Heap Architectures for Concurrent Languages using Message Passing

Erik Johansson
Computing Science Dept.
Uppsala University, Sweden
happi@csd.uu.se

Konstantinos Sagonas
Computing Science Dept.
Uppsala University, Sweden
kostis@csd.uu.se

Jesper Wilhelmsson
Computing Science Dept.
Uppsala University, Sweden
jesperw@csd.uu.se

ABSTRACT

We discuss alternative heap architectures for languages that rely on automatic memory management and implement concurrency through asynchronous message passing. We describe how interprocess communication and garbage collection happens in each architecture, and extensively discuss the tradeoffs that are involved. In an implementation setting (the Erlang/OTP system) where the rest of the runtime system is unchanged, we present a detailed experimental comparison between these architectures using both synthetic programs and large commercial products as benchmarks.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures, dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection), runtime environments*; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Languages, Performance, Measurement

Keywords

Runtime systems, concurrent languages, message passing, Erlang, garbage collection

1. INTRODUCTION

In recent years, concurrency as a form of abstraction has become increasingly popular, and many modern programming languages (such as Occam, CML, Caml, Erlang, Oz, Java, and C#) come with some form of built-in support for concurrent processes (or threads). Depending on the concurrency model of the language, interprocess communication takes place either using asynchronous message passing or through (synchronized) shared structures. These languages typically also require support for automatic memory management, usually implemented using a garbage collector. By now, many different garbage collection techniques have been proposed and their characteristics are well-known; see [17, 24] for comprehensive treatments on the subject. A less treated, albeit key issue in the design of a concurrent language implementation is that of the runtime system's memory architecture. It is clear that there exist many different ways of structuring the architecture of the runtime system, each having its pros and cons. Despite its importance, this issue has received remarkably little attention in the literature. Although many of its aspects are folklore, to our knowledge there has never been an in-depth investigation of the performance tradeoffs that are involved based on a non-toy implementation where the rest of the system remains unchanged. The main aim of this paper is to fill this gap. In particular, we systematically examine and experimentally evaluate the tradeoffs of different heap architectures for concurrent languages focusing on those languages where process communication happens through message passing.

More specifically, in this paper we focus on three different runtime system architectures for concurrent language implementations: One where each process allocates and manages its private memory area and all messages have to be copied between processes, one where all processes share the same heap, and a hybrid architecture where each process has a private heap for local data but where a shared heap is used for data sent as messages. For each architecture, we discuss the architectural impact on the speed of interprocess communication and garbage collection. To evaluate the performance of these architectures, we have implemented them in an otherwise unchanged, industrial-strength, Erlang/OTP system. This system was chosen in part due to our involvement in its development (cf. the HiPE native code compiler [16]), but more importantly due to the existence of real-world highly concurrent programs which can be used as benchmarks. By instrumenting this system, we have been able to measure the impact of the architecture both on large commercial applications, and on concurrent synthetic benchmarks constructed to examine the tradeoffs that are involved.

The rest of the paper is structured as follows: We begin by presenting aspects of Erlang which are relevant for our work, and by a brief overview of previous work on memory management of concurrent language implementations. Then, in Section 3, we describe a memory architecture where each process allocates and manages its own memory area. In Section 4 we present the architecture of a system with

ISMM'02, June 20-21, 2002, Berlin, Germany.

only one heap which is shared among all processes. Then in Section 5 we develop a hybrid memory architecture with a shared memory area for all messages and private heaps for data which is private to each process. An extensive performance evaluation of these architectures is presented in Section 6. The paper ends with some concluding remarks which include directions for future work.

2. PRELIMINARIES & RELATED WORK

2.1 Erlang and Erlang/OTP

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 support for multiple platforms [2]. Erlang was designed to ease the programming of large soft real-time control systems commonly developed by the telecommunications industry. It has so far been used quite successfully both by Ericsson and by other companies around the world to develop large commercial applications.

Erlang's basic data types are atoms, numbers (floats and arbitrary precision integers), and process identifiers; compound data types are lists and tuples. A notation for objects (records in the Erlang lingo) is supported but the underlying implementation of records is as tuples. To allow efficient implementation of telecommunication protocols, Erlang also includes a binary data type (a vector of byte-sized data). 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 combined with the use of flat guards in the head of the rule. Since recursion is the only means to express iteration in Erlang, tail call optimization is a required feature of Erlang implementations.

Processes in Erlang are extremely light-weight (lighter than OS threads), 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.

Erlang is often used in "five nines" high-availability (i.e., 99.999% of the time available) systems, where down-time 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. Consequently, an Erlang system comes with support for upgrading code while the system is running, a mechanism known as *hot-code loading*.

The Erlang language is small, but an Erlang system comes with libraries containing a large set of built-in functions for various tasks. With the *Open Telecom Platform* (OTP) middleware [22], Erlang is further extended with a library of standard components for telecommunication applications (real-time databases, servers, state machines, process monitors, tools for load balancing), standard interfaces such as CORBA, and a variety of communication protocols (e.g., HTTP, FTP, etc.).

2.2 Memory management in Erlang and other concurrent languages

As in other functional languages, memory management in Erlang is a responsibility of the runtime system and happens through garbage collection. The soft real-time concerns of the language call for bounded-time garbage collection techniques. One such technique, based on a mark-and-sweep algorithm taking advantage of the fact that the heap in an Erlang system is *unidirectional* (i.e., is arranged so that the pointers point in only one direction), has been proposed by Armstrong and Virding in [1], but imposes a significant overhead and was never fully implemented. In practice, in a tuned Erlang system with a generational copying garbage collector, garbage collection latency is usually low (less than 10 milliseconds) as most processes are short-lived or small in size. Longer pauses are quite infrequent. However, a blocking collector provides no guarantees for the real-time responsiveness that some applications may desire.

In the context of strict, concurrent functional language implementations, there has been work that aims at achieving low garbage collection latency without paying the full price in performance that a guaranteed real-time garbage collector usually requires. Notable among them is the work of Doligez and Leroy [10] who combine a fast, asynchronous copying collector for the thread-specific young generations with a non-disruptive concurrent mark-and-sweep collector for the old generation (which is shared among all threads). The result is a quasi-real-time collector for Concurrent Caml Light. Also, Larose and Feeley in [12] describe the design of a near-real-time compacting collector in the context of the Gambit-C Scheme compiler. This garbage collector was intended to be used in the Etos (Erlang to Scheme) system, but to the best of our knowledge, it has not yet made it to an Etos distribution. In order to achieve low garbage collection pause times, concurrent or real-time multiprocessor collectors have also been proposed; both for (concurrent) variants of ML [14, 18, 7], and recently for Java; see e.g. [4, 13].

An issue which is to a large extent orthogonal to that of the garbage collection technique used is that of the memory organization of a concurrent system: Should one use an architecure which facilitates sharing, or one that requires copying of data? The issue often attracts heated debates both in the programming language implementation community and elsewhere.[1] Traditionally, operating systems allocate memory on a per-process basis. The architecture of KaffeOS [3] uses process-specific heaps for Java processes and shared heaps for data shared among processes. Objects in the shared heaps are not allowed to reference objects in process-specific heaps and this restriction is enforced with page protection mechanisms. In the context of a multi-threaded Java implementation, the same architecture is also

[1] For example, in the networking community an issue which is related to those discussed in this paper is whether packets will be passed up and down the stack by reference or by copying. Also, during the mid-80's the issue of whether files can be passed in shared memory was investigated by the operating systems community in the context of user-level kernel extensions.

proposed by Steensgaard [21] who argues for thread-specific heaps for thread-specific data and a shared heap for shared data. The paper reports statistics showing that, in a small set of multi-threaded Java programs, there are very few conflicts between threads, but provides no experimental comparison of this memory architecture with another.

Till the fall of 2001, the Ericsson Erlang implementation had exclusively a memory architecture where each process allocates and manages its own memory area. We describe this architecture in Section 3. The main reason why this architecture was chosen is that it is believed it results in lower garbage collection latency. Wanting to investigate the validity of this belief, we have been working on a shared heap memory architecture for Erlang processes. We describe this architecture in Section 4; it is already included in the Erlang/OTP release. Concurrently with our work, Feeley [11] argued the case for a unified memory architecture for Erlang, an architecture where all processes get to share the same stack and heap. This is the architecture used in the Etos system that implements concurrency through a *call/cc* (*call-with-current-continuation*) mechanism. The case for the architecture used in Etos is argued convincingly in [11], but on the other hand it is very difficult to draw conclusions from the small experimental comparison between Etos and the Ericsson Erlang/OTP implementation due to the differences in performance between the two systems, the lack of experimental evaluation using large programs, and, more importantly, due to the big differences in the parameters (e.g., initial sizes of memories, garbage collector settings) that are involved. As mentioned, one of our aims is to compare memory architectures for concurrent languages in a setting where the rest of the system is unchanged.

Assumptions. Throughout the paper, for simplicity of presentation, we make the following assumptions: 1) the system is running on a uniprocessor, 2) the heap garbage collector is similar to the collector currently used in Erlang/OTP: a Cheney-style semi-space stop and copy collector [6] with two generations, and 3) message passing and garbage collection cannot be interrupted by the scheduler. For a more detailed description of the garbage collector in Erlang/OTP refer to [23].

3. AN ARCHITECTURE WITH PRIVATE HEAPS

The first memory architecture we examine is *process-centric.* In this architecture, each process allocates and manages its own memory area which typically includes a process control block (PCB), private stack, and private heap. Other memory areas, e.g. a space for large objects, might also exist either on a per-process basis or as a global area.

This is the default architecture of the Erlang/OTP R8 system, the version of Erlang released by Ericsson in the fall of 2001. The stack is used for function arguments, return addresses, and local variables. Compound terms such as lists, tuples, and objects which are larger than a machine word such as floating point numbers and arbitrary precision integers (bignums) are stored on the heap. One way of organizing the memory areas is with the heap co-located with the stack (i.e., the stack and the heap growing towards each other). The advantage of doing so, is that stack and heap overflow tests become cheap, just a comparison between the

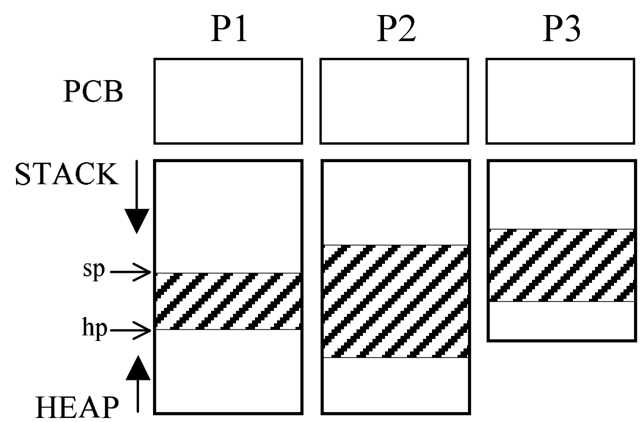

Figure 1: Memory architecture with private heaps.

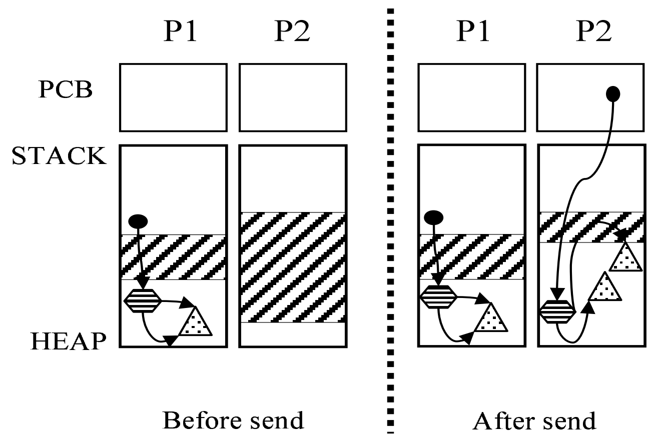

Figure 2: Message passing in a private heap system.

stack and heap pointers which can usually be kept in machine registers. A disadvantage is that expansion or relocation of the heap or stack involves both areas. As mentioned, Erlang also supports large vectors of byte-sized data (binaries). These are not stored on the heap; instead they are reference-counted and stored in a separate global memory area. Henceforth, we ignore the possible existence of a large object space as the issue is completely orthogonal to our discussion.

Figure 1 shows an instance of this architecture when three processes (P1, P2, and P3) are present; shaded areas represent unused memory.

Process communication. Message passing is performed by copying the term to be sent from the heap of the sender to the heap of the receiver, and then inserting a pointer to the message in the mailbox of the receiver which is contained in its PCB; see Figure 2. As shown in the figure, a local data structure might share the same copy of a subterm, but when that data structure is sent to another process each subterm will be copied separately. As a result, the copied message occupies more space than the original. However, message expansion due to loss of sharing is quite rare in practice.[2] This phenomenon could be avoided by using some marking

[2] In particular it does not occur in our benchmarks.

technique and forwarding pointers, but note that doing so would make the message passing operation even slower.

Garbage collection. When a process runs out of heap (or stack) space, the process's private heap is garbage collected. In this memory architecture, the root set of the garbage collection is the process' stack and mailbox. Recall that a two-generational (young and old) Cheney-style stop-and-copy collector is being used. A new heap, local to a process, where live data will be placed, is allocated at the beginning of the collection. The old heap contains a high water mark (the top of the heap after the last garbage collection) and during a minor collection data below this mark is forwarded to the old generation while data above the mark is put on the new heap. During a major collection the old generation is also collected to the new heap. At the end of the garbage collection the stack is moved to the area containing the new heap and the old heap is freed.

In a system which is not multi-threaded, like the current Erlang/OTP system, the mutator will be stopped and all other processes will also be blocked during garbage collection.

Pros and cons. According to its advocates, this design has a number of advantages:

+ No cost memory reclamation – When a process terminates, its memory can be freed directly without the need for garbage collection. Thus, one can use processes for some simple form of memory management: a separate process can be spawned for computations that will produce a lot of garbage.

+ Small root sets – Since each process has its own heap, the root set for a garbage collection is the stack and mailbox of the current process only. This is expected to help in keeping the GC stop times short. However, as noted, without a real-time garbage collector there is no guarantee for this.

+ Improved cache locality – Since each process has all its data in one contiguous (and often small) stack/heap memory area, the cache locality for each process is expected to be good.

+ Cheaper tests for stack/heap overflow – With a per-process heap, the heap and stack overflow tests can be combined and fewer frequently accessed pointers need to be kept in machine registers.

Unfortunately this design also has some disadvantages:

– Costly message passing – Messages between processes must be copied between the heaps. The cost of interprocess communication is proportional to the size of the message. In some implementations, the message might need to be traversed more than once: one pass to calculate its size (so as to avoid overflow of the receiver's heap and trigger its garbage collection or expansion if needed) and another to perform the actual copy.

– More space needs – Since messages are copied, they require space on each heap they are copied to. As shown, if the message contains the same subterm several times, there can even be non-linear growth when sending messages. Also, if a (sub-)term is sent back and forth between two processes a new copy of the term is created for each send—even though the term already resides on the appropriate heap before the send.

– High memory fragmentation – A process cannot utilize the memory (e.g., the heap) of another process even if there are large amounts of unused space in that memory area. This typically implies that processes can allocate only a small amount of memory by default. This in turn usually results in a larger number of calls to the garbage collector.

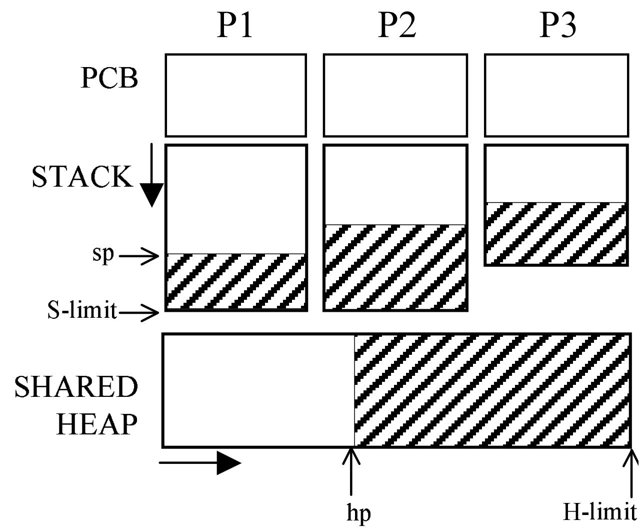

Figure 3: Memory architecture with shared heap.

From a software development perspective, a process-centric memory architecture can have an impact on how programs are written. For example, due to the underlying implementation which until recently was exclusively based on the memory architecture described in this section, the recommendation in the Erlang programming guidelines has been to keep messages small. This might make programming of certain applications awkward.

4. AN ARCHITECTURE WITH A SHARED HEAP

The problems associated with costly message passing in a private heap system can be avoided by a memory architecture where the heap is shared. In such a system each process can still have its own stack, but there is only one global heap, shared by all processes. The shared heap contains both messages and all compound terms. Figure 3 depicts such an architecture.

Process communication. Message passing is done by just placing a pointer to the message in the receiver's mailbox (located in its PCB); see Figure 4. The shared heap remains unchanged, and neither copying nor traversal of the message is needed. In this architecture, message passing is a constant time operation.

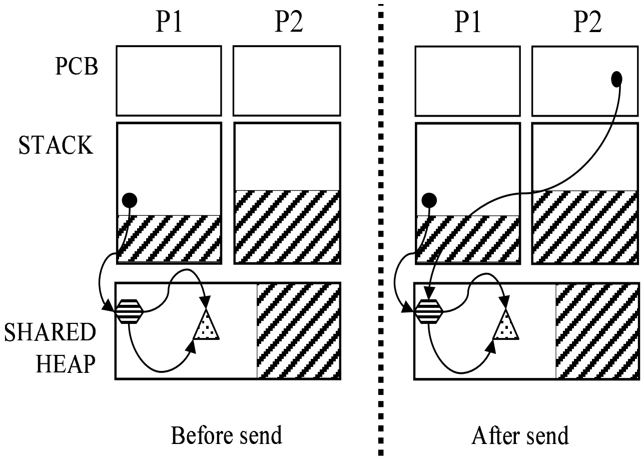

Figure 4: Message passing in a shared heap system.

Garbage collection. Conceptually, the garbage collector for this system is the same as in the private heap one, the only difference being that the root set includes the stacks and mailboxes of all processes; not just those of the process forcing the garbage collection. This implies that, even in a multi-threaded system, all processes get blocked by GC.

Pros and cons. This design avoids the disadvantages of the private heap system, which are now turned into advantages:

+ Fast message passing – As mentioned, message passing only involves updating a pointer; an operation which is independent of the message size.
+ Less space needs – Since data passed as messages is shared on the global heap, the total memory requirements are lower than in a private heap system. Also, note that since nothing is changed on the heap, shared subterms of messages remain of course shared within a message.
+ Low fragmentation – The whole memory in the shared heap is available to any process that needs it.

Unfortunately, even this system has disadvantages:

– Larger root set – Since all processes share the heap, the root set for each GC conceptually includes the stacks of all processes. Unless a concurrent garbage collector is used, all processes remain blocked during GC.

– Larger to-space – With a copying collector a to-space as large as the heap which is being collected needs to be allocated. One would expect that in general this area is larger when there is a shared heap than when collecting the heap of each process separately.

– Higher GC times – When a copying collector is used, all live data will be moved during garbage collection. As an extreme case, a sleeping process that is about to die with lots of reachable data will affect the garbage collection times for the whole system. With private heaps, the live data of only the process that forces the garbage collection needs to be moved during GC.

– Separate and probably more expensive tests for heap and stack overflows.

The following difference between the two memory architectures also deserves to be mentioned: In a process-centric system, it is easy to impose limits on the space resources that a particular (type of) process can use. Doing this in a shared heap system is significantly more complicated and probably quite costly. Currently, this ability is not required by Erlang.

Optimizations. The problems due to the large root set can be to a large extent remedied by some simple optimizations. For the frequent minor collections, the root set need only consist of those processes that have touched the shared heap since the last garbage collection. Since each process has its own stack, a safe approximation, which is cheap to maintain and is the one we currently use in our implementation, is to consider as root set the set of processes that have been *active* (have executed some code or received a message in their mailbox) since the last garbage collection.[3]

A natural refinement is to further reduce the size of the root set by using *generational stack collection* techniques [8] so that, for processes which have been active since the last GC, their entire stack is not rescanned multiple times. Notice however that this is an optimization which is applicable to all memory architectures. We are currently investigating the effect of generational stack scanning.

Finally, the problem of having to move the live data of sleeping processes could be remedied by employing a non-moving garbage collector for the old generation.

5. AN ARCHITECTURE WITH PRIVATE HEAPS & A SHARED MESSAGE AREA

Each of the memory architectures described so far has its advantages. Chief among them are that the private heap system allows for cheap reclamation of memory upon process termination and for garbage collection to occur independently of other processes, while the shared heap system optimizes interprocess communication and does not require unnecessary traversals of messages. Ideally, we want an architecture that combines the advantages of both systems without inheriting (m)any of its disadvantages.

Such an architecture can be obtained by a hybrid system in which there is one shared memory area where messages (i.e., data which is exchanged between processes) are placed, but each process has its private heap for the rest of its data (which is local to the process). In order to make it possible to collect the private heap of a process without touching data in the global area, and thus without having to block other processes during GC, there should not be any pointers from the shared message area to a process' heap. Pointers from private heaps (or stacks) to the shared area are allowed. Figure 5 shows this memory architecture: The three processes P1, P2, and P3 each have their own PCB, stack, and private heap. There is also a shared area for messages. The picture shows pointers of all allowed types. Notice that there are no pointers out of the shared area, and no pointers between private heaps.

[3]In our setting, this optimization turns out to be quite effective independently of application characteristics. This is because in an Erlang/OTP system there is always a number of system processes (spawned at system start-up and used for monitoring, code upgrading, or exception handling) that typically stay inactive throughout program execution.

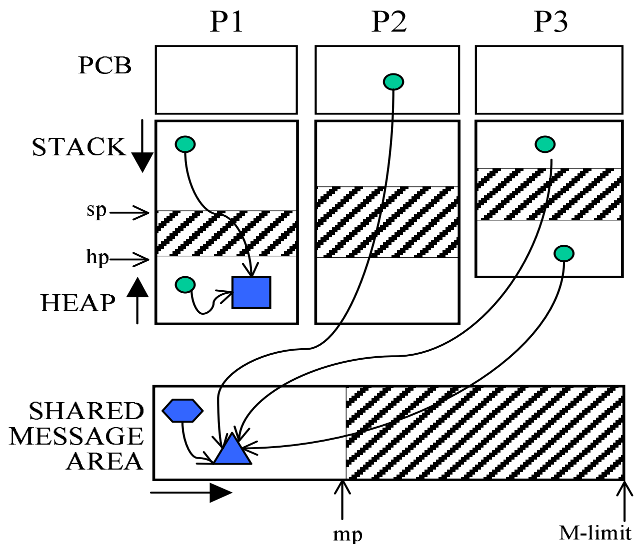

Figure 5: A hybrid memory architecture.

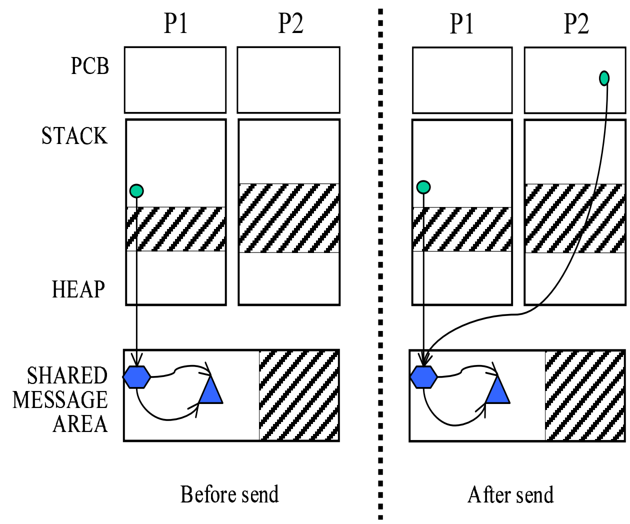

Figure 6: Message passing in a hybrid architecture.

Allocation strategy. This hybrid architecture requires information whether data is local to a process or will be sent as a message (and thus is shared). It is desirable that such information is available *at compile time* and can be obtained either by programmer annotations, or automatically through the use of an *escape analysis*. Such analyses have been previously developed for allowing stack allocation of data structures in functional languages [19] and more recently for synchronization removal from Java programs [5, 9, 20]. It is likely that separate compilation, dynamically linked libraries, or other language constructs (e.g., in Erlang the ability to dynamically update the code of a particular module) might in practice render such analyses imprecise. Hence such a hybrid system which depends on analysis has to be designed with the ability to handle imprecise escape information.

More specifically, the information returned by such an escape analysis is that at a particular program point either an allocation is of type *local* to a process, or *escapes* from the process (i.e., is part of a message), or is of *unknown* type (i.e., *might* be sent as a message). The system should then decide where data of *unknown* type is to be placed. If allocation of *unknown* data happens on the local heap, then each send operation has to test whether its message argument resides on the local heap or the message area. If the data is already global, all is fine and a pointer can be passed to the receiver. Otherwise the data has to be copied from the local heap to the message area. This design minimizes the amount of data on the shared message area. Still, some messages will need to be copied with all the disadvantages of copying data. If, on the other hand, allocation of *unknown* data happens on the shared memory area, then no test is needed and no data ever needs to be copied. The downside is that some data that is really local to a process might end up on the shared area where they can only be reclaimed by garbage collection.

Process communication. Provided that the message resides in the shared message area, message passing in this architecture happens exactly as in the shared heap system and is a constant time operation. For uniformity, Figure 6 depicts the operation. As mentioned, if a piece of data which is actually used as a message is somehow not recognized as such by the escape analysis, it first has to be copied from the private heap of the sender to the shared message area.

Garbage collection. Since there exist no external pointers into a process' private area, neither from another process nor from the shared message area, local minor and major collections (i.e., those caused by overflow of a private heap) can happen independently from other processes (no synchronization is needed) and need not block the system. This is contrary to Steensgaard's scheme [21] where GCs always collect the shared area and thus require locking.

In our scheme, garbage collection of the shared message area requires synchronization. To avoid the problems of repeated traversals of long-lived messages and of having to update pointers in the private heaps of processes, the shared message area (or just its old generation) can be collected with a *non-moving* mark-and-sweep collector. This type of collector has the added advantage that it is typically easier to be made incremental (and hence also concurrent) than a copying collector. Another alternative could be to collect messages using *reference counting*. As an aside, we note that usual drawbacks of reference counting are not a problem in our setting since there are no cycles between pointers in the message area.

Pros and cons. As mentioned, with this hybrid architecture we get most of the advantages of both other systems:

+ Fast message passing.
+ Less space needs – The memory for data passed as messages between processes is shared.
+ No cost memory reclamation – When a process dies, its stack and heap can be freed directly without the need for garbage collection.
+ Small root sets for the frequent local collections – Since each process has its own heap, the root set for a lo-

cal garbage collection is only the stack of the process which is forcing the collection.

+ Cheap stack/heap overflows.

Still, this hybrid system has some disadvantages:

– Memory fragmentation.

– Large root set for the shared message area – A garbage collection of the shared area needs to examine all processes' stacks and local heaps rendering the collection costly. In the worst case, the cost of GC will be as big as in the shared heap system. However, since in many applications messages typically occupy only a small fraction of the data structures created during a program's evaluation and since this shared area can be quite large, it is expected that these global GCs will be infrequent. Moreover, the root set can be further reduced with the optimizations described in Section 4.

– Requires escape analysis – The system's performance is to a large extent dependent on the precision of the analysis which is employed.

6. PERFORMANCE EVALUATION

The first two memory architectures, the one based on private heaps and that based on a shared heap, have been fully implemented and released since the fall of 2001 as part of Erlang/OTP R8.[4] The user chooses between them through a configure option. The development of the hybrid architecture has taken place after the release of R8. It is currently in a prototype stage: the runtime system support is rock-solid but the compiler does not yet feature an escape analysis component. Our plan is to complete this work and also include this memory architecture in a future Erlang/OTP release.

An extensive performance comparison of all architectures under various initial memory configurations has been performed, and the complete set of time and space measurements can be found in [23]. Due to space limitations, we only present a small subset of these measurements here; the interested reader should also look at [23]. In particular, in this paper we refrain from discussing issues related to the expansion/resizing policy used or the impact of the initial memory size of each architecture. We instead use the same expansion policy in all architectures and fix *a priori* what we believe are reasonable, albeit very conservative, initial sizes for all memory areas.

More specifically, in all experiments the private heap architecture is started with an initial combined stack/heap size of 233 words per process. We note that this is the default setting in Erlang/OTP and thus the setting most frequently used in the Erlang community. In the comparison between the private and the shared heap architecture (Section 6.2), the shared heap system is started with a stack of 233 words and an initial shared heap size of 10,946 words. At first glance it might seem unfair to use a bigger heap for the shared heap system, but since all processes in this system get to share a single heap, there is no real reason to start with a small heap size as in the private heap system. In contrast, there is a need to keep heaps small in a private heap system in order to avoid running out of memory and reduce fragmentation as in such an architecture a process that allocates a large heap hogs memory from other processes. In any case, note that these heap sizes are extremely small by today's standards (even for embedded systems). In all systems, the expansion policy expands the heap to the closest Fibonacci number which is bigger than the size of the live data[5] plus the additional memory need.

6.1 The benchmarks and the setting

The performance evaluation was based on the following benchmarks:

ring A concurrent benchmark which creates a ring of 100 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.

procs(number of processes, message size) A synthetic concurrent benchmark which 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). Each message has a counter that ensures it will be sent exactly 10 times to other processes.

sendsame, garbage, and **keeplive** are variations of the **procs** benchmark designed to test the behavior of the memory architectures under different program characteristics. The arguments to the programs are those of **procs** together with an extra parameter: the counter which denotes the number of times a message is to be sent (which is fixed to 10 for **procs**). The **sendsame** benchmark creates *a single* message and distributes it among other processes. **garbage** creates a new message each time and *makes the old one inaccessible*, while **keeplive** creates a new message each time but *keeps the old ones live* by storing them in a list.

In addition, we used the following "real-life" Erlang programs:

eddie A medium-sized (≈2,000 lines of code) application implementing a HTTP parser which handles http-get requests.

BEAM compiler A large program (≈30,000 lines of code excluding code for libraries) which is mostly sequential; processes are used only for I/O. The benchmark compiles the file `lib/gs/src/gstk_generic.erl` of the Erlang/OTP R8 distribution to bytecode.

NETSim (**N**etwork **E**lement **T**est **Sim**ulator) A large commercial application (≈630,000 lines of Erlang code) 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. The **NETSim** application consists of several different Erlang nodes. Only three of these nodes are used as benchmarks, namely a network **TMOS** server, a network **coordinator**, and the **alarm server**.

Some additional information about the benchmarks is contained in Table 1. Detailed statistics about message sizes can be found in [23].

Due to licensing reasons, the platform we had to use for the **NETSim** program was a SUN Ultra 10 with a 300 MHz

[4] Erlang/OTP can be downloaded from http://www.erlang.org.

[5] The size of live data is the size of the heap after GC.

Table 1: Number of processes and messages.

Benchmark	Processes	Messages
ring	100	100,000
life	100	800,396
eddie	2	2,121
BEAM compiler	6	2,481
NETSim TMOS	4,066	58,853
NETSim coordinator	591	202,730
NETSim alarm server	12,353	288,675
procs 100x100	100	6,262
procs 1000x100	1,000	512,512
procs 100x1000	100	6,262
procs 1000x1000	1,000	512,512

Sun UltraSPARC-IIi processor and 384 MB of RAM running Solaris 2.7. The machine was otherwise idle during the benchmark runs: no other users, no window system. Because of this, and so as to get a consistent picture, we decided to also use this machine for all other benchmarks too. Performance of all heap architectures on a dual-processor SUN machine are reported in [23].

In the rest of this section, all figures containing execution times present the data in the same form. Measurements are grouped by benchmark, and times have been normalized so that the execution time for the private heap system (leftmost bar in each group and identified by P) is 1. Bars to its right show the relative execution time for the shared heap (S) and, wherever applicable, the hybrid (H) system. For each system, the execution time is subdivided into time spent in the mutator, time spent in the send operation, time spent copying messages, and time taken by the garbage collector further subdivided into time for minor and major collections. For the private heap system, in Figures 8 and 7 we also explicitly show the time to traverse the message in order to calculate its size (this is part of the send operation). In Figures 10–12 this time is folded into the send time.

6.2 Comparison of private vs. shared heap architecture

Time performance. As can be seen in Figure 7(a), in the synthetic **procs** benchmark, the shared heap system is much faster when it comes to sending small-sized messages among 100 Erlang processes. This is partly due to the send operation being faster and partly because the shared heap system starts with a bigger heap and hence does not need to do as much garbage collection. When messages are small, increasing the number of processes to 1000 does not change the picture much as can be seen in Figure 7(b). On the other hand, if the size of the message is increased so that the shared heap system also requires garbage collection often, then the effect of the bigger root set which increases garbage collection times becomes visible; see Figures 7(c) and 7(d). This is expected, since the number of processes which have been active between garbage collections (i.e., the root set) is quite high.

The performance of the two architectures on real programs shows a more mixed picture; see Figure 8. The shared heap architecture outperforms the private heap architecture on many real-world programs. For **eddie**, the gain is unrelated to the initial heap sizes; cf. [23]. Instead, it is due to the shared heap system having better cache behavior by sharing messages and by avoiding garbage collections. In the truly concurrent programs, **ring** and **life**, the private heap system spends 18% and 25% of the execution time in interprocess communication. In contrast, the shared heap system only spends less than 12% of its time in message passing. The speedup for the **BEAM compiler** can be explained by the larger initial heap size for the shared heap system which reduces the total time spent in garbage collection to one third. The performance of the shared heap architecture is worse than that of the private heap system in two of the **NETSim** programs and there is a speedup only in the case where the number of processes is moderate. This is to some extent expected, since **NETSim** is a commercial product developed over many years using a private heap-based Erlang/OTP system and tuned in order to avoid garbage collection and reduce send times. For example, from the number of processes in Table 1 and the maximum total heap sizes which these programs allocate (data shown in Table 2), it is clear that in the **NETSim** programs either the majority of the processes do not trigger garbage collection in the private heap system as their heaps are small, or processes are used as a means to get no-cost heap reclamation. As a result, the possible gain from a different memory architecture cannot be big. Indeed, as observed in the case of **NETSim alarm server**, the large root set (cf. Table 1) can seriously increase the time spent in garbage collection and slow down execution of a program which has been tuned for a private heap architecture.

We suspect that the general speedup for the mutator in the shared heap system is due to better cache locality: partly due to requiring fewer garbage collections by sharing data between processes and partly due to having heap data in cache when switching between processes. Note that this is contrary to the general belief in the Erlang community—and perhaps elsewhere—that a process-centric memory architecture results in better cache behavior. To verify our hunch, we measured the number of data cache misses of some of these benchmarks using the UltraSPARC hardware performance counters. In programs that required garbage collection, the number of data cache misses of the shared heap system is indeed smaller than that of the private heap system; however only by about 3%. Although this confirms

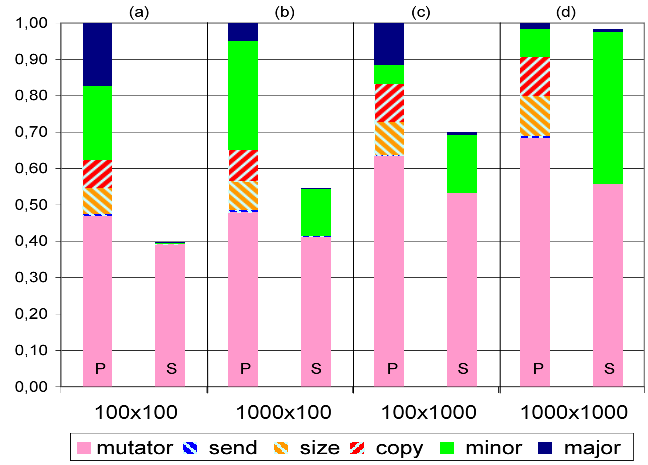

Figure 7: Normalized times for the procs benchmark.

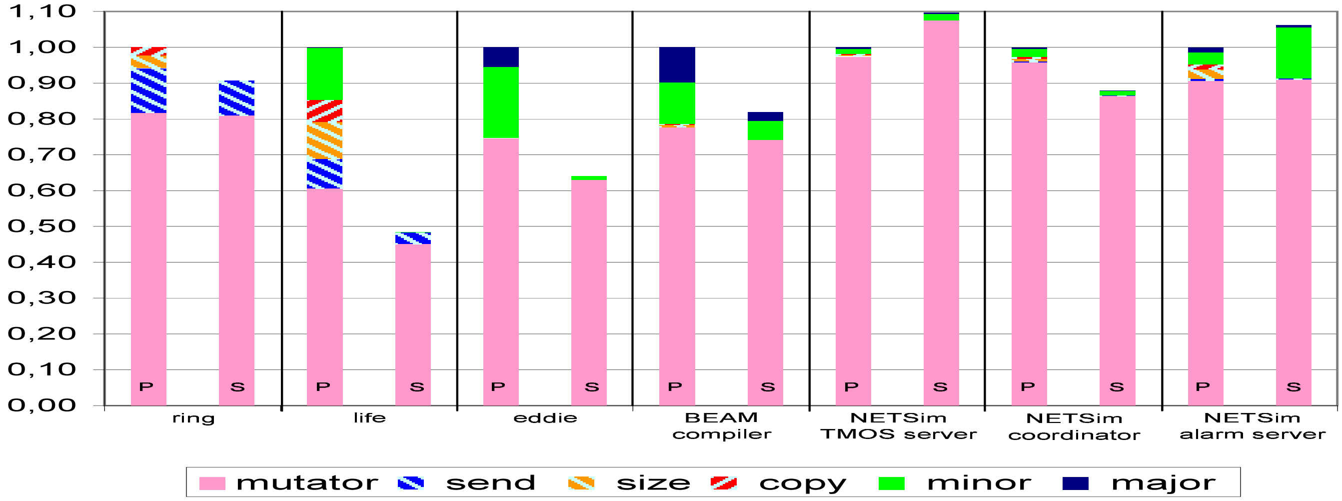

Figure 8: Normalized execution times.

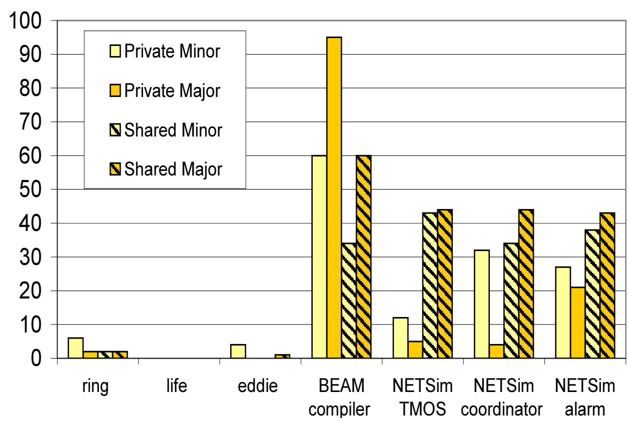

Figure 9: Max garbage collection stop times (ms).

that a shared heap system can have a better cache behavior, we are not sure whether the difference in cache misses accounts for all the mutator speedup we observe or not.

Stop times. Figure 9 shows the longest garbage collection stop time in milliseconds for each benchmark. As can be seen, the concern that many processes can lead to a larger root set and hence longer garbage collection latency is justified. When the root set consists of many processes, the stop times for the shared heap system are slightly longer than those of the private heap system.

As the memory requirements of a program increase (data shown in Table 2), the garbage collection stop times also increase. Also, the bigger the size of the live data, the more are garbage collection times likely to be negatively influenced by caching effects. Bigger heap needs also mean that collection is required more often, which increases the likelihood that GC will be triggered at a moment when the root set is large or there is a lot of live data. We mention, that although the general picture is similar, the GC latency decreases when starting the systems with bigger initial heap sizes; cf. [23].

Notice that the difference in maximum stop times between

Table 2: Heap sizes allocated and used (in K words).

	Private		Shared	
Benchmark	Allocated	Used	Allocated	Used
ring	41.6	11.7	10.9	2.3
life	52.8	33.1	28.6	28.6
eddie	78.1	67.3	46.3	46.3
BEAM compiler	1375.0	1363.0	1346.0	1346.0
NETSim TMOS	2670.5	1120.6	317.8	317.8
NETSim coordinator	233.0	162.0	121.4	121.4
NETSim alarm server	2822.9	2065.7	317.8	317.8

the two systems is not very big and that a private heap system is no guarantee for short GC stop times. True real-time GC latency can only be obtained using an on-the-fly or real-time garbage collector.

Space performance. Table 2 contains a space comparison of the private vs. the shared heap architecture on all non-synthetic benchmarks. For each program, maximum sizes of heap allocated and used is shown in thousands of words. Recall that in both systems garbage collection is triggered whenever the heap is full; after GC, the heap is not expanded if the heap space which is recovered satisfies the need. This explains why maxima of allocated and used heap sizes are often identical for the shared heap system. From these figures, it is clear that space-wise the shared heap system is a winner. By sharing messages, it usually allocates less heap space; the space performance on the **NETSim** programs is especially striking. Moreover, by avoiding fragmentation, the shared heap system has better memory utilization.

6.3 Comparison of all 3 architectures

As mentioned, the runtime system of the hybrid memory architecture is implemented, but no escape analysis is currently integrated in the compiler. For this reason, the large benchmarks cannot yet be run in this configuration. However, for the small synthetic benchmarks **keeplive**, **garbage**, and **sendsame**, we generated allocation code by hand and fed it into the system. In all benchmarks of this section, the

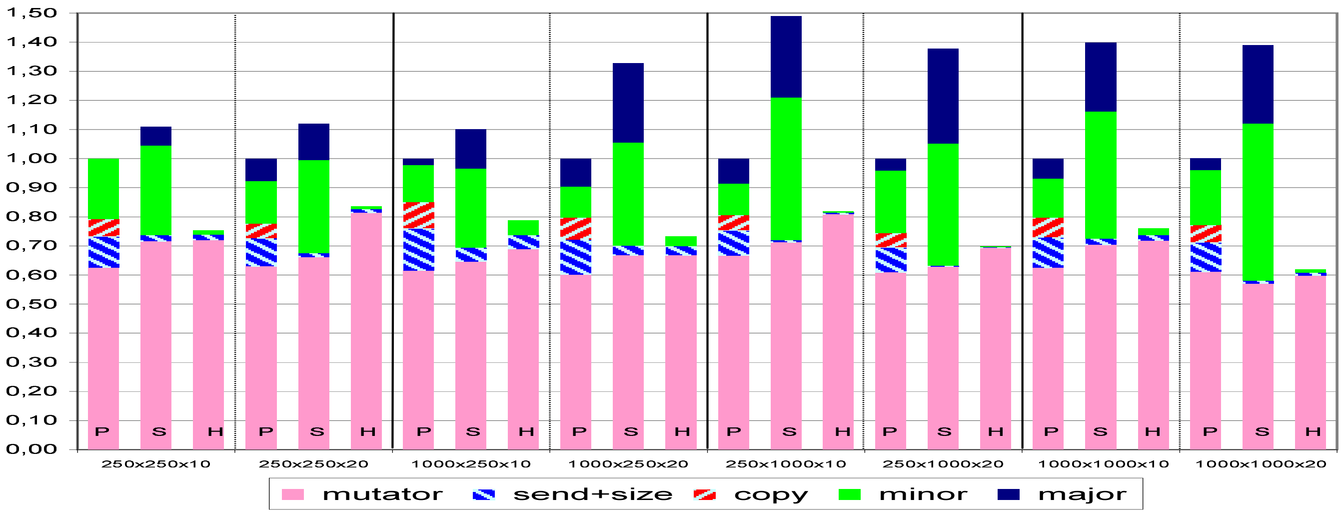

Figure 10: Performance of the keeplive benchmark.

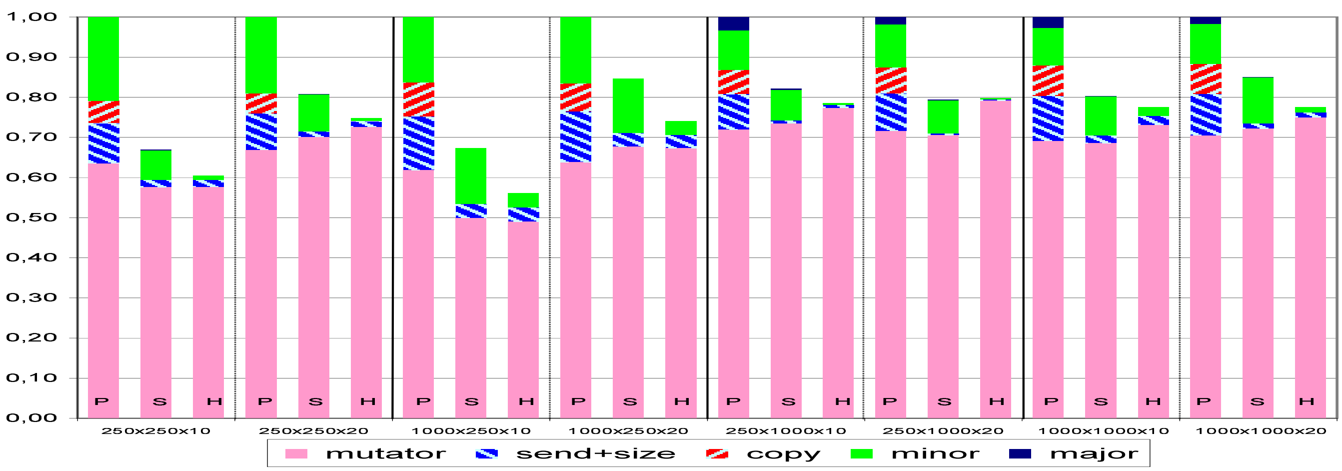

Figure 11: Performance of the garbage benchmark.

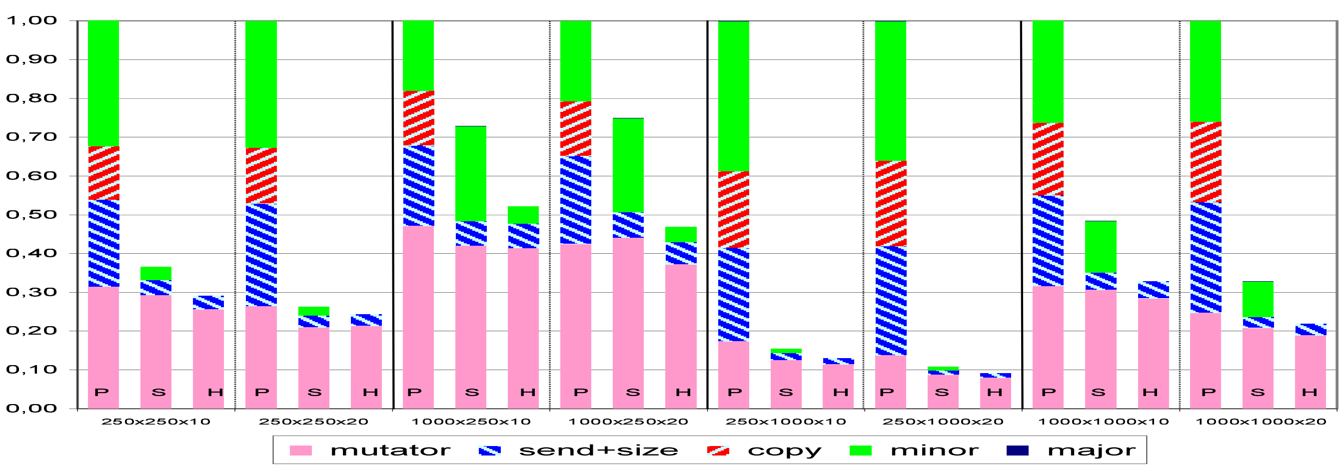

Figure 12: Performance of the sendsame benchmark.

shared memory area of the hybrid system is large enough so that none of them triggers its garbage collection. On the other hand, each of the per-process heaps for the non-shared data has an initial size of 233 words, as in the private heap system, and does require GC.

Figures 10, 11, and 12 present normalized execution times for these benchmarks. The 8 groups in each figure correspond to different arguments (NxSxT) passed to the benchmarks where N is the number of processes, S is the size of each message, and T denotes how many times each message is sent. The 3 bars in each group show normalized execution times for each system. Recall that the execution time for the shared heap (S) and the hybrid (H) system is normalized to the execution time of the private heap (P) system *for each group*. This means that one cannot compare bars from two different groups directly.

The **keeplive** benchmark is an extreme case for a copying garbage collector: each process keeps all its incoming messages live. From Figure 10, we can see that the shared heap system spends less time in send (and copy) than the private heap system. However, when the number of processes or the size of the message increases, the time that the shared heap system spends in garbage collection becomes a bottleneck, making overall execution slower than the private heap system. The hybrid system on the other hand has very low send times (no copying is required) and also very low garbage collection times due to the fact that the shared memory area is big enough to not need any garbage collection.

In the **garbage** benchmark each process throws away the incoming messages and instead creates a new message that it sends to the next process. As we can see in Figure 11, the shared heap system behaves better when the heap is not constantly overflowing with more and more live data. The hybrid system is slightly faster overall, despite the fact that its mutator is often slightly slower (perhaps due to the runtime system requiring more machinery).

Finally, the **sendsame** benchmark is an extreme case for sharing messages: a single message is created which is distributed to all the processes in the ring and then passed around to another 10 or 20 processes, depending on the benchmark's last parameter. In the private heap system the message is copied from heap to heap requiring a considerable amount of garbage collection even for modest-sized messages. In this benchmark, all that the mutator does, after creating one message once and for all, is to receive a message, decrement a counter, and pass the message on to another process. (Note once again that the bars in all groups are normalized to the the total time for the private heap system for that group; the absolute times for this benchmark are about half of those of the other benchmarks.) Both the shared heap system and the hybrid system behave extremely well on this benchmark, since message passing is much faster (no need to copy or calculate the size of the message), and since the message is not copied no new data is created and hence no garbage collection is needed. In the best case (250x1000x20) the shared heap system is over nine times faster than the private heap system. In general, the shared heap system and the hybrid system are behaving similarly on this benchmark.

7. CONCLUDING REMARKS

In this paper we have presented three alternative memory architectures for high-level programming languages that implement concurrency through message passing. We have systematically investigated aspects that might influence the choice between them, and extensively discussed the associated performance tradeoffs. Moreover, in an implementation setting where the rest of the runtime system is unchanged, we have presented a detailed experimental comparison between these architectures both on large highly concurrent programs and on synthetic benchmarks. To the best of our knowledge, all these fill a gap in the literature.

It would be ideal if the paper could now finish by announcing the "winner" heap architecture. Unfortunately, as our experimental evaluation shows, performance does depend on program characteristics and the tradeoffs that we discussed do exhibit themselves in programs. Perhaps it is better to leave this choice to the user, which is the approach we are currently taking by providing more than one heap architecture in the Erlang/OTP release. When the choice between these architectures has to be made *a priori*, it appears that the shared heap architecture is preferable to the private heap one: it results in better space utilization and is often faster, except in cases with many processes with high amounts of live data. The hybrid system seems to nicely combine the advantages of the two other architectures, and it would have been our recommendation if we had hard data on the precision of the escape analysis.

However, perhaps there are other criteria that might also influence the decision. Architectures where messages get placed in an area which is shared between processes free the programmer from worrying about message sizes. Moreover, they open up new opportunities for interprocess optimizations. For example, within a shared heap system one could, with a lower overhead than in a private heap scheme, switch to the receiving processes at a message send, achieving a form of fast remote procedure call between processes. It would even be possible to merge (and further optimize) code from two communicating processes in a straightforward manner as discussed in [15]. We intend to investigate this issue.

We are currently incorporating the escape analysis into the compiler in order to evaluate the performance of the hybrid architecture on large applications. In addition, we intend to investigate how concurrent or real-time garbage collection techniques fit into the picture.

8. ACKNOWLEDGMENTS

This research has been supported in part by the ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson Development. We thank members of the Erlang/OTP team for discussions, an anonymous referee for suggestions that improved the presentation, and Bengt Tillman and Tomas Abrahamsson from the **NETSim** team for allowing and helping us use their product as a benchmark.

9. REFERENCES

[1] J. Armstrong and R. Virding. One pass real-time generational mark-sweep garbage collection. In H. G. Baker, editor, *Proceedings of IWMM'95: International Workshop on Memory Management*, number 986 in LNCS, pages 313–322. Springer-Verlag, Sept. 1995.

[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[3] G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2000. http://www.cs.utah.edu/flux/papers/.

[4] D. F. Bacon, C. R. Attanasio, V. T. Lee, Han B. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 92–103. ACM Press, June 2001.

[5] B. Blanchet. Escape analysis for object oriented languages. Application to Java^{TM}. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34. ACM Press, Nov. 1999.

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

[7] P. Cheng and G. E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 125–136. ACM Press, June 2001.

[8] P. Cheng, R. Harper, and P. 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. ACM Press, 1998.

[9] J.-D. Choi, M. Gupta, M. Serrano, V. C. Shreedhar, and S. Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 1–19. ACM Press, Nov. 1999.

[10] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–123. ACM Press, Jan. 1993.

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

[12] M. Feeley and M. Larose. A compacting incremental collector and its performance in a production quality compiler. In *Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management*, pages 1–9. ACM Press, Oct. 1998.

[13] R. L. Hudson and J. E. B. Moss. Sapphire: Copying GC without stopping the world. In *Proceedings of the ACM Java Grande Conference*, pages 48–57. ACM Press, June 2001.

[14] L. Huelsbergen and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, pages 73–82. ACM Press, May 1993.

[15] E. Johansson and S.-O. Nyström. Profile-guided optimization across process boundaries. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 23–31. ACM Press, Jan. 2000.

[16] E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, Sept. 2000.

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

[18] S. Nettles and J. O'Toole. Real-time replication garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 217–226. ACM Press, June 1993.

[19] Y. G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–127. ACM Press, July 1992.

[20] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–218. ACM Press, June 2000.

[21] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, pages 18–24. ACM Press, Oct. 2000.

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

[23] J. Wilhelmsson. Exploring alternative memory architectures for Erlang: Implementation and performance evaluation. Uppsala master thesis in computer science 212, Uppsala University, Apr. 2002. Available at http://www.csd.uu.se/projects/hipe.

[24] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Proceedings of IWMM'92: International Workshop on Memory Management*, number 637 in LNCS, pages 1–42. Springer-Verlag, Sept. 1992. See also expanded version as Univ. of Texas Austin technical report submitted to ACM Computing Surveys.