Uppsala Master's Theses in
Computer Science 212
Examensarbete DV3
2002-04-04

Exploring Alternative Memory Architectures for Erlang: Implementation and Performance Evaluation

Jesper Wilhelmsson

Information Technology
Computing Science Department
Uppsala University
Box 311
S-751 05 Uppsala
Sweden

Abstract

The Erlang/OTP system is currently using a memory architecture with a private heap for each process. This choice was made in an attempt to lower garbage collection times with the copying generational garbage collector the system is using, since the garbage collection stop time interferes with the real-time behavior that Erlang aims at. This however results in costly communication since all messages has to be copied between processes.
The aim of this thesis is to design and implement two alternative memory architectures, in an attempt to improve performance of concurrent applications. The first one is a shared heap memory architecture, where all processes share a global heap. This architecture optimizes inter-process communication as it avoids copying of messages. The shared heap implementation described here is currently shipped together with the Open Source Erlang/OTP release 8 (R8). The other one is an experimental, hybrid architecture: provided that it can be decided at compile time that an allocated object is going to be sent as a message the object is going to be allocated in a global heap. All other objects are going to be placed in the private heap of each process. This hybrid architecture combines the advantages of the private and shared heap architectures. A thorough experimental evaluation of all architectures is also presented.

Supervisor and Examiner: Konstantinos Sagonas

1 Introduction

The concurrent functional programming language Erlang has been designed to ease the development of large-scale distributed soft real-time control applications. To measure up with the demands of this kind of applications, response times in the order of milliseconds are required. The memory model of Erlang works against these response times since it requires some kind of garbage collection. Either the garbage collector must guarantee that stop times for collection are short enough, or the system must be designed in a way that makes garbage collection times short. The later would probably not suffice for a real-time application since it is mostly based on luck.

The current implementation of the Erlang/OTP system has a stop-and-copy collector. This means that there are no actual real-time guarantees, hence the *soft* real-time claim instead. The memory architecture of Erlang/OTP is *process centric*, this means that every process allocates its own private memory (which includes the process's heap). With a model like this, garbage collection is usually not a problem since each collection only scans a single process and the programming style of Erlang encourages small and short lived processes.

Another characteristic of Erlang is that there is no shared memory. The only way for processes to communicate is through message passing. In Erlang/OTP this includes copying all the message data to the receiver's process heap. The time it takes to send a message is proportional to its size and sending of large messages is expensive. In fact it might be so expensive that Erlang users tend to avoid sending large messages, even in cases where they really want to send large amounts of data as a message. Even the Erlang programming rules and conventions at `www.erlang.org` specify: "Processes are the basic system structuring elements. But do not use processes and message passing when a function call can be used instead." So the cost of sending messages has made it bad programming style to use the language constructs as they were intended when Erlang was created.

In this report, besides the memory model of Erlang/OTP, two additional memory architectures will be explored. The aim is to improve overall performance by minimizing the time spent in message passing. To reduce communication overhead something else will have to be paid, and in this case the garbage collector will take the load.

The same problem exists in other concurrent languages as well where fast communication between processes stands against the cost and stop times of garbage collection.

The remaining part of this report will be structured in the following way: Section 2 gives a short description of Erlang. The basic ideas behind the language and its design are covered. Then follows descriptions of the three different memory architectures explored in this report. Advantages and disadvantages are discussed and implementation details are described. Section 3, (the Private Heap architecture), describes the memory management in the current Erlang/OTP system. There seems to have been some confusion in the Erlang community about the algorithms used, I hope this section settles any remaining questions. Section 4, (the Shared Heap architecture), presents the scheme now implemented and included as an optional part of the Erlang/OTP R8[1]. Section 5 gives a description of a potentially optimal system where the best features from the two earlier schemes are combined. Section 6 contains some benchmark results and the performance evaluation. More extensive benchmark results are presented in the appendix.

2 Erlang

Erlang is a strict, dynamically typed functional programming language with support for concurrency, communication, distribution, fault-tolerance, on-the-fly code reloading, automatic memory management, and multiple platforms [**?**]. It has been designed aiming to ease the programming of large soft real-time control systems which are commonly developed by the telecommunications industry. Erlang does have function closures, but typical Erlang programs are mostly first-order. Erlang's basic data types are atoms, numbers (floats and arbitrary precision integers), process identifiers, and references; compound data types are lists and tuples. There is no destructive assignment of variables or data, and the first occurrence of a variable is its binding instance. Function rule selection is done with pattern matching. Erlang inherits some ideas from concurrent constraint logic programming languages, such as the use of flat guards in function clauses.

[1] Erlang/OTP open source distribution can be downloaded from `www.erlang.org`

Processes in Erlang are extremely light-weight, their number in typical applications is quite large, and their memory requirements vary dynamically. Erlang's concurrency primitives—`spawn`, "`!`" (send), and `receive`—allow a process to spawn new processes and communicate with other processes through asynchronous message passing. Any data value can be sent as a message and processes may be located on any machine. Each process has a *mailbox*, essentially a message queue, where each message sent to the process will arrive. Message selection from the mailbox occurs through pattern matching. There is no shared memory between processes and distribution is almost invisible in Erlang. To support robust systems, a process can register to receive a message if another one terminates. Erlang provides mechanisms for allowing a process to timeout while waiting for messages and a catch/throw-style exception mechanism for error handling.

For programming in-the-large, Erlang comes with a module system. An Erlang module defines a number of functions. Only explicitly exported functions may be called from other modules. Calls to functions in different modules, called *remote calls*, are done by supplying the name of the module of the called function. Tail call optimization is a required feature of Erlang. As in other functional languages, memory management in Erlang is automatic through garbage collection. The real-time concerns of the language call for bounded-time garbage collection techniques; see [11, 6]. In practice, garbage collection times are usually small as most processes are short-lived or small in size.

Erlang is used in "five nines" high-availability (i.e., 99.999% of the time available) systems, where downtime is required to be less than five minutes per year. Such systems cannot be taken down, upgraded, and restarted when software patches and upgrades arrive, since that would not respect the availability requirement.

To perform system upgrading while allowing continuous operation, an Erlang system needs to cater for the ability to change the code of a module while the system is running, so called *hot-code loading*. Processes that execute old code can continue to run, but are expected to eventually switch to the new version of the module by issuing a remote call (which will always invoke the most recent version of that module). Once the old code is no longer in use, the old module is unloaded.

The Erlang language was purposely designed to be small, but it comes with libraries containing a large set of *built-in functions* (known as *BIFs*). With the Open Telecom Platform (OTP) middleware [10], Erlang is further extended with a library of standard solutions to common requirements in telecommunication applications (real-time databases, servers, state machines, process monitors, load balancing), standard interfaces (CORBA), and standard communication protocols (e.g., HTTP, FTP).

Erlang is currently used industrially both by Ericsson Telecom and by other companies for the development of high-availability servers and networking equipment. Some example products built using Erlang/OTP are: AXD/301, a scalable ATM switching system [1], ANx, an ADSL delivery system [7], a switching hardware control system, a next-generation call center, and a suite of scalable internet servers from Bluetail AB. Since 1994, the annual Erlang User Conference is the principal forum for reporting work done in Erlang and provides a record of Erlang's evolving industrial use; additional information about Erlang applications can be obtained through the relevant pages at `www.erlang.org`.

3 Private Heaps

The basic idea of the Private Heap (PH) architecture is that each process allocates and maintains its own, local, heap. The heap and the stack for each process are allocated in the same memory area and grow towards each other.

The main advantage with a scheme like this is that processes' heaps in general are small which usually makes garbage collection times fairly short. There are no guaranties though that this will keep the Erlang system from being blocked longer times than other architectures though. The possibility to schedule one garbage collecting process directly after another can make the total stop time much longer than a single garbage collection.

The PH architecture described here and used throughout this report is the memory model of Erlang/OTP R8. In this scheme, the heap is allocated when each process is created. When an Erlang process is created using the command `spawn` the user passes a list of arguments to be used by the new process. These arguments are allocated on the parent's heap before the new child process is created. In the creation of the new process, the argument list is copied to the child process's heap. To select heap sizes, the Fibonacci series is used.

3.1 Messages

A major drawback of the PH architecture can be seen when sending messages. All message data has to be copied from the sending process to the receiving. This imposes a major overhead in interprocess communication, especially if the message is big.

The system handles messages in two different ways depending on the size of the message. Messages smaller than or equal to 64 words are copied directly to the receiver's heap. All other messages are copied to a newly allocated memory area, and are then linked to a list of "off-heap" structures. This list is later copied in the garbage collection, this time to the young generation of the heap. This means that a long lived message will be copied once more to the old generation in the next garbage collection.

Each process has a queue of messages that has been sent to it but not yet handled by the process. All messages are placed in this queue, regardless of their size. This means that some of them are already allocated on the process heap and some are in the "off-heap" list.

3.2 Garbage collector

Each time a process wants to allocate heap or stack space, a check is made to ensure that there is enough room on the heap. If there is not enough room the garbage collector is invoked.

The garbage collector is a two generational stop-and-copy. It has two different modes of operation, corresponding to the minor and the major collection. The root set for each garbage collection consists of the process stack, the message queue and an optional vector of pointers sent to the garbage collector. Note that the garbage collector does not have to scan the stack of any other process. Instead garbage collection happens locally.

3.2.1 The first collection

The opinions differ wheter to call the first collection in the system a minor or a major one. The garbage collector scans the entire heap in the first collection which speaks in favour of calling it a major collection. However since there is no old generation one might argue that the first collection is to be considered a minor one since only the young generation is scanned. In many papers displaying statistics over generational garbage collectors, it can be observed that programs has made a couple of minor collections and no major. This inferes that the first collection is considered a minor one in these papers. This is also the case in this report. In our case it depends on the implementation of the system, the first collection simply use the routines for minor collection since there is no old heap to consider and it seems likely that this is the case in other papers to.

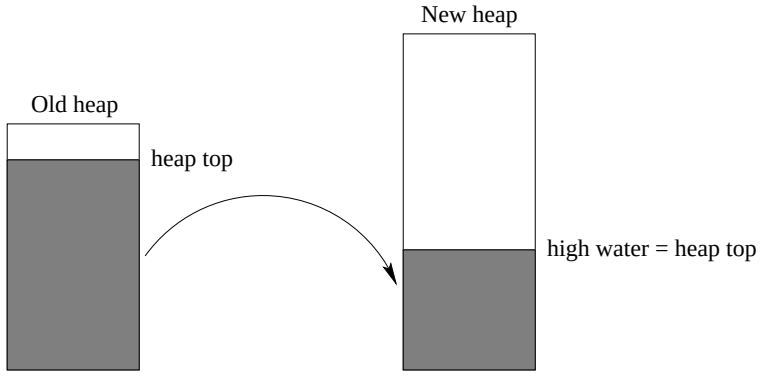


Figure 1: A new heap is allocated and the live data on the old heap is copied.

The first garbage collection in the system, and also the first minor collection after each major collection will follow this scheme: A new heap is allocated whose new size is the closest Fibonacci number bigger than old heap size + message queue size. *(The message queue size is the size of all messages in the message queue.)* All live data on the heap and in the messages are copied to the new heap and the high water mark is set; see Figure 1.

The general case of garbage collection will start with a minor collection, but there are a couple of cases where the minor collection will be skipped:

- If the data below the high water mark will not fit on the old generation
- If a flag indicates that a major collection is needed

3.2.2 Minor collections

If there is data below the high water mark at the beginning of the collection, the garbage collector checks whether there is an old generation. If there is not one already, a new memory area is allocated for the old generation. Since the high water mark is where the heap top was at the end of the last collection, all data above the high water mark has been created since the last garbage collection. Data below this mark survived the last collection but has not yet been placed in the old generation.

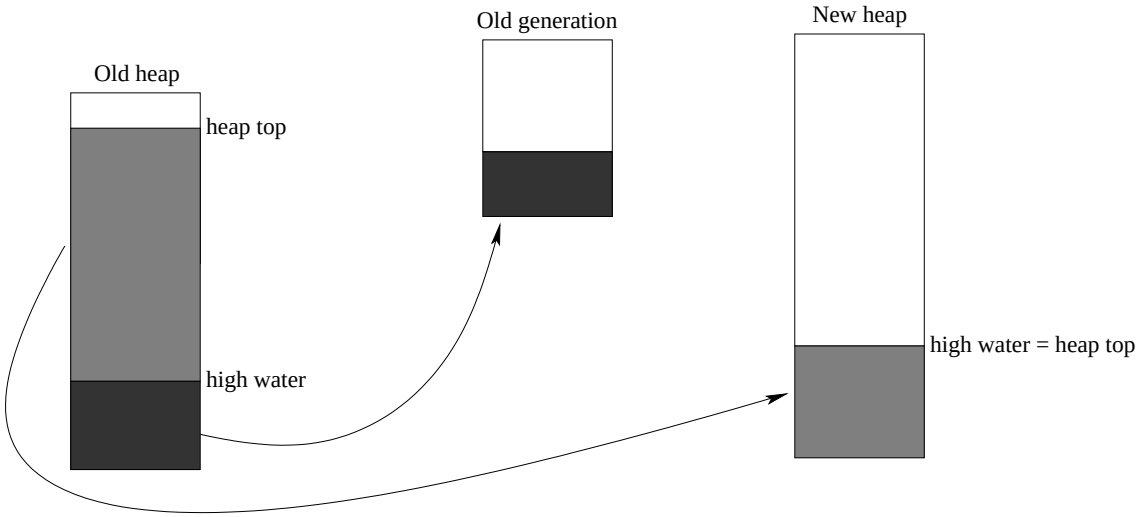


Figure 2: Depending on which side of the high water mark the data originates from, it will be copied to either the old generation or the new young generation.

A new heap is allocated that will hold all data available in the heap and the "off-heap" list. As can be seen in Figure 2, all live data from the heap will be moved to:

- the old generation if it came from below the high water mark;
- the new young generation otherwise.

The copying is done in two passes, using a Cheney scan [2]: First all pointers in the root set are followed and copied, then the new heap is scanned and all pointers are checked and copied if they point to the old, soon to be deleted, heap.

When all data is moved, the old heap is freed and the high water mark is set to the new heap top. A test is then made to check that the heap is not unnecessarily big. The test says that a heap where less than 25% is needed, is too big if it is bigger than 8000 words or bigger than the old generation. If the test finds that the heap is too big, it will shrink the new heap to a more convenient size (which is the closest heap size from the Fibonacci series bigger than 3*(the size of the live data + need + stack size)). This is normally done without additional copying of data, except for the stack of course which has to be moved since it is located in the top of the heap memory area.

If the free space on the new heap is enough to accommodate the need, the garbage collection ends. Otherwise a major collection takes place.

3.2.3 Major collections

When a major collection occurs, a new heap is allocated to hold all data available in the heap, the old generation, and the "off-heap" list. All live data is copied to the new heap using the same Cheney scan as in minor collections. The old heap and the old generation is freed and the high water mark is set to the new heap top; see Figure 3.

If the heap is still not big enough after a major collection, it will be enlarged and the data on the heap might be copied one more time to the new, enlarged, heap. If more than 75% of the new heap is needed, a flag is set to indicate to the collector that the next time a garbage collection occurs, the heap should be enlarged. This means that the new heap size in that collection will be chosen larger than actually needed at that point.

If less than 25% of the new heap is needed, the heap is shrunk to the next heap size bigger than 2*(size of live data + need + stack size).

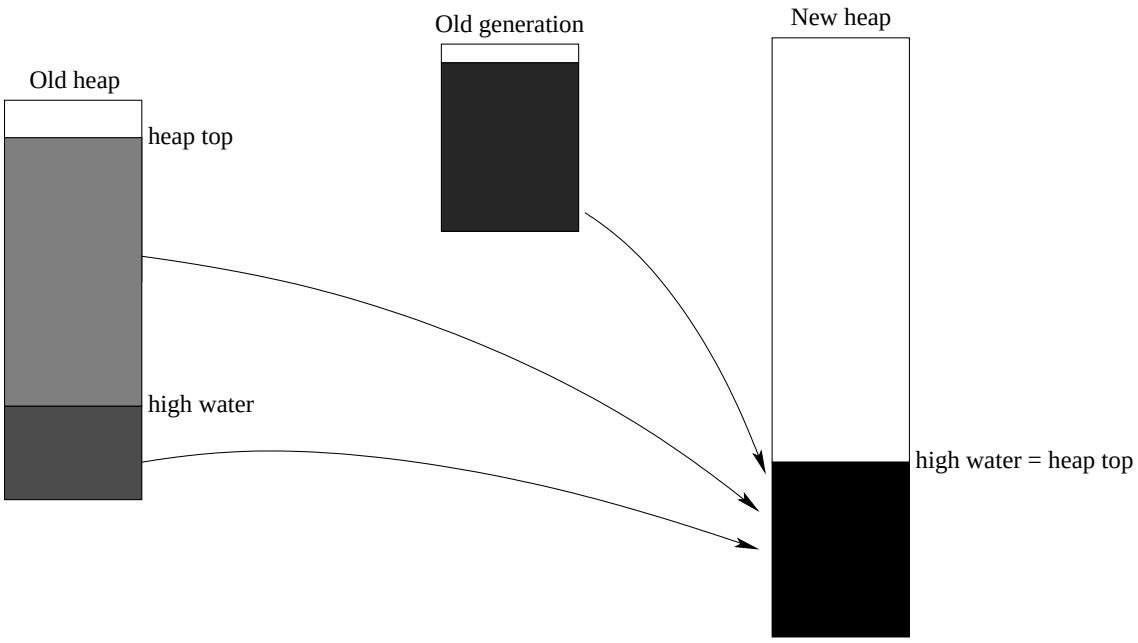


Figure 3: The resulting new heap will contain all live data for this process in the young generation. There is no old generation after a major collection.

4 Shared Heap

The main motivation for developing the Shared Heap (SH) architecture is to provide better usage of the allocated space and at the same time improve the performance by eliminating the copying of messages which are sent between processes. If all processes have access to all memory, they will only have to pass a pointer around instead of copying the entire message. A SH is also likely to give better memory usage, thanks to its global memory management. In the PH architecture each process allocates a chunk of memory from the operating system. For processes that does not use much heap, this memory is mainly unused. With the SH architecture the global memory manager will make sure there is enough memory for all processes without locking memory areas for processes that do not need the memory. This will save a lot of memory in systems with many small processes. The fact that there is only one large heap instead of several small ones will also have a positive effect on fragmentation. No small holes in the memory will appear because of short lived processes. On the other hand, a possible drawback is that since in the SH architecture data from different processes are interleaved, the cache performance might deteriorate. No special effort has been made to measure cache performance in this report. A larger heap, and more important, a larger root set is expected to increase the garbage collection times. But the total amount of time spent in garbage collection does not necessarily have to increase since the garbage collector is

expected to be invoked less times. The crucial time for Erlang's soft real-time properties are however the time the system is blocked by garbage collection. The present belief of Erlang community is that with shared heap architecture for processes the maximum (or even average) stop time will be longer which would break the soft real-time properties of Erlang. One aim of this thesis is to investigate whether this is true. The first implementation of a SH architecture has the following characteristics:

- Separate stack for each process
- One shared heap for all processes
- No messages are copied between processes
- Copying generational garbage collector

4.1 Separate stacks

In the SH architecture, the stack is separated from the heap and kept private to each process. The fact that the stack is not a part of the heap anymore means that less work is needed when resizing the heap in garbage collection, since the stack will not have to be copied/moved. This is not really noticeable in the benchmarks though.

The implementation was straightforward. Allocate and initialize the stack the same way the heap used to be. A couple of extra stack pointers were added to the process structure, and all memory allocation routines and the garbage collector were changed to handle the new design.

4.2 Shared heap

To accommodate the SH, some global heap pointers have been added to the system. The global heap pointers are used to pass the values from one executing process to the next. During execution of a process the current heap pointers are stored in the process structure as before. The global heap is allocated before the first process is created so there will always be a heap to use for passing arguments to a new process. Since each parent now shares the heap with its children, no copying of the arguments is needed.

4.3 Messages

In the SH architecture, the only piece of information that a process has to send to another process it wants to communicate with is a pointer to the message data. No messages are copied in the send operation. In the PH architecture big messages were placed on an "off-heap" list which had to be copied to the heap in the following garbage collection. This copying will not be needed in the SH architecture since the message data is never placed in the "off-heap" list in the first place. Messages will however be copied once to the old generation if they live long enough. This means that long lived messages will only be copied once, and that is during the garbage collection phase, instead of three times per receiving process, which is the case for messages larger than 64 words in the PH architecture. All messages are kept within the heap at all times and there is no need to allocate data "off-heap" while sending messages.

4.4 Garbage collector

The shared heap memory architecture might suffer from increased garbage collection times. The minor and major collections are the same. The difference lies in what is considered as the root set. In the SH architecture, the garbage collector has to consider all processes as a root set since any process might have pointers to the heap in its stack or message queue. The root set might thus be quite large. A large root set is likely to result in long garbage collection stop times.

To reduce the size of the root set an optimization has been added to the naive way of considering what is the root set. There is now an array of "live processes" where a pointer to every process created is stored. When a process dies, the pointer is removed. At each major collection all live processes are considered as a root set. At the first minor collection all live processes have to be considered as a root set too, since there is no old generation after a major collection. However, at all subsequent minor collections only the processes that have actually been active since the last major collection are considered as a root set.

A process is marked as having been active if it:

- Is scheduled for execution. It may be the case that a process which is scheduled for execution never touches the heap, which means it should not really be part of the root set, but the current implementation marks it anyway.
- Receives a message from a working process. This must be done since the receiving process might be the only process that keeps the pointer to the message. The sender may throw the pointer away or might actually even have died by the time of the next garbage collection.

The functions which build the root set have been completely reworked to now use an array of root sets, one for each live process. A flag was added to the process structure to mark active processes, and at several points in the garbage collector a loop was added to make the garbage collector consider all active processes as a root set.

In the current implementation the shrinking of the heap is still needed since the allocated new heap at most times is bigger than the old heap. This is due to the fact that the "off-heap" list have to fit on the heap as well after the collection. The "off-heap" list is not only used by messages in the PH architecture, it is also used by the guard BIFs to allocate memory, since these are not allowed to allocate directly on the heap for safety reasons. So, even though no placing of messages "off-heap" happens in the SH architecture, the guard BIFs make it necessary to consider the "off-heap" list anyway.

5 An Optimal Memory Architecture?

Both architectures described above have their advantages and disadvantages. The PH architecture keeps the garbage collection local to the process and has the potential of shorter garbage collection times, which is something Erlang/OTP is referring to when claiming soft real-time properties [**?**]. The garbage collection of one process does not interfere with other process heaps. An other important advantage of the PH architecture is the fast deallocation of dead processes. When a process dies its stack and heap can be reclaimed immediately without the need of a garbage collection. This can be used for simple memory management by spawning a new process for operations that creates a lot of garbage. The cache performance is also something that might speak in favor of the PH.

The SH on the other hand avoids copying of messages, which is crucial since the only way to communicate between processes in Erlang is through message passing. This means that message data is not traversed when a message is sent. The only time the message data will be traversed is while it is being created and perhaps during garbage collection if it is a long lived message.

It is possible to produce a system that combines the advantages of the two schemes if we can decide at compile time if data is sent in a message or not. This would give the opportunity to allocate all message data in a global message area and keep the rest of the data in a private heap. In this way we can reduce the cost of garbage collection and avoid copying of messages. To be able to decide this some kind of *escape analysis* is needed. There are analyses that do this for the Marmot native Java compiler [8]. Some similar analysis might be tuned to be effective in an Erlang environment. To further improve the system, a non-moving garbage collector should be used on the global message area. With a system like this, message data is only traversed when creating the message, not when sending nor in garbage collection.

5.1 Implementation of a Hybrid Architecture

The Hybrid architecture implemented and described here is very similar to the PH architecture. This makes it possible to see the difference in performance without interference from other differences in the implementation. The private heaps, all allocation and garbage collection algorithms used, are exactly the same as in the PH architecture. The stack for each process is allocated in the same memory area as the local heap and they grow towards each other like in the PH architecture.

The global message area is similar to that of the SH. No escape analysis has been implemented yet, so to be able to decide what is a message and what is local data three new BIFs were added. These new BIFs, `cons`, `tuple` and `send`, operate exclusively on the global message area and assume all incoming data is either simple data like numbers or atoms or already allocated on the global message area. The

Program	Number of processes	Number of messages
procs 100 100	100	15,352
procs 100 1000	100	15,352
procs 1000 100	1,000	602,602
eddie	2	2,121
ring	100	501,093
life	100	800,396
BEAM compiler	6	801
NETSim TMOS	4,066	123,869
NETSim alarm server	12,353	288,835
NETSim coordinator	591	215,536

Table 1: Number of processes created and messages sent.

benchmark programs has been rewritten to use these BIFs instead of the `|` (cons), {...} (tuple) and `!` (send). When running the rewritten benchmarks in the PH architecture and the SH architecture the BIFs operate on the normal heap and does exactly the same thing as the built in commands `|`, {...} and `!`. The BIFs are used in these architectures to make sure that the hybrid architecture is not penalized by the fact that the global send and cons are BIFs while `|`, {...} and `!` are built in commands.

In most places where data is copied in the PH architecture the call to copy can be avoided completely in the hybrid architecture. This is the case in send. But in some cases the system might want to copy something that may or may not be global message data, depending on the Erlang program. This is the case for the arguments when creating a new process. To avoid the copying in these cases a test is made in copy to see if the object resides in the message area or not. If the object is allocated in the message area the copying of the data is skipped.

The size of the message area is chosen big enough to prevent any need to invoke the garbage collector. In this way we simulate the speed of a non-moving collector since this has not been implemented yet. A non-moving garbage collector would of course have some overhead involved in keeping track of free memory areas etc. While we expect this time overhead to be quite small, the run times presented in the tables should be considered minimum times for the benchmarks, and perhaps not achievable times for a real system.

6 Performance Evaluation

The performance of all architectures is of course dependent on the initial heap size used, since this will affect the frequency of the garbage collections in the system. Appendix A contains extensive tables comparing the PH architecture and the SH architecture with different initial heap sizes. In this section only one of each is considered. The PH architecture is started with an initial heap size of 233 words per process since this is the default setting in Erlang/OTP and the setting most frequently used in the Erlang community.

To be able to decide whether the benchmarks are concurrent, Table 1 shows the number of processes created and the number of messages sent in each benchmark. The number of messages in the **NETSim** application differ with some hundred messages between runs (less than 0.1%), the numbers here are mean values. Tables showing the message sizes can be found in Appendix B.

The time and space performance of the PH architecture is presented in Tables 2 and 3. The SH architecture is used with an initial heap size of 10.946 words. Its performance is shown in Tables 4 and 5. Tables 6 and 7 show the performance of the hybrid architecture started with private heap sizes of 233 words each and a global message area big enough not to invoke any garbage collection in this set of benchmark programs. All times are shown in milliseconds. Minor and major refers to garbage collection.

The programs used as benchmarks are:

procs(number of processes, message size) Sends messages in a ring of processes. Each process creates a new message when it is spawned and sends it to the next process in the ring (its child).

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
procs 100 100	1,042	248	81	179	1	0	0	23.80	7.77	17.18	0.10
procs 100 1000	9,327	2,036	845	1,424	272	1	0	21.83	9.06	15.27	2.92
procs 1000 100	42,366	10,295	3,757	7,810	77	33	1	24.30	8.87	18.43	0.18
eddie	28,656	19	6	6,540	793	57	54	0.07	0.02	22.82	2.77
ring	2,165	819	205	0	0	0	0	37.83	9.47	0.00	0.00
life	5,299	1,830	712	523	2	5	0	34.53	13.44	9.87	0.04
BEAM compiler	10,579	84	54	1,752	342	129	100	0.79	0.51	16.56	3.23
NETSim TMOS	530,126	1,817	581	7,893	849	56	16	0.34	0.11	1.49	0.16
NETSim alarm server	138,474	5,525	2,553	8,332	588	110	59	3.99	1.84	6.02	0.42
NETSim coordinator	153,333	2,888	1,137	3,867	106	64	3	1.88	0.74	2.52	0.07
same 250 250	450	164	62	139	0	0	0	36.44	13.78	30.89	0.00
same 1000 250	4,000	1,415	519	701	0	1	0	35.38	12.97	17.52	0.00
same 250 1000	994	440	189	377	1	1	1	44.27	19.01	37.93	0.10
same 1000 1000	6,141	2,525	1,033	1,558	1	2	1	41.12	16.82	25.37	0.02
keep 250 250	900	153	50	179	0	1	0	17.00	5.56	19.89	0.00
keep 1000 250	5,909	1,398	490	730	124	0	0	23.66	8.29	12.35	2.10
keep 250 1000	2,673	381	142	293	237	2	1	14.25	5.31	10.96	8.87
keep 1000 1000	13,667	2,362	879	1,798	1,001	5	43	17.28	6.43	13.16	7.32
garbage 250 250	916	150	50	170	0	0	0	16.38	5.46	18.56	0.00
garbage 1000 250	5,948	1,398	483	802	0	0	0	23.50	8.12	13.48	0.00
garbage 250 1000	2,504	382	151	240	86	0	0	15.26	6.03	9.58	3.43
garbage 1000 1000	12,664	2,523	905	1,172	362	11	0	19.92	7.15	9.25	2.86

Table 2: Time performance of the private heap architecture.

Each message has a counter that ensures it will be sent 100 times. The code of the benchmark is shown in Appendix D.1.

eddie An Erlang application implementing an HTTP parser which handles http-get requests.

ring A concurrent benchmark which creates a ring of 10 processes and sends 100,000 messages.

life Conway's game of life on a 10 by 10 board where each square is implemented as a process.

BEAM compiler Compiles the file `otp/lib/gs/src/gtk_generic.erl` from the Erlang/OTP R8 open source.

NETSim A large Erlang application (630.000 lines of Erlang code) implementing a **N**etwork **E**lement **T**est **Sim**ulator. A product family mainly used to simulate the operation and maintenance behavior of a network. In the actual benchmark a network with 20 nodes is started and then each node sends 100 alarm bursts through the network. Three parts of the NETSim system is used as benchmarks, the coordinator, the TMOS server, and the Alarm server.

same, garbage and keep Rewritten versions of the program **procs**. They have been modified to use the global message area BIFs. The arguments to the programs are the same as to **procs**. These are the only benchmarks rewritten to work on the hybrid system at the moment. **same** creates *one* message and distributes it among the processes. **garbage** creates a new message each time and throws the old one away, and **keep** creates new messages each time and stores the old ones in a list. In these three benchmarks each message is sent 10 times. These benchmarks are shown in Appendix D.2.

The platform used for the **NETSim** benchmarks is a SUN Ultra 10 with a 333 MHz Sun UltraSPARC-IIi and 256 MB memory. For all other benchmarks a SUN Enterprise 3000 with two 248 MHz Sun UltraSPARC-II and 1.2 GB memory has been used.

The columns "Max time in..." in Tables 2 and 4 shows the maximum stop times of each benchmark. This is the time the entire system is blocked, working on garbage collection. A comparison between them shows that the expected increase in stop time does not necessarily appear when the root set becomes

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
procs 100 100	1,355	10	20	627,486	328,611	486,250	1,210,601	6,363
procs 100 1000	13,812	1,667	1,399	5,740,018	429,837	613,565	10,344,278	2,218,289
procs 1000 100	45,046	320	68	23,080,952	2,765,285	3,997,853	45,576,943	373,321
eddie	3,601	566	14,600	16,919,311	67,325	78,109	27,870,820	4,179,040
ring	4	1	3	3	11,742	41,650	0	9
life	28,690	82	56,814	119,326	33,139	52,834	469,079	3,592
BEAM compiler	1,620	21	757,836	2,864,006	1,363,278	1,375,659	4,469,462	1,787,055
NETSim TMOS	108,817	16,370	307,365	10,840,825	1,097,314	2,663,556	17,845,366	4,324,525
NETSim alarm server	121,938	6,342	533,281	18,452,823	2,031,703	2,720,603	35,331,680	3,197,814
NETSim coordinator	38,037	686	222,028	12,689,143	146,521	176,424	25,920,306	1,184,128
same 250 250	526	0	2	690,618	884,285	998,086	1,389,263	0
same 1000 250	5,636	0	2	2,852,968	1,645,754	2,942,869	5,810,813	0
same 250 1000	451	1	2	1,889,718	2,807,446	6,581,363	3,781,931	12,042
same 1000 1000	2,142	1	2	7,605,773	13,401,739	25,134,875	15,228,296	12,042
keep 250 250	742	1	485	841,621	1,180,532	2,698,329	1,435,731	4,042
keep 1000 250	2,992	247	1,985	3,414,871	6,935,181	12,572,252	5,835,231	1,126,342
keep 250 1000	876	250	5	1,312,575	4,538,804	5,828,673	1,790,979	2,464,350
keep 1000 1000	3,862	1,000	908	7,571,296	19,252,504	26,972,138	11,095,219	9,892,725
garbage 250 250	2,025	1	2	614,881	197,714	484,810	1,240,333	987
garbage 1000 250	11,272	1	2	2,547,663	902,417	1,849,400	5,204,179	987
garbage 250 1000	1,352	250	4	910,208	1,486,467	1,792,840	1,742,738	901,088
garbage 1000 1000	8,676	1,000	4	3,696,325	2,641,906	3,835,095	7,149,464	3,618,338

Table 3: Space performance of the private heap architecture.

larger. In the SH architecture the stop times are in many cases about half of those of the PH architecture. **eddie** got maximum stop times around 55 ms in the PH architecture, in the SH architecture these pauses are only about 11 ms. But there are also examples where the stop times actually do increase. A good example of this is **procs**. This increase in stop time is expected for **procs**, since the number of active processes between each garbage collection is very high which makes the root set quite large. The **BEAM compiler** is the benchmark that suffers from the longest stop times in the PH architecture (129 ms). The longest stop time in the SH architecture happens in **NETSim alarm server** (124 ms). This is without respect to the **keep** benchmark that is designed to break the shared heap and give long stop times.

As expected the run times for the Hybrid Architecture are shorter than the other two architectures.

The global memory management in the SH architecture makes a notisable difference. Comparisions between Tables 3 and 5 show programs that in the SH architecture only needs 12% of the memory required for the PH architecture. **NETSim TMOS server** and **procs 1000 100** is two examples. The part of the private heaps that is occupied by the stack is included in the allocated heap space, but not in the used. This might seem unfair to the SH architecture where the stacks are not accounted for in the heap measurements. The stack sizes are however very small so the difference is not noticeable. For **procs 1000 100** there are 1000 processes with stacks of 4 words, which adds up to 0.1% of the allocated memory in the PH architecture. The servers in **NETSim** has equally sized stacks.

The tables also show the number of pointers from the root set to the young and the old generation in garbage collection, and the sum of live data recovered in major and minor collections.

7 Related work

There has recently been some similar work done by Marc Feeley [4], where the private heap architecture of Erlang/OTP was compared to the unified heap architecture used in the ETOS system. In Mark Feeley's paper cache performance is said not to be destroyed by the shared heap. The two different systems, (Erlang/OTP and ETOS), might have other differences in implementation that interfere with

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
procs 100 100	742	9	0	139	5	4	2	1.21	0.00	18.73	0.67
procs 100 1000	8,966	10	0	3,339	59	76	31	0.11	0.00	37.24	0.66
procs 1000 100	31,693	335	0	8,174	47	62	19	1.06	0.00	25.79	0.15
eddie	21,902	2	0	3,456	356	13	9	0.01	0.00	15.78	1.63
ring	1,635	328	0	0	0	0	0	20.06	0.00	0.00	0.00
life	3,493	467	0	176	1	1	1	13.37	0.00	5.04	0.03
BEAM compiler	8,560	0	0	945	161	64	51	0.00	0.00	11.04	1.88
NETSim TMOS	502,729	139	0	8,775	934	68	41	0.03	0.00	1.75	0.19
NETSim alarm server	146,072	295	0	30,517	570	124	45	0.20	0.00	20.89	0.39
NETSim coordinator	155,510	205	0	1,732	30	37	4	0.13	0.00	1.11	0.02
same 250 250	158	18	0	13	0	0	0	11.39	0.00	8.23	0.00
same 1000 250	3,018	274	0	918	4	5	4	9.08	0.00	30.42	0.13
same 250 1000	167	18	0	14	0	0	0	10.78	0.00	8.38	0.00
same 1000 1000	2,937	270	0	832	4	5	4	9.19	0.00	28.33	0.14
keep 250 250	961	18	0	242	47	89	40	1.87	0.00	25.18	4.89
keep 1000 250	6,556	275	0	1,486	720	492	430	4.19	0.00	22.67	10.98
keep 250 1000	4,237	18	0	1,235	733	511	388	0.42	0.00	29.15	17.30
keep 1000 1000	18,855	272	0	5,375	2,897	1,877	1,427	1.44	0.00	28.51	15.36
garbage 250 250	750	18	0	71	1	1	1	2.40	0.00	9.47	0.13
garbage 1000 250	5,331	275	0	1,012	1	6	1	5.16	0.00	18.98	0.02
garbage 250 1000	2,457	18	0	183	8	6	5	0.73	0.00	7.45	0.33
garbage 1000 1000	12,040	273	0	1,279	8	20	5	2.27	0.00	10.62	0.07

Table 4: Time performance of the shared heap architecture.

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
procs 100 100	97	3	133	719,523	75,025	75,025	1,406,471	54,427
procs 100 1000	130	6	162	9,325,503	514,229	514,229	18,245,249	516,995
procs 1000 100	849	7	805	29,071,002	514,229	514,229	58,193,774	381,359
eddie	3,600	200	26,267	17,513,842	46,368	46,368	26,919,506	3,587,600
ring	0	0	0	0	2,392	10,946	0	0
life	347	0	68,546	157,549	28,656	28,657	613,709	0
BEAM compiler	783	5	585,555	2,483,086	1,346,264	1,346,269	3,948,724	1,405,146
NETSim TMOS	10,421	205	1,974,496	15,148,233	317,810	317,811	16,185,966	8,230,296
NETSim alarm server	25,676	67	43,633,622	15,060,866	317,811	317,811	24,448,981	4,418,664
NETSim coordinator	7,494	12	539,556	4,588,232	121,392	121,393	9,404,429	323,489
same 250 250	22	0	276	7,077	17,710	17,710	11,835	0
same 1000 250	395	1	14,569	273,227	28,656	28,657	797,706	9,139
same 250 1000	22	0	277	9,343	17,710	17,710	14,879	0
same 1000 1000	395	1	5,288	276,308	28,656	28,657	800,728	10,450
keep 250 250	7	2	78	986,361	832,038	1,346,269	1,103,701	451,399
keep 1000 250	13	4	449	6,043,469	5,135,604	5,135,606	4,599,760	6,528,852
keep 250 1000	9	4	162	4,493,235	4,108,483	4,108,485	3,312,473	5,689,189
keep 1000 1000	13	6	309	18,996,219	15,672,619	15,672,621	13,950,787	24,072,391
garbage 250 250	64	1	14	233,128	46,367	46,368	463,900	12,943
garbage 1000 250	347	1	58	1,360,432	46,367	46,368	2,892,763	12,943
garbage 250 1000	36	2	10	823,849	196,417	196,418	1,635,658	85,951
garbage 1000 1000	171	2	50	3,835,885	196,417	196,418	7,742,065	85,951

Table 5: Space performance of the shared heap architecture.

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
same 250 250	129	16	0	0	0	0	0	12.40	0.00	0.00	0.00
same 1000 250	2,044	244	0	182	0	0	0	11.94	0.00	8.90	0.00
same 250 1000	134	16	0	0	0	0	0	11.94	0.00	0.00	0.00
same 1000 1000	1,873	247	0	0	0	0	0	13.19	0.00	0.00	0.00
keep 250 250	604	17	0	12	0	0	0	2.81	0.00	1.99	0.00
keep 1000 250	4,136	248	0	288	0	0	0	6.00	0.00	6.96	0.00
keep 250 1000	1,983	17	0	13	0	0	0	0.86	0.00	0.66	0.00
keep 1000 1000	9,725	254	0	304	0	0	0	2.61	0.00	3.13	0.00
garbage 250 250	611	17	0	12	0	0	0	2.78	0.00	1.96	0.00
garbage 1000 250	4,068	249	0	288	0	0	0	6.12	0.00	7.08	0.00
garbage 250 1000	1,979	17	0	12	0	0	0	0.86	0.00	0.61	0.00
garbage 1000 1000	9,761	270	0	302	0	0	0	2.77	0.00	3.09	0.00

Table 6: Time performance of the hybrid architecture.

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
same 250 250	0	0	0	0	0	0	0	0
same 1000 250	3,528	0	0	0	0	0	0	0
same 250 1000	0	0	0	0	0	0	0	0
same 1000 1000	0	0	0	0	0	0	0	0
keep 250 250	276	0	0	0	0	0	0	0
keep 1000 250	5,478	0	0	0	0	0	0	0
keep 250 1000	276	0	0	0	0	0	0	0
keep 1000 1000	5,478	0	0	0	0	0	0	0
garbage 250 250	270	0	0	0	0	0	0	0
garbage 1000 250	5,412	0	0	0	0	0	0	0
garbage 250 1000	270	0	0	0	0	0	0	0
garbage 1000 1000	5,412	0	0	0	0	0	0	0

Table 7: Space performance of the hybrid architecture.

the benchmarks which might make the comparison harder. In this report a more thorough investigation has been made with the two architectures using the same surrounding system. The overall conclusions of this report are pretty much the same as in Mark Feeley's paper.

There is a system similar to the Hybrid Architecture implemented in the runtime system for Marmot [9, 5], a native code compiler for a subset of Java. Memory allocation in this system is managed the same way as in the Hybrid Architecture. There are two different allocation methods, one for local data and one for shared data. The escape analysis is made by the compiler. This is something that should be implemented in the Hybrid Architecture too, as is noted in the future work section. The garbage collection in this system is quite different from the one in the Hybrid Architecture. First all threads must rendez-vous to let one thread collect the shared data. After this collection is done each thread goes on collecting its private data. This means that all threads must garbage collect at the same time, even though it might not be needed in most threads. In the Hybrid Architecture all processes must wait for the global area to be collected in the same way, but after that collection each process goes about their business. Collection of the private heap only occurs when actually needed.

8 Conclusions and future work

Currently it seems that in the long run the SH is a better architecture than the PH. Nearly all benchmarks have better performance in the SH architecture. Just to strengthen the point, writing a benchmark that actually performed noticeably worse in the SH architecture was not an easy task. The mean garbage collection times are longer, but not the worse case garbage collection times. The fact that the garbage collection occurs very seldom in the SH architecture gives shorter execution times anyway.

The better memory utilization in the SH architecture is also something to strive for. Examples in the tables show programs that in the SH architecture only needs 12% of the memory required for the PH architecture. (**NETSim TMOS server** for instance in tables 3 and 5).

The main method to increase performance in the SH architecture is to reduce the root set in the general case of garbage collection. One optimization, the active process array, has already been described and implemented, another could be a generational stack scan as described in e.g. [3]. The idea is to only look at the part of the stack that has changed since the last collection. All data that was reachable from the stack at the last collection has already been placed in the old generation and there is no need to scan through these pointers again. Another possibility to reduce the root set is to change the notion of an active process. In the current scheme all processes that are scheduled are marked as active. This is clearly an over-approximation as some of these processes might not actually touch the heap. To move this activation of the process to the places where new data is actually stored in the heap would be quite a job since it means that it have to activate the process in most BIFs and in several instructions in the BEAM emulator. The activation is merely one instruction to set a flag, but considering all the places this would have to be done it might not give the desired effect after all. This would of course depend on how usual it is for processes to be scheduled without modifying data on the heap.

To make the Hybrid Architecture practical an escape analysis has to be implemented. This might turn out to be a quite complex problem since Erlang is a dynamically typed language.

As described in section 4.4 the heap still needs shrinking in the current implementation of the SH architecture. If the shrinking is to be removed, which has been suggested, all data has to be strictly allocated on the heap. This means that some new method of allocating memory for guard BIFs will have to be implemented.

References

[1] Staffan Blau and Jan Rooth. AXD 301—A new generation ATM switching system. *Ericsson Review*, 75(1):10–17, 1998.

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

[3] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'98*, pages 162–173, New York, N.Y., 1998. ACM Press.

[4] Marc Feeley. A case for the unified heap approach to Erlang memory management. In *Proceedings of the PLI'01 Erlang Workshop*, September 2001.

[5] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. *Software – Practice and Experience*, 30(3):199–232, March 2000.

[6] Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for automatic memory management*. John Wiley & Sons, 1996.

[7] Patrick Nilsson and Michael Persson. ANx — high-speed internet access. *Ericsson Review*, 75(1b):24–31, 1998.

[8] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–218, New York, N.Y., June 2000. ACM Press.

[9] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, pages 18–24, New York, N.Y., October 2000. ACM Press.

[10] Seved Torstendahl. Open Telecom Platform. *Ericsson Review*, 75(1):14–17, 1997. See also: http://www.erlang.se.

[11] R. Virding. A garbage collector for the concurrent real-time language Erlang. In Henry G. Baker, editor, *Proceedings of IWMM'95: International Workshop on Memory Management*, number 986 in LNCS, pages 343–354. Springer-Verlag, September 1995.

A How does the initial heap size affect the runtime?

A.1 Runtime vs. memory usage

As shown in Tables 8 through 15, the runtime in the PH architecture decreases when the initial heap size increases. It is quite obvious why; garbage collection does not occur as often on a larger heap. In the PH architecture however we cannot choose bigger heaps for all processes since it would require too much memory, and most of this allocated memory is unused, as shown in the columns "Max heap used/allocated". The SH architecture also shows better performance with bigger initial heap sizes (Tables 16 through 23), but most of the allocated memory is actually used in this scheme. The SH hardly ever gets near the total amount of memory that the private heap allocates, so it is easy to see that the global memory management of the SH utilizes memory better regardless of the initial heap size. In Tables 14 and 15 the memory requirement for **NETSim** became so high it was unable to complete its task.

In Table 14 where processes start with 121.393 words of heap the total garbage collection time increase dramatically in some cases, and even though the number of garbage collections only are about 1% of the numbers in Table 13 (starting with 10.946 words), the total run times have increased or are more or less the same. This probably depends more on the system itself being slower to work with when a lot of memory is allocated, due to swapping of virtual memory etc, than the time it takes to collect garbage since the total run time of those programs that do not do any garbage collection at all still are slower with bigger initial heap sizes.

All times are given in milliseconds.

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
procs 100 100	1,042	248	81	179	1	0	0	23.80	7.77	17.18	0.10
procs 100 1000	9,327	2,036	845	1,424	272	1	0	21.83	9.06	15.27	2.92
procs 1000 100	42,366	10,295	3,757	7,810	77	33	1	24.30	8.87	18.43	0.18
eddie	28,656	19	6	6,540	793	57	54	0.07	0.02	22.82	2.77
ring	2,165	819	205	0	0	0	0	37.83	9.47	0.00	0.00
life	5,299	1,830	712	523	2	5	0	34.53	13.44	9.87	0.04
BEAM compiler	10,579	84	54	1,752	342	129	100	0.79	0.51	16.56	3.23
NETSim TMOS	530,126	1,817	581	7,893	849	56	16	0.34	0.11	1.49	0.16
NETSim alarm server	138,474	5,525	2,553	8,332	588	110	59	3.99	1.84	6.02	0.42
NETSim coordinator	153,333	2,888	1,137	3,867	106	64	3	1.88	0.74	2.52	0.07

Table 8: Time performance of the Private Heap architecture, initial heap size = 233 words

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
procs 100 100	1,355	10	20	627,486	328,611	486,250	1,210,601	6,363
procs 100 1000	13,812	1,667	1,399	5,740,018	429,837	613,565	10,344,278	2,218,289
procs 1000 100	45,046	320	68	23,080,952	2,765,285	3,997,853	45,576,943	373,321
eddie	3,601	566	14,600	16,919,311	67,325	78,109	27,870,820	4,179,040
ring	4	1	3	3	11,742	41,650	0	9
life	28,690	82	56,814	119,326	33,139	52,834	469,079	3,592
BEAM compiler	1,620	21	757,836	2,864,006	1,363,278	1,375,659	4,469,462	1,787,055
NETSim TMOS	108,817	16,370	307,365	10,840,825	1,097,314	2,663,556	17,845,366	4,324,525
NETSim alarm server	121,938	6,342	533,281	18,452,823	2,031,703	2,720,603	35,331,680	3,197,814
NETSim coordinator	38,037	686	222,028	12,689,143	146,521	176,424	25,920,306	1,184,128

Table 9: Space performance of the Private Heap architecture, initial heap size = 233 words

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
procs 100 100	998	244	79	157	5	1	0	24.45	7.92	15.73	0.50
procs 100 1000	8,800	2,041	835	1,152	101	1	0	23.19	9.49	13.09	1.15
procs 1000 100	41,136	10,255	3,734	7,107	185	22	0	24.93	9.08	17.28	0.45
eddie	28,621	10	5	6,669	733	13	6	0.03	0.02	23.30	2.56
ring	2,235	809	202	0	0	0	0	36.20	9.04	0.00	0.00
life	4,752	1,790	751	98	0	0	0	37.67	15.80	2.06	0.00
BEAM compiler	10,153	80	53	1,717	346	123	104	0.79	0.52	16.91	3.41
NETSim TMOS	497,283	1,633	586	4,565	238	60	3	0.33	0.12	0.92	0.05
NETSim alarm server	132,895	5,501	2,608	5,588	393	52	3	4.14	1.96	4.20	0.30
NETSim coordinator	159,597	2,914	1,148	3,529	54	19	3	1.83	0.72	2.21	0.03

Table 10: Time performance of the Private Heap architecture, initial heap size = 1,597 words

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
procs 100 100	1,296	55	16	509,775	289,109	466,675	1,020,610	35,988
procs 100 1000	10,753	508	516	4,407,450	375,119	612,510	8,466,539	784,004
procs 1000 100	42,383	1,103	62	20,833,425	2,812,491	4,217,129	41,798,882	947,852
eddie	3,600	421	14,597	16,919,398	70,963	98,603	27,871,000	4,145,852
ring	3	0	2	17	23,149	167,219	35	0
life	3,988	14	7,546	18,213	178,751	212,922	66,817	452
BEAM compiler	1,235	17	659,930	2,905,709	1,368,853	1,398,504	4,637,730	1,884,585
NETSim TMOS	36,814	3,350	95,981	6,502,074	1,176,973	2,973,110	12,009,877	844,676
NETSim alarm server	36,380	4,133	175,363	13,786,182	2,093,604	2,922,095	27,714,149	2,080,544
NETSim coordinator	25,671	268	166,177	13,214,849	155,781	238,029	26,897,265	427,310

Table 11: Space performance of the Private Heap architecture, initial heap size = 1,597 words

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
procs 100 100	928	259	92	66	0	0	0	27.91	9.91	7.11	0.00
procs 100 1000	8,187	2,155	937	486	15	1	0	26.32	11.44	5.94	0.18
procs 1000 100	37,214	10,368	4,061	3,128	0	1	0	27.86	10.91	8.41	0.00
eddie	26,461	11	5	5,223	1,153	18	8	0.04	0.02	19.74	4.36
ring	2,126	772	191	0	0	0	0	36.31	8.98	0.00	0.00
life	4,465	1,719	752	10	0	0	0	38.50	16.84	0.22	0.00
BEAM compiler	9,949	80	54	1,443	320	164	98	0.80	0.54	14.50	3.22
NETSim TMOS	513,148	1,811	616	1,985	13	44	3	0.35	0.12	0.39	0.00
NETSim alarm server	130,026	6,240	3,008	4,805	52	68	8	4.80	2.31	3.70	0.04
NETSim coordinator	162,992	3,243	1,408	2,369	103	172	4	1.99	0.86	1.45	0.06

Table 12: Time performance of the Private Heap architecture, initial heap size = 10,946 words

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
procs 100 100	266	1	2	128,083	675,667	1,335,412	257,287	1,444
procs 100 1000	2,958	105	56	1,621,659	669,823	1,335,412	3,243,543	98,315
procs 1000 100	11,904	1	32	5,566,708	5,930,226	11,186,812	11,180,222	1,647
eddie	4,641	665	35,990	18,865,497	93,407	276,234	25,967,556	3,658,617
ring	0	0	0	0	57,899	251,758	0	0
life	487	13	548	3,676	1,130,592	1,324,466	9,488	416
BEAM compiler	654	12	540,042	2,375,252	1,378,824	1,576,135	3,893,405	1,645,644
NETSim TMOS	6,888	21	13,088	3,134,758	1,584,090	4.157.716	6,327,928	61,201
NETSim alarm server	5,303	100	31,628	10,035,163	2,397,539	3.610.751	21,645,217	304,454
NETSim coordinator	8,903	212	60,961	6,089,891	202,692	646,801	12,295,129	413,110

Table 13: Space performance of the Private Heap architecture, initial heap size = 10,946 words

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
procs 100 100	987	283	106	0	0	0	0	28.67	10.74	0.00	0.00
procs 100 1000	9,360	2,249	1,003	579	1	9	1	24.03	10.72	6.19	0.01
procs 1000 100	36,896	10,912	4,327	983	0	9	0	29.58	11.73	2.66	0.00
eddie	25,286	10	5	2,374	0	11	0	0.04	0.02	9.39	0.00
ring	2,307	783	191	0	0	0	0	33.94	8.28	0.00	0.00
life	4,819	1,774	768	0	0	0	0	36.81	15.94	0.00	0.00
BEAM compiler	10,158	90	63	1,451	269	104	68	0.89	0.62	14.28	2.65
NETSim TMOS	–	–	–	–	–	–	–	–	–	–	–
NETSim alarm server	–	–	–	–	–	–	–	–	–	–	–
NETSim coordinator	–	–	–	–	–	–	–	–	–	–	–

Table 14: Time performance of the Private Heap architecture, initial heap size = 121,393 words

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
procs 100 100	0	0	0	0	195,373	2,792,039	0	0
procs 100 1000	204	1	2	166,926	7,397,015	14,809,946	333,901	2,026
procs 1000 100	518	0	0	379,342	52,104,176	124,063,646	762,295	0
eddie	695	0	1,388	3,369,873	433,191	2,670,646	6,828,637	39
ring	0	0	0	0	132,430	2,792,039	0	0
life	0	0	0	0	160,105	2,792,039	0	0
BEAM compiler	95	5	308,804	2,075,775	898,732	3,502,686	3,347,311	1,407,497
NETSim TMOS	–	–	–	–	–	–	–	–
NETSim alarm server	–	–	–	–	–	–	–	–
NETSim coordinator	–	–	–	–	–	–	–	–

Table 15: Space performance of the Private Heap architecture, initial heap size = 121,393 words

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
procs 100 100	818	8	0	216	4	2	1	0.98	0.00	26.41	0.49
procs 100 1000	8,980	11	0	3,073	38	48	27	0.12	0.00	34.22	0.42
procs 1000 100	31,804	316	0	8,287	50	43	19	0.99	0.00	26.06	0.16
eddie	22,654	4	0	3,259	539	8	10	0.02	0.00	14.39	2.38
ring	1,672	337	0	0	0	0	0	20.16	0.00	0.00	0.00
life	3,773	479	0	347	2	6	1	12.70	0.00	9.20	0.05
BEAM compiler	8,583	0	0	983	152	57	76	0.00	0.00	11.45	1.77
NETSim TMOS	508,667	156	0	9,610	847	70	48	0.03	0.00	1.89	0.17
NETSim alarm server	138,830	382	0	20,605	523	208	71	0.28	0.00	14.84	0.38
NETSim coordinator	165,484	236	0	3,182	66	299	4	0.14	0.00	1.92	0.04

Table 16: Time performance of the Shared Heap architecture, initial heap size = 233 words

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
procs 100 100	145	3	212	1,145,852	46,368	46,368	2,250,266	47,969
procs 100 1000	103	3	256	7,951,960	514,229	514,229	15,886,533	219,415
procs 1000 100	885	8	805	29,568,367	514,229	514,229	59,143,224	410,990
eddie	3,331	333	24,368	16,502,190	46,368	46,368	25,978,713	5,258,174
ring	1	0	10	61	987	1,597	118	0
life	630	2	124,902	286,125	28,656	28,657	1,106,945	8,544
BEAM compiler	1,201	4	696,786	2,474,088	1,682,835	1,682,836	4,305,868	1,760,324
NETSim TMOS	10,541	186	1,664,133	14,640,505	196,417	196,418	17,284,630	7,227 354
NETSim alarm server	18,647	65	26,183,596	16,871,158	317,811	317,811	28,565,183	4,009,965
NETSim coordinator	14,435	23	875,030	8,748,892	121,393	121,393	17,566,720	592,325

Table 17: Space performance of the Shared Heap architecture, initial heap size = 233 words

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
procs 100 100	848	8	0	225	4	3	2	0.94	0.00	26.53	0.47
procs 100 1000	8,926	12	0	3,098	52	65	30	0.13	0.00	34.71	0.58
procs 1000 100	31,935	357	0	8,259	51	63	18	1.12	0.00	25.86	0.16
eddie	21,998	3	0	3,217	498	7	6	0.01	0.00	14.62	2.26
ring	1,653	331	0	0	0	0	0	20.02	0.00	0.00	0.00
life	3,624	473	0	343	1	4	1	13.05	0.00	9.46	0.03
BEAM compiler	8,936	0	0	989	153	72	69	0.00	0.00	11.07	1.71
NETSim TMOS	578,173	133	0	7,999	779	120	64	0.02	0.00	1.38	0.13
NETSim alarm server	137,053	318	0	19,892	498	85	35	0.23	0.00	14.51	0.36
NETSim coordinator	156,949	224	0	2,890	72	96	13	0.14	0.00	1.84	0.05

Table 18: Time performance of the Shared Heap architecture, initial heap size = 1,597 words

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
procs 100 100	145	3	212	1,145,852	46,368	46,368	2,250,266	47,969
procs 100 1000	112	4	373	8,308,507	514,229	514,229	16,516,839	338,380
procs 1000 100	885	8	805	29,568,367	514,229	514,229	59,143,224	410,990
eddie	3,332	333	24,355	16,507,784	46,368	46,368	25,981,945	5,253,169
ring	1	0	9	12	4,179	6,765	32	0
life	636	1	127,319	289,592	17,708	17,711	1,119,176	8,564
BEAM compiler	1,269	4	691,237	2,441,439	1,682,832	1,682,836	4,209,747	1,180,802
NETSim TMOS	9,473	172	1,534,496	14,594,696	196,417	196,418	17,824,843	6,723,331
NETSim alarm server	17,441	66	25,212,644	17,061,925	317,811	317,811	28,340,806	4,370,656
NETSim coordinator	12,846	20	798,886	8,496,614	121,393	121,393	17,168,736	552,859

Table 19: Space performance of the Shared Heap architecture, initial heap size = 1,597 words

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
procs 100 100	742	9	0	139	5	4	2	1.21	0.00	18.73	0.67
procs 100 1000	8,966	10	0	3,339	59	76	31	0.11	0.00	37.24	0.66
procs 1000 100	31,693	335	0	8,174	47	62	19	1.06	0.00	25.79	0.15
eddie	21,902	2	0	3,456	356	13	9	0.01	0.00	15.78	1.63
ring	1,635	328	0	0	0	0	0	20.06	0.00	0.00	0.00
life	3,493	467	0	176	1	1	1	13.37	0.00	5.04	0.03
BEAM compiler	8,560	0	0	945	161	64	51	0.00	0.00	11.04	1.88
NETSim TMOS	502,729	139	0	8,775	934	68	41	0.03	0.00	1.75	0.19
NETSim alarm server	146,072	295	0	30,517	570	124	45	0.20	0.00	20.89	0.39
NETSim coordinator	155,510	205	0	1,732	30	37	4	0.13	0.00	1.11	0.02

Table 20: Time performance of the Shared Heap architecture, initial heap size = 10,946 words

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
procs 100 100	97	3	133	719,523	75,025	75,025	1,406,471	54,427
procs 100 1000	130	6	162	9,325,503	514,229	514,229	18,245,249	516,995
procs 1000 100	849	7	805	29,071,002	514,229	514,229	58,193,774	381,359
eddie	3,600	200	26,267	17,513,842	46,368	46,368	26,919,506	3,587,600
ring	0	0	0	0	2,392	10,946	0	0
life	347	0	68,546	157,549	28,656	28,657	613,709	0
BEAM compiler	783	5	585,555	2,483,086	1,346,264	1,346,269	3,948,724	1,405,146
NETSim TMOS	10,421	205	1,974,496	15,148,233	317,810	317,811	16,185,966	8,230,296
NETSim alarm server	25,676	67	43,633,622	15,060,866	317,811	317,811	24,448,981	4,418,664
NETSim coordinator	7,494	12	539,556	4,588,232	121,392	121,393	9,404,429	323,489

Table 21: Space performance of the Shared Heap architecture, initial heap size = 10,946 words

Program	Time spent in					Max time in		% of time spent in			
	total	send	copy	minor	major	minor	major	send	copy	minor	major
procs 100 100	671	8	0	54	3	2	3	1.19	0.00	8.05	0.45
procs 100 1000	8,821	11	0	3,207	57	63	34	0.12	0.00	36.36	0.65
procs 1000 100	31,157	327	0	7,578	40	56	17	1.05	0.00	24.32	0.13
eddie	19,369	3	0	792	0	8	0	0.02	0.00	4.09	0.00
ring	1,763	343	0	0	0	0	0	19.46	0.00	0.00	0.00
life	3,628	505	0	17	0	1	0	13.92	0.00	0.47	0.00
BEAM compiler	9,640	0	0	944	113	97	45	0.00	0.00	9.79	1.17
NETSim TMOS	510,750	145	0	9,914	1,323	110	41	0.03	0.00	1.94	0.26
NETSim alarm server	155,859	311	0	34,470	495	456	26	0.20	0.00	22.12	0.32
NETSim coordinator	153,700	273	0	556	11	38	3	0.18	0.00	0.36	0.01

Table 22: Time performance of the Shared Heap architecture, initial heap size = 121,393 words

Program	Number of				Max heap		Sum of live data in	
	minor	major	ptr to old	ptr to young	used	allocated	minor	major
procs 100 100	28	1	46	263,626	121,393	121,393	519,015	17,423
procs 100 1000	120	4	164	9,008,886	514,229	514,229	17,708,317	453,429
procs 1000 100	633	5	454	27,228,782	514,229	514,229	54,524,330	337,334
eddie	700	0	7,697	3,442,013	121,393	121,393	6,973,607	0
ring	0	0	0	0	2,531	121,393	0	0
life	26	0	4,908	13,344	121,392	121,393	47,928	0
BEAM compiler	102	3	341,173	1,999,320	832,040	832,040	3,438,580	849,244
NETSim TMOS	13,610	301	2,139,043	19,439,329	121,393	196,418	18,018,576	12,044,022
NETSim alarm server	30,183	66	49,424,576	8,379,335	121,393	196,418	10,259,033	4,134,546
NETSim coordinator	1,527	4	139,258	466,681	121,393	121,393	802,457	100,034

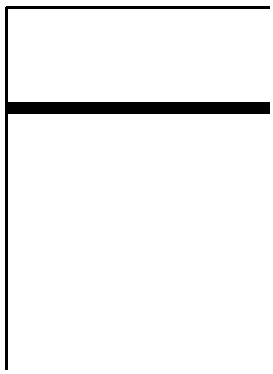
Table 23: Space performance of the Shared Heap architecture, initial heap size = 121,393 words

B Message statistics

These diagrams show the number of messages of different sizes sent. Each diagram corresponds to one execution of the given benchmark. Most interesting is it of course to compare number of messages smaller than or equal to 64 words versus bigger sizes since this is where one can expect to get significantly better results when avoiding message copying.

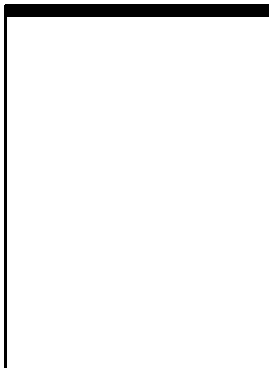
procs(100,100)

size in words	number	percentage
0–64	(97)	
65–99	(2)	
100–199	(2)	
200–299	(15251)	
300–399	(0)	
400–499	(0)	
500–599	(0)	
600–699	(0)	
700–799	(0)	
800–899	(0)	
900–999	(0)	
>= 1000	(0)	



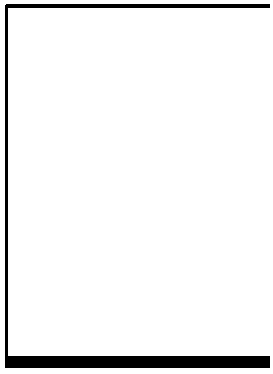
eddie

size in words	number	percentage
0–64	(2117)	
65–99	(1)	
100–199	(3)	
200–299	(0)	
300–399	(0)	
400–499	(0)	
500–599	(0)	
600–699	(0)	
700–799	(0)	
800–899	(0)	
900–999	(0)	
>= 1000	(0)	



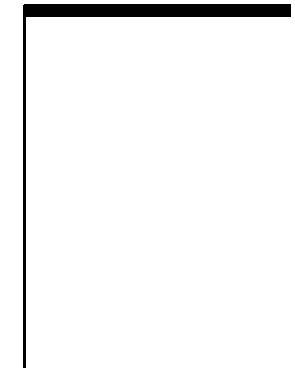
procs(100,1000)

size in words	number	percentage
0–64	(97)	
65–99	(2)	
100–199	(1)	
200–299	(1)	
300–399	(0)	
400–499	(0)	
500–599	(0)	
600–699	(0)	
700–799	(0)	
800–899	(0)	
900–999	(0)	
>= 1000	(15251)	



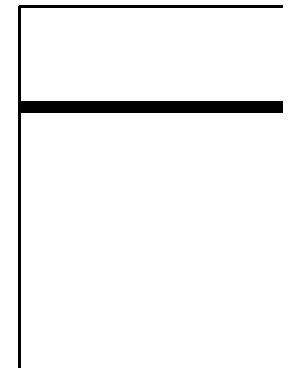
ring

size in words	number	percentage
0–64	(501092)	
65–99	(1)	
100–199	(0)	
200–299	(0)	
300–399	(0)	
400–499	(0)	
500–599	(0)	
600–699	(0)	
700–799	(0)	
800–899	(0)	
900–999	(0)	
>= 1000	(0)	



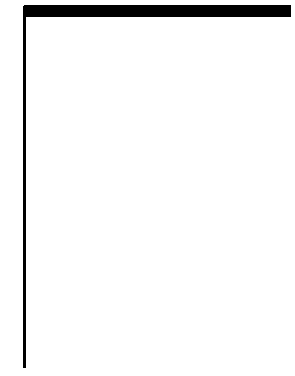
procs(1000,100)

size in words	number	percentage
0–64	(997)	
65–99	(2)	
100–199	(1)	
200–299	(601602)	
300–399	(0)	
400–499	(0)	
500–599	(0)	
600–699	(0)	
700–799	(0)	
800–899	(0)	
900–999	(0)	
>= 1000	(0)	



life

size in words	number	percentage
0–64	(800395)	
65–99	(1)	
100–199	(0)	
200–299	(0)	
300–399	(0)	
400–499	(0)	
500–599	(0)	
600–699	(0)	
700–799	(0)	
800–899	(0)	
900–999	(0)	
>= 1000	(0)	



BEAM compiler

size in words	number	percentage
0–64	(453)	
65–99	(16)	
100–199	(136)	
200–299	(45)	
300–399	(86)	
400–499	(15)	
500–599	(14)	
600–699	(6)	
700–799	(6)	
800–899	(0)	
900–999	(2)	
>= 1000	(22)	

NETSim alarm server

size in words	number	percentage
0–64	(228460)	
65–99	(197)	
100–199	(36300)	
200–299	(13766)	
300–399	(4738)	
400–499	(114)	
500–599	(227)	
600–699	(87)	
700–799	(127)	
800–899	(7)	
900–999	(44)	
>= 1000	(4686)	

NETSim TMOS

size in words	number	percentage
0–64	(72965)	
65–99	(175)	
100–199	(18858)	
200–299	(31227)	
300–399	(37)	
400–499	(48)	
500–599	(23)	
600–699	(33)	
700–799	(5)	
800–899	(7)	
900–999	(4)	
>= 1000	(521)	

NETSim coordinator

size in words	number	percentage
0–64	(158005)	
65–99	(9880)	
100–199	(30816)	
200–299	(10931)	
300–399	(231)	
400–499	(16)	
500–599	(15)	
600–699	(16)	
700–799	(12)	
800–899	(14)	
900–999	(13)	
>= 1000	(5586)	

C New BIFs for benchmarking

A number of new BIFs were implemented to ease the benchmark of the system. All these BIFs are placed in the `hipe_bifs` module, so HiPE must be enabled to use them. The new BIFs are:

system_timer_get/0 -> {`Minutes, Seconds, Milliseconds, Microseconds`}.
Returns the internal system timer.

system_timer_clear/0 -> `true`.
Reset the internal system timer.

gc_timer_get/0 -> {`Major_GC, Minor_GC, Max_Major, Max_minor`}.
Returns the garbage collection timers. Each timer is a tuple of the form: {Minutes, Seconds, Milliseconds, Microseconds}

gc_timer_clear/0 -> `true`.
Reset all garbage collection timers.

send_timer_get/0 -> {`Minutes, Seconds, Milliseconds, Microseconds`}.
Returns the send timer. This is the total time spent in the send-command and includes the time for copying the message to the receiver.

send_timer_clear/0 -> `true`.
Resets the send timer.

copy_timer_get/0 -> {`Minutes, Seconds, Milliseconds, Microseconds`}.
Returns the copy timer. This is the time spent copying messages to the receiver.

copy_timer_clear/0 -> `true`.
Resets the copy timer.

gc_info/0 -> {`Major_GCs, Minor_GCs, Live_in_Major, Live_in_Minor, Ptrs_to_Old, Ptrs_to_Young`}.
Returns information about the garbage collector. `Major_GCs` and `Minor_GCs` are the total number of major and minor garbage collections respectively. `Live_in_*` is the sum of live data in the two garbage collections. `Ptrs_to_*` is the number of pointers that point from live data into the old generation and the young generation.

clear_gc_info/0 -> `true`.
Reset the garbage collection counters.

bench_info/0 -> {`Live_Processes, Spawned_Processes, Messages_Sent, Current_Heap_Size, Current_Allocated_Heap, Max_Heap_Size, Max_Allocated_Heap`}.
Returns information about the Erlang node. `Live_Processes` are the current number of processes, `Spawned_Processes` are the total number spawned since the last reset. Heap size refers to total amount of used heap, this does not include the stack.

clear_bench_info/0 -> `true`.
Reset the bench info.

message_info/0 -> `true`.
Prints a diagram over message sizes for the messages sent in the system since start or the latest clear_bench_info/0.

All BIFs return `false` if the system is compiled without benchmarking enabled. The `__BENCHMARK__` define is located in the file `erts/emulator/beam/benchmark.h`. The benchmark code is written in such a way that it is to affect the runtime as little as possible.

D Benchmark Source Code

D.1 procs

```
-module(procs).
-export([start/3,child/4]).

start(Proc,Times,Size) ->
    Pid = spawn(procs,child,[self(),Proc - 1,Times,Size]),
    Pid!{ Times, build_list(Size,[]) },
    leader_loop(Pid,true,Size).

child(Leader,0,Times,Size) ->
    Leader!{ Times, build_list(Size,[]) },
    loop(Leader,Size);
child(Leader,Number,Times,Size) ->
    Pid = spawn(procs,child,[Leader,Number - 1,Times,Size]),
    Pid!{ Times, build_list(Size,[]) },
    loop(Pid,Size).

loop(Receiver,Size) ->
    receive
        { 0, Block } ->
            Receiver!{ 0, build_list(Size,[]) },
            loop(Receiver,Size);
        { Number, Block } ->
            Receiver!{ Number - 1, build_list(Size,[]) },
            loop(Receiver,Size);
        stop ->
            Receiver!stop,
            ok
    end.

leader_loop(Receiver,Sending,Size) ->
    receive
        { 0, Block } ->
            if
                Sending == true ->
                    Receiver!stop,
                    leader_loop(Receiver,false,Size);
                true ->
                    leader_loop(Receiver,false,Size)
            end;
        { Number, Block } ->
            if
                Sending == true ->
                    Receiver!{ Number - 1, build_list(Size,[]) },
                    leader_loop(Receiver,Sending,Size);
                true ->
                    leader_loop(Receiver,Sending,Size)
            end;
        stop ->
            ok
    end.

build_list(0,List) -> List;
```

```
build_list(N,List) -> build_list(N-1,[42 | List]).
```

D.2 nagbif

```
-module(nagbif).
-author('jewi7435@csd.uu.se').
-export([name/0,args/0,test/1,garbage_child/4,same_child/4,keep_child/4]).

name() -> nagbif.

args() -> [{100,100,100,garbage},
           {100,100,1000,garbage},
           {1000,100,100,garbage},
           {1000,100,1000,garbage},
           {100,100,100,same},
           {100,100,1000,same},
           {1000,100,100,same},
           {1000,100,1000,same},
           {100,100,100,keep},
           {100,100,1000,keep},
           {1000,100,100,keep},
           {1000,100,1000,keep}].


test({Number,Times,Size,garbage}) ->
    Pid = spawn(nagbif,garbage_child,[self(),Number - 2,Times,Size]),
    hipe_bifs:send(Pid,small_tuple(Times - 1,build_list(Size,[]))),
    garbage_coord_loop(Pid,Size,Number);
test({Number,Times,Size,keep}) ->
    Pid = spawn(nagbif,keep_child,[self(),Number - 2,Times,Size]),
    hipe_bifs:send(Pid,small_tuple(Times - 1,build_list(Size,[]))),
    keep_coord_loop(Pid,Size,[],Number);
test({Number,Times,Size,same}) ->
    Message = build_list(Size,[]),
    Args = hipe_bifs:cons(self(),
                          hipe_bifs:cons(Number - 2,
                                         hipe_bifs:cons(Times,
                                                        hipe_bifs:cons(Message,
                                                                       [])))),
    Pid = spawn(nagbif,same_child,Args),
    hipe_bifs:send(Pid,small_tuple(Times - 1,Message)),
    same_coord_loop(Pid,Number).

garbage_child(Leader,0,Times,Size) ->
    hipe_bifs:send(Leader,small_tuple(Times - 1,build_list(Size,[]))),
    garbage_loop(Leader,Size);
garbage_child(Leader,Number,Times,Size) ->
    Pid = spawn(nagbif,garbage_child,[Leader,Number - 1,Times,Size]),
    hipe_bifs:send(Pid,small_tuple(Times - 1,build_list(Size,[]))),
    garbage_loop(Pid,Size).

garbage_loop(Receiver,Size) ->
    receive
        stop -> hipe_bifs:send(Receiver,stop);
        { 0, _ } ->
```

```
            hipe_bifs:send(Receiver,small_tuple(0,empty)),
            garbage_loop(Receiver,Size);
        { Times, Message } ->
            hipe_bifs:send(Receiver,small_tuple(Times - 1,
                                                build_list(Size,[]))),
            garbage_loop(Receiver,Size)
    end.

garbage_coord_loop(Receiver,_,0) ->
    hipe_bifs:send(Receiver,stop),
    receive
        stop -> ok
    end;
garbage_coord_loop(Receiver,Size,Number) ->
    receive
        { 0, _ } -> garbage_coord_loop(Receiver,Size,Number - 1);
        { Times, Message } ->
            hipe_bifs:send(Receiver,small_tuple(Times - 1,
                                                build_list(Size,[]))),
            garbage_coord_loop(Receiver,Size,Number)
    end.


keep_child(Leader,0,Times,Size) ->
    hipe_bifs:send(Leader,small_tuple(Times - 1,build_list(Size,[]))),
    keep_loop(Leader,Size,[]);
keep_child(Leader,Number,Times,Size) ->
    Pid = spawn(nagbif,keep_child,[Leader,Number - 1,Times,Size]),
    hipe_bifs:send(Pid,small_tuple(Times - 1,build_list(Size,[]))),
    keep_loop(Pid,Size,[]).

keep_loop(Receiver,Size,Storage) ->
    receive
        stop -> hipe_bifs:send(Receiver,stop);
        { 0, _ } ->
            hipe_bifs:send(Receiver,small_tuple(0,empty)),
            keep_loop(Receiver,Size,Storage);
        { Times, Message } ->
            hipe_bifs:send(Receiver,small_tuple(Times - 1,
                                                build_list(Size,[]))),
            keep_loop(Receiver,Size,hipe_bifs:cons(Message,Storage))
    end.

keep_coord_loop(Receiver,_,_,0) ->
    hipe_bifs:send(Receiver,stop),
    receive
        stop -> ok
    end;
keep_coord_loop(Receiver,Size,Storage,Number) ->
    receive
        { 0, _ } -> keep_coord_loop(Receiver,Size,Storage,Number - 1);
        { Times, Message } ->
            hipe_bifs:send(Receiver,small_tuple(Times - 1,Message)),
            keep_coord_loop(Receiver,
                            Size,
                            hipe_bifs:cons(Message,Storage),
```

```
                            Number)
    end.

same_child(Leader,0,Times,Message) ->
    hipe_bifs:send(Leader,small_tuple(Times - 1,Message)),
    same_loop(Leader);
same_child(Leader,Number,Times,Message) ->
    Args = hipe_bifs:cons(Leader, hipe_bifs:cons(Number - 1,
                          hipe_bifs:cons(Times, hipe_bifs:cons(Message,[])))),
    Pid = spawn(nagbif,same_child,Args),
    hipe_bifs:send(Pid,small_tuple(Times - 1, Message)),
    same_loop(Pid).

same_loop(Receiver) ->
    receive
        stop -> hipe_bifs:send(Receiver,stop);
        { 0, _ } ->
            hipe_bifs:send(Receiver,small_tuple(0,empty)),
            same_loop(Receiver);
        { Times, Message } ->
            hipe_bifs:send(Receiver,small_tuple(Times - 1,Message)),
            same_loop(Receiver)
    end.

same_coord_loop(Receiver,0) ->
    hipe_bifs:send(Receiver,stop),
    receive
        stop -> ok
    end;
same_coord_loop(Receiver,Number) ->
    receive
        { 0, _ } -> same_coord_loop(Receiver,Number - 1);
        { Times, Message } ->
            hipe_bifs:send(Receiver,small_tuple(Times - 1,Message)),
            same_coord_loop(Receiver,Number)
    end.


build_list(0,List) -> List;
build_list(N,List) -> build_list(N-1,hipe_bifs:cons(42,List)).


small_tuple(H,T) ->
    hipe_bifs:tuple(hipe_bifs:localcons(H,hipe_bifs:localcons(T,[]))).
```