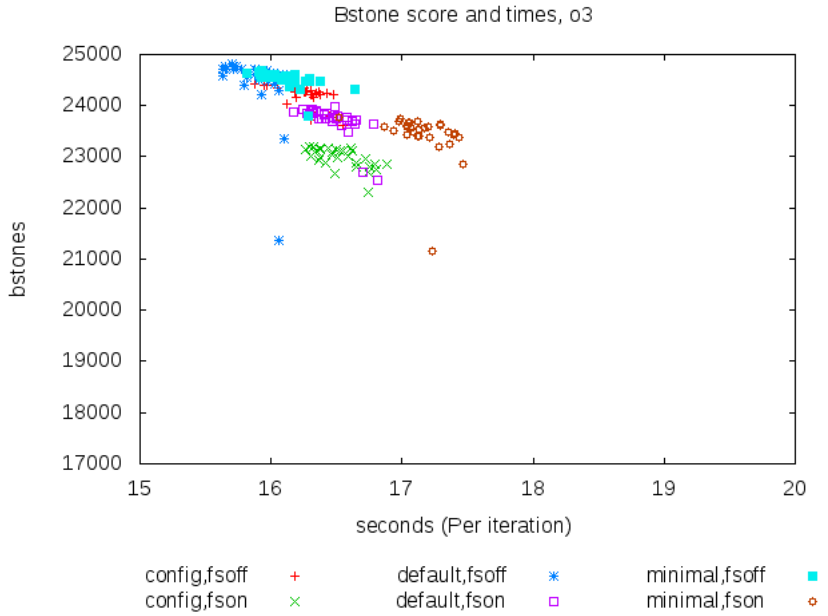


Figure 13: Built with `-o3`, no stripping or compression

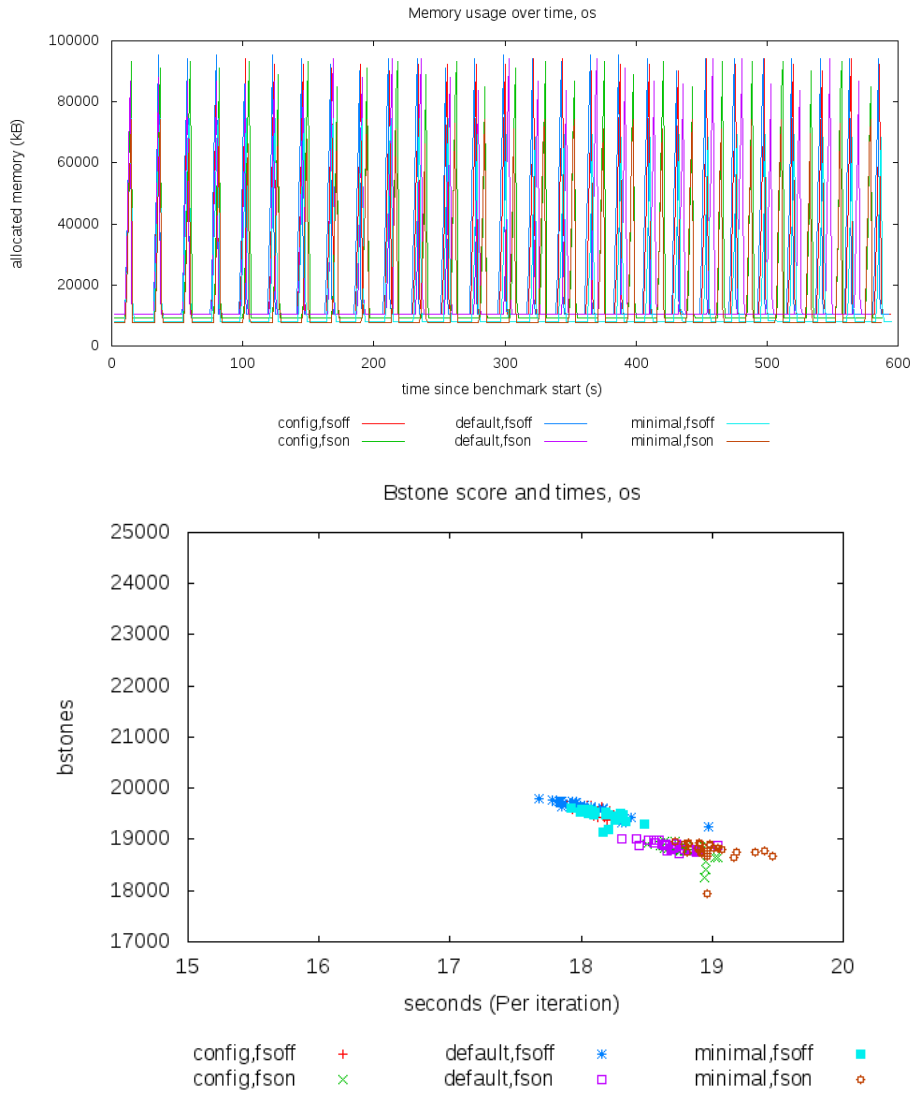


Figure 14: Built with `-os`, no stripping or compression

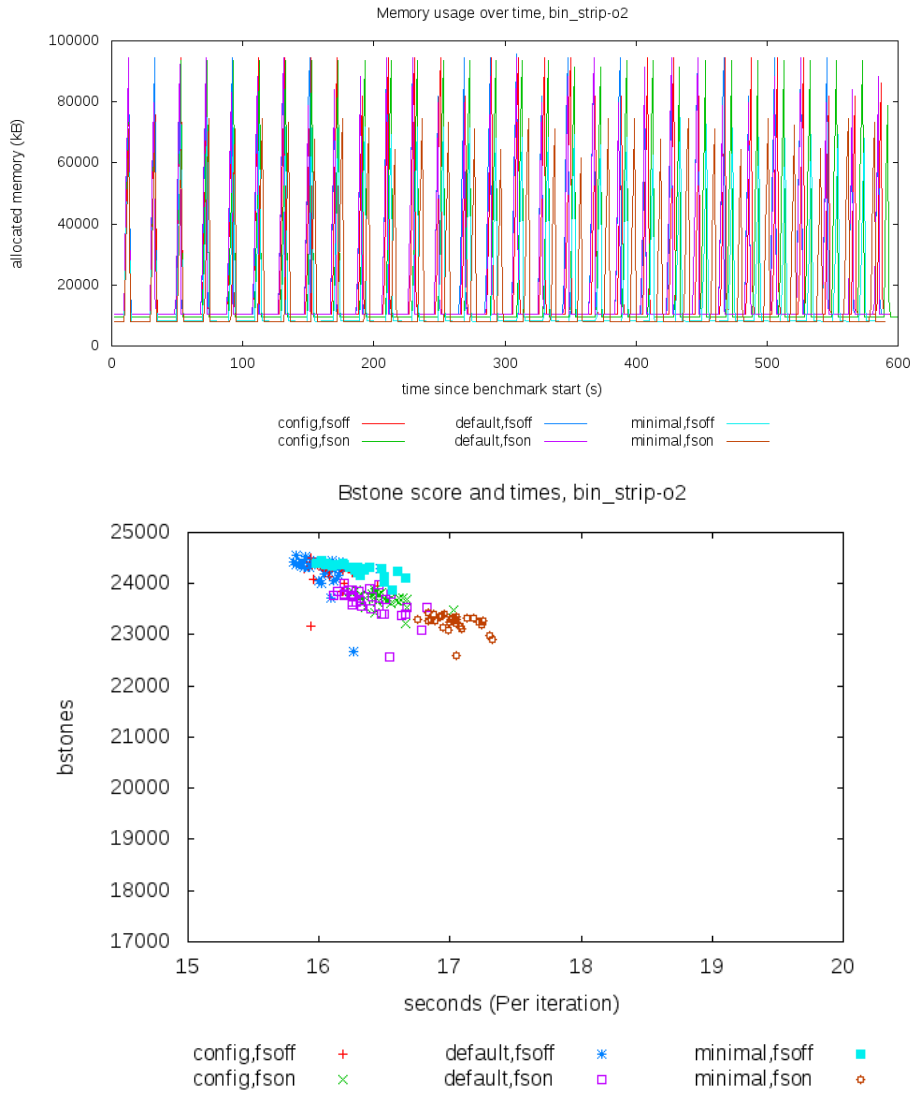


Figure 15: Built with `-o2`, stripped binaries

B References

ARM. Press release: Arm announces 2ghz capable cortex-a9 dual core processor implementation. http://www.arm.com/about/newsroom/25922.php, 2009.

Joe Armstrong. Erlang - a survey of the language and its industrial applications. In *In Proceedings of the symposium on industrial applications of Prolog (INAP96)*, 1996.

Joe Armstrong. Concurency oriented programming in erlang, November 2002.

Joe Armstrong. A history of erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X. doi: http://doi.acm.org/10.1145/1238844.1238850. URL http://portal.acm.org/citation.cfm?id=1238844.1238850.

Thomas Arts, Koen Claessen, and Hans Svensson. Semi-formal development of a fault-tolerant leader election protocol in erlang. In Jens Grabowski and Brian Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2004. ISBN 3-540-25109-X. URL http://dblp.uni-trier.de/db/conf/fates/fates2004.html#ArtsCS04.

Niclas Axelsson. A context aware file sharing protocol for mobile devices. Master's thesis, Uppsala University, To be published 2011.

BeagleBoard System Reference Manual Rev C4. beagleboard.org, December 2009. Revision 0.0.

BeagleBoard.org. Omap3530 beagleboard revc4 schematics, October 2010.

Staffan Blau, Jan Rooth, Jörgen Axell, Fiffi Hellstrand, Magnus Buhrgard, Tommy Westin, and Göran Wicklund. Axd 301: A new generation atm switching system. *Computer Networks*, 31(6):559–582, 1999. URL http://dblp.uni-trier.de/db/journals/cn/cn31.html#BlauRAHBWW99.

Tino Breddin. Taking the erlang/otp distro on github to the next level: Continuous builds and tests for community patches. Presented at Erlang Factory London. http://www.erlang-factory.com/conference/London2010/speakers/TinoBreddin, June 2010.

Jamil Chaoui. Omap^{TM} : Enabling multimedia applications in third generation (3g) wireless terminals. 2001.

Olivier Dubuisson. *ASN.1: communication between heterogeneous systems*. Morgan Kaufmann, 2001.

Bjarne Däcker. Concurrent functional programming for telecommunications: A case study of technology introduction. In *Licentiate Thesis. APPENDIX D. COLOPHON*, 2000.

Electronics Industries Association. Eia standard RS-232-C interface between data terminal equipment and data communication equipment employing serial data interchange, August 1969.

Ericsson. Erlang OTP R11B0 readme. `http://erlang.org/download/otp_src_R11B-0.readme`, May 2006.

Ericsson. Erlang OTP R13B04 documentation. `http://erlang.org/download/otp_doc_html_R13B04.tar.gz`, Feburary 2010a.

Ericsson. Erlang OTP R13B04 github repository. `https://github.com/erlang/otp/tree/OTP_R13B04`, Feburary 2010b.

Ericsson. Erlang OTP R14B1 readme. `http://erlang.org/download/otp_src_R14B01.readme`, December 2010c.

Christoffer Ferm. Adding special-purpose processor support to the erlang vm. Master's thesis, Uppsala University, To be published 2011.

Kunal Mankodiya; Simon Vogt; Ulrich Hofmann. Omap 3 based signal processing for biomedical engineering teaching. Presented at 17th European Signal Processing Conference (EUSIPCO 2009). `http://www.eurasip.org/Proceedings/Eusipco/Eusipco2009/contents/Papers-g.html`, August 2009.

Danny Hughes, Phil Greenwood, Barry Porter, Paul Grace, Geoff Coulson, Gordon Blair, Francois Taiani, Florian Pappenberger, Paul Smith, and Keith Beven. Using grid technologies to optimise a wireless sensor network for flood management. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 389–390, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3. doi: http://doi.acm.org/10.1145/1182807.1182869.

A. Ferworn J. Tran, M. Gerdzhev. Continuing progress in augmenting urban search and rescue dogs. 6th International Wireless Communications and Mobile Computing Conference (IWCMC 2010), June - July 2010.

Erik Johansson, Mikael Pettersson, Konstantinos Sagonas, and Thomas Lindgren. The development of the hipe system: design and experience report. *International Journal of Software Tools for Technology Transfer*, 4, 2002.

Håkan Mattsson, Hans Nilsson, and Claes Wikstrom. Mnesia - a distributed robust dbms for telecommunications applications. In Gopal Gupta, editor, *PADL*, volume 1551 of *Lecture Notes in Computer Science*, pages 152–163. Springer, 1999. ISBN 3-540-65527-1. URL `http://dblp.uni-trier.de/db/conf/padl/padl99.html#MattssonNW99`.

Renzo De Nardi and Owen Holland. Ultraswarm: A further step towards a flock of miniature helicopters. `http://cogprints.org/5571`, 2006.

Ulf Wiger. personal communication, June 2010.