IT Licentiate theses
2005-001

Efficient Memory Management for Message-Passing Concurrency

Part I: Single-threaded execution

JESPER WILHELMSSON

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA
UNIVERSITET

Efficient Memory Management for Message-Passing Concurrency
Part I: Single-threaded execution

BY
JESPER WILHELMSSON

May 2005

COMPUTING SCIENCE DIVISION
DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Science
at Uppsala University 2005

Efficient Memory Management for Message-Passing Concurrency

Part I: Single-threaded execution

Jesper Wilhelmsson

Jesper.Wilhelmsson@it.uu.se

Computing Science Division
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

Manual memory management is error prone. Some of the errors it causes, in particular memory leaks and dangling pointers, are hard to find. Manual memory management becomes even harder when concurrency enters the picture. It therefore gets more and more important to overcome the problems of manual memory management in concurrent software as the interest in these applications increases with the development of new, multi-threaded, hardware.

To ease the implementation of concurrent software many programming languages these days come with automatic memory management and support for concurrency. This support, called the concurrency model of the language, comes in many flavors (shared data structures, message passing, etc.). The performance and scalability of applications implemented using such programming languages depends on the concurrency model, the memory architecture, and the memory manager used by the language. It is therefore important to investigate how different memory architectures and memory management schemes affect the implementation of concurrent software and what performance tradeoffs are involved.

This thesis investigates ways of efficiently implementing the memory architecture and memory manager in a concurrent programming language. The concurrency model which we explore is based on message passing with copying semantics. We start by presenting the implementation of three different memory architectures for concurrent software and give a detailed characterization of their pros and cons regarding message passing and efficiency in terms of memory management. The issue examined is whether to use private memory for all processes in a program or if there may be benefits in sharing all or parts of the memory. In one of the architectures looked at, called *hybrid*, a static analysis called *message analysis* is used to guide the allocation of message data.

Because the hybrid architecture is the enabling technology for a scalable multi-threaded implementation, we then focus on the hybrid architecture and investigate how to manage the memory using different garbage collection techniques. We present pros and cons of these techniques and discuss their characteristics and their performance in concurrent applications. Finally our experiences from turning the garbage collector incremental are presented. The effectiveness of the incremental collector is compared to the non-incremental version. On a wide range of benchmarks, the incremental collector we present is able to sustain high mutator utilization (about 80% during collection cycles) at a low cost.

This work is implemented in an industrial-strength implementation of the concurrent functional programming language ERLANG. Our eventual goal is to use the hybrid architecture and the incremental garbage collector as the basis for an efficient multi-threaded implementation of ERLANG. The work described in this thesis is a big step in that direction.

Acknowledgments

First of all I would like to thank my supervisor, Konstantinos Sagonas, for offering me a PhD position, showing genuine interest in and providing constructive feedback on my work. He has many times pointed me in the right direction and given me papers from which I have gotten several good ideas. I am also indebted for all the effort he spends in showing me what science is about and how to pursue it.

Erik 'Happi' Stenman and Mikael Pettersson deserve my gratitude for their effort and time helping me sort out the bugs in the implementation of the shared and hybrid memory architectures which underlie this thesis. Many thanks go to my fellow PhD students in the HiPE team as well, Richard Carlsson, author of the message analysis used in this work, Per Gustafsson, and Tobias Lindahl, for their help in bouncing ideas and very enjoyable on- and off-topic discussions.

I would like to thank Björn Gustavsson of the Ericsson/OTP team for the help testing the implementation of this work and for fixing bugs in Erlang/OTP triggered by this work. I would also like to thank the NETSim team and David Wallin for providing me with real-world benchmarks and helping me use their tools.

This research has been supported in part by grant #621-2003-3442 from the Swedish Research Council (Vetenskapsrådet) and by the ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson and T-Mobile.

Finally I want to thank Camilla for all her love, patience and time. Camilla, this work would not have been possible without your sacrifices.

This thesis is dedicated to Oliver.

List of Appended Papers

The main part of this thesis consists of the following three papers. We refer to these papers as **Paper A**, **Paper B**, and **Paper C**.

Paper A Erik Johansson, Konstantinos Sagonas, and Jesper Wilhelmsson. Heap architectures for concurrent languages using message passing. In *Proceedings of ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, pages 88-99, New York, N.Y., USA, June 2002. ACM Press.

Paper B Richard Carlsson, Konstantinos Sagonas, and Jesper Wilhelmsson. Message analysis for concurrent languages. In *Static Analysis: Proceedings of the 10th International Symposium*, number 2694 in LNCS, pages 73-90, Berlin, Germany, June 2003. Springer.

Paper C Konstantinos Sagonas and Jesper Wilhelmsson. Message analysis-guided allocation and low-pause incremental garbage collection in a concurrent language. In *Proceedings of ISMM'2004: ACM SIGPLAN International Symposium on Memory Management*, pages 1-12, New York, N.Y., USA, October 2004. ACM Press.

1 Introduction

Once upon a time, computer systems were single-user, single-task machines. They could only serve one user and executed only one program at the same time. In the late 50's time sharing systems were introduced where multiple users could work concurrently on a single machine. The machine could still only execute one program at the same time, but with the use of a timing mechanism to swap between different programs the illusion of concurrency was obtained. In the 60's the first multi-processor computers were built by connecting several computers to a shared memory. For a long time large scale servers were the only place where the need for concurrency was recognized, but as computer systems evolved, concurrency found its way into new areas. In the 80's operating systems for personal computers introduced "multitasking", with which a single user could run several programs concurrently on cheap hardware. In the 90's, communication networks like those of cell phones and the Internet became an important part of our everyday life. Both are examples of concurrent application domains.

Today, general purpose processors are moving fast towards highly-concurrent computing. Several companies have during the last year announced multi-threaded and even multi-core CPUs. For example, Sun has released the UltraSPARC IV CPU with two threads built in and is about to release the Niagara, a CPU with eight processor cores built in and 32 threads. Intel's Pentium 4 is available with hyper-threading and Intel has announced that multi-core CPUs will be used for their desktop and notebook computers. Most major CPU manufacturers have announced some dual- or multi-core CPU to be released in the next few years.

It is no longer a matter of running different programs concurrently, now we want to run different parts of the same program concurrently. The fast development in the hardware industry has created a gap between hardware and software. The multi-threaded hardware is out there, but hardly any software takes advantage of it. This means that concurrency and support for multi-threading are important issues to consider in software development.

1.1 Concurrency models

There are several programming languages today with support for concurrency. This support is called the *concurrency model* of the language. It consists, among other things, of methods to pass information between different parts of a program, something that is needed to build concurrent software.

A few examples of concurrency models are synchronized shared data structures as used for example in Java and C#; synchronous message passing as used for example in Concurrent ML and Concurrent Haskell; rendevouz as used for example in Ada and Concurrent C; and asynchronous message passing as used for example in ERLANG.

In concurrency models where the communication occurs through shared data structures, synchronization trough locks is required to protect the shared objects from concurrent modifications. Unless it is decided that certain objects are thread-local with the use of, for example, an escape analysis [8], potentially all objects need locking. These lock-intensive concurrency models do not efficiently scale to large number of processors. Compared to shared data structures, a concurrency model based on message passing has the advantage that it requires fewer locks and can therefore scale better to large number of processors.

Synchronization through shared data structures is the most common technique today. However, a number of different techniques based on message passing has evolved after the advance in communication networks during the last decade. Together with the widespread use of networks, network programming became common and message passing is a concurrency model suitable for that kind of software. Efficient concurrency through message passing requires a system, or more precisely a memory architecture, that is designed with fast message passing in mind.

Paper A investigates different memory architectures and their impact on the performance of concurrent applications and presents methods for faster concurrency in programming languages using message passing.

1.2 The context of this work

This work has been performed in the context of Erlang/OTP, an industrial-strength implementation of the functional programming language ERLANG [2].

ERLANG is a strict, dynamically typed programming language with support for concurrency, communication, and distribution. ERLANG was designed to ease the implementation of concurrent control systems commonly developed by the data- and tele-communications industry. Its implementation, Erlang/OTP, is equipped with standard components for telecommunication applications (an ASN.1 compiler, the Mnesia distributed database, servers, state machines, process monitors, tools for load balancing, etc.), standard interfaces such as CORBA and XML, and a variety of communication protocols (e.g., HTTP, FTP, SMTP, etc.). In ERLANG, there is no destructive assignment of variables or data, and the first occurrence of a variable is its binding instance.

Processes in ERLANG are extremely light-weight (significantly lighter than OS threads) and the amount of processes in typical programs is large (in some cases more than 100,000 processes on a single node). ERLANG's concurrency primitives — `spawn`, “`!`” (send), and `receive` — allow a process to spawn new processes and communicate with other processes through asynchronous message passing with *copying semantics*. Any data value can be sent as a message and the recipient may be located on any machine on the network.

Erlang/OTP has so far been used successfully both by Ericsson and by other companies around the world (e.g., T-Mobile, Nortel Networks) to develop large (several hundred thousand lines of code) commercial applications.

Prior to the work described in this thesis, the only memory architecture available to Erlang/OTP was based on private memory areas for each process. The memory area where a process allocates its data is called the *heap*, and therefore we refer to this architecture as the *private heap architecture*.

In the private heap architecture, sending a message to another process involves copying the message data to the receiver's heap. This is an $O(n)$ operation, where n is the size of the message. With such a memory architecture, sending large messages is expensive. In fact it is considered so expensive that even the programming guidelines at `www.erlang.org` advise against using message passing with large messages. The cost of sending messages has made it bad programming style to use the language constructs the way they were intended when ERLANG was designed.

Since communication between processes is an important operation in concurrent software, we set out to improve the efficiency of message passing. Programmers should not be forced to consider whether to send data in a message or not; message passing is the method for interprocess communication in ERLANG, and it should be cheap enough for programmers to use it without having to worry about performance.

1.3 Improving message passing

In the private heap architecture, the send operation consists of three parts:

1. Calculate the size of the message to allocate space in the receiver's heap;
2. copy the message data to the receiver's heap; and finally,
3. deliver the message to the receiver's message queue.

To reduce the complexity of the send operation, we want to remove the parts of the send whose cost is proportional to the message size, namely 1 and 2. By introducing a new memory architecture, where all process-local heaps are replaced by a shared memory area, we can achieve this and reduce the cost of sending messages to $O(1)$, that is, make it a constant time operation. We call this the *shared heap architecture*.

Paper A describes the design and implementation of the shared heap architecture and compares it to the private heap architecture in detail. As can be seen in **Paper A**, the time to send messages is reduced in the shared heap architecture.

The shared heap architecture allows sharing of data objects in the ERLANG heap. This is safe even without locking since objects in ERLANG are immutable and the copying semantics of the send operation is preserved.

We also investigate a *hybrid architecture* where we combine the private heaps with a shared memory area called the *message area*. In the message area we preallocate data that will be used in messages. This is done with the guidance of a static message analysis. The message analysis is accurate and finds almost all messages (99% in our benchmarks). In the few cases where the analysis is unable to determine if data will end up in a message or not, the data will be allocated locally (on the private heap) and copied on demand if it is sent. This calls for a mechanism to copy (parts of) messages on the fly in the send operation.

The details of the hybrid architecture are described in **Paper A** and **Paper B**, and the message analysis used is described in **Paper B**. **Paper B** also includes a description of the algorithm to perform copy on demand as used in the send operation.

Currently, Erlang/OTP is single-threaded. The implementation of the hybrid architecture is the enabling technology for a high-performance concurrency model on top of a multi-threaded implementation of ERLANG. We are working towards such an implementation. This has played a role in the design decisions we have made and in the following sections we can see how multi-threading is one of the motivations behind the design of the hybrid architecture.

2 Automatic Memory Management

In all kinds of computer applications, manual memory management is often the source of hard-to-find software errors. The problem is to keep track of all pieces of memory that the program uses and know when to return these pieces of memory to the system. In many contexts (e.g., in concurrent software) it is hard or even impossible to know if a piece of memory can be deallocated or not without looking at the entire application. Therefore, most high-level programming languages do not allow the programmer to manually allocate and deallocate memory, but rely on *automatic memory management* like implicit allocation and the use of a *garbage collector* for deallocation.

Garbage collectors have been developed since the late 50's and come in many flavors, but the basic principles are the same in all of them. The only data a computer program can access directly, provided that programs do not access random addresses in memory, is the values stored in the computer registers. These registers will normally contain references to other data stored in the main memory of the machine. By following these references recursively until we reach a fix-point, we can find all data objects in the memory that the program can access. We call these objects *live*. When the last reference to an object is removed from the set of live objects, the object becomes *dead*. Note that the actual liveness of an object is determined by the future use of that object and the definition of live objects used in this thesis is a conservative approximation of the actual liveness. The live data may contain a reference to an object that will never be used again and is technically reclaimable, but as long as the reference is kept live, a garbage collector will not treat the object as garbage.

We distinguish between the garbage collector and the rest of the program. The program that operates in the memory managed by the garbage collector is called the *mutator* since it is the one to mutate the memory contents.

2.1 Different garbage collection techniques

The task of a garbage collector is to make sure that the memory where dead objects reside is given back to (reclaimed by) the memory allocator, for example of the operating system or the virtual machine. This can be done directly or indirectly.

A direct method monitors all updates to memory and maintains a record per data object to keep track of all references to that particular object. When there are no references left, the object is dead and its memory can be immediately reclaimed. Keeping the records up to date imposes an overhead on all memory allocations and updates. The most common direct technique is *reference counting* [19], where the record consists of a counter that keeps track of how many references there are to that particular object.

An indirect (*tracing*) garbage collector will not reclaim dead objects immediately. Instead the mutator is interrupted occasionally, for example when the allocated memory is exhausted, and the garbage collector will reclaim all objects that are dead at that point. The garbage collector will start out with a set of roots (the *root-set*). These are the memory references directly reachable from the program, for example the registers. The root-set is traversed and all objects found are marked as live and checked for references to other objects. Newly found objects will in turn be scanned and the process will continue until all objects reachable from the root-set have been marked. The remaining, unmarked objects are dead and will be reclaimed. Once the garbage collector is done, the mutator can continue. The time when the mutator is halted and the garbage collector is working is called *garbage collection stop-time*.

The marking of live data can be done in several ways. One common technique, called copying garbage collection [14], is to "rescue" the live objects by copying them to a new memory area directly when they are found. Afterwards, the entire old heap area is reclaimed. Another common technique, first described in [34], is to use a mark bit associated with each object to tag it as live. The mark bits are then used either to compact the memory, that is to copy the live data to a contiguous "block" of memory, or to create a list of the free areas in between the live objects called the *free-list*. The last technique is known as *mark-sweep collection*. Since mark-sweep never changes the location of any live data, it is referred to as a *non-moving* garbage collection algorithm.

A heap managed by a copying garbage collector is compacted and afterwards the free memory resides in one continuous area. Allocating in this area is fast and usually only involves increasing a pointer and comparing it to the end of the heap. When allocating using the free-list maintained by a non-moving garbage collector, the entire free-list may have to be traversed to find an area large enough for the allocation need. If the memory areas in the free-list are scattered in small pieces over the heap, jammed in-between used areas, the memory is *fragmented* [31]. In such cases the allocator might be unable to find a large enough free chunk of memory even though the total free memory is large enough.

During garbage collection, mark-sweep needs to traverse the memory twice. The first traversal is needed to mark the live data (only the live data is scanned in this phase), and the other to sweep the entire memory to create the free-list. A copying collector will only traverse the live data, and it will only do it once. On the other hand, it requires a free area to copy live data to during garbage collection. This area must be as large as the heap area we collect to fit all possible live data. This means that in practice only half of the available memory can be used for allocation. Copying garbage collectors also suffer from "motion sickness" — after a collection the data objects might be stored in a different order than the one they were allocated in. Objects that were allocated in a consecutive chunk might be scattered all over the heap after a copying collection leading to different cache behavior as a result.

For a thorough survey on different garbage collection techniques, see [32].

2.2 Garbage collection in ERLANG

The garbage collection techniques investigated in this thesis share two properties. They are based on tracing collectors and they are *generational*. Generational means that the garbage collector uses the *weak generational hypothesis* that "most objects die young" [45][33]. That is, most objects allocated in the heap will be dead before they get the chance to survive a garbage collection. The objects that do survive a garbage collection will most likely be around for some time in the program.

In Erlang/OTP, the heap is divided into two generations, *young* and *old*. New data is allocated in the young generation. When a garbage collection occurs, live data from the young generation is moved to the old generation. Since a garbage collection of the young generation only looks at the youngest part of the heap we call it a *minor collection*. A garbage collection of the old generation traverses all the live objects and is therefore called a *major collection*. In this thesis we have kept the generational structure in all memory architectures and garbage collection techniques we investigate.

Since ERLANG does not allow destructive assignment of variables or data, a new copy of an object needs to be created when the object is updated. Due to this allocation-intensive nature of ERLANG programs, the young generation is frequently garbage collected. Since we expect most of the data in the young generation to be dead, and we only trace live data, minor collections are expected to be fast. The old generation will normally not fill up as quickly and thus major collections are expected to be infrequent. Once a major garbage collection occurs it will in general take more time to finish than the minor collections. The reason for this is that all live data in both generations must be traversed to find all the roots for the old generation.

The private heap architecture and the shared heap architecture use the same garbage collector. This is a copying collector which means a new heap will be allocated at each garbage collection and all live data will be copied to it. In this collector, data must survive two garbage collections in the young generation before being promoted to the old generation. **Paper A** gives more details about this garbage collector.

Paper A's Table 2 shows mutator memory consumption. In the table we notice that the shared heap architecture shows a prominent decrease in memory consumption thanks to its ability to share message data. As mentioned, the shared heap architecture needs to allocate a new memory area the same size as the total shared heap during major garbage collections. The processes in the private heap architecture only garbage collect their own private heap and in a program with thousands of processes the overhead of temporarily allocating another heap area for one process is in general small.

2.2.1 Process isolation and multi-threading

In the private heap architecture each process has control over its own memory area. There are no references between private heaps which means that the garbage collection of a private heap only concerns that particular process. In a multi-threaded implementation, the fact that there are no references between the private heaps means that we can spawn of a separate thread to do the garbage collection, leaving the mutator-thread free to execute the remaining processes. This heap organization also has the property that when a process terminates, its memory can be immediately reclaimed without the need for garbage collection.

In the shared heap architecture, data from all processes is interleaved and a garbage collection requires that the root-set is gathered from all processes. This means that, at least conceptually, there is a point of global synchronization of all processes. The fact that data is interleaved also means that we are not able to reclaim the memory from private data of a process without a garbage collection when that process terminates. The larger root-set incurs more live data to copy during garbage collection which in turn results in increased stop-times. As expected, **Paper A** shows that the shared heap architecture has longer stop-times than the private heap architecture.

In the hybrid architecture, process-local data is kept in the private heaps and is garbage collected using the same algorithm as in the private and shared heap architectures. This encapsulates a process so that it can garbage collect its private heap without any synchronization with other processes. It also makes it possible to reclaim the private heap immediately when a process terminates, without garbage collection. By keeping the local data in the private heaps, less data is allocated in the message area and the number of garbage collections needed in the message area decreases. Unfortunately the major disadvantage of the shared heap can not be fully avoided. The size of the root-set for the message area is in theory as large as the root-set in the shared heap architecture. Even though less data needs to be copied since we do not copy private data during a message area collection, the number of memory words to traverse is at least the same. If the message analysis fails to recognize some message data, the data will be copied to the message area by the send operation and we end up with two live copies of the message to traverse, one in the sender's local heap referenced by the sender, and one in the message area referenced by the receiver. In practice this results in longer garbage collection stop-times.

As in the private heap architecture, the garbage collection of a private heap in the hybrid architecture can be performed in a separate thread in a multi-threaded implementation.

2.3 Garbage collection in real-time software

In the application domain of control systems, servers and communicating devices, it is common to refer to systems as *real-time*. Real-time software has to be able to respond to external signals or perform scheduled tasks within a certain deadline. If the program does not respond to a signal reasonably fast, the signal might get lost, or the response may come too late to act on it. Such systems can not be occupied doing internal maintenance work like garbage collection for a long period of time. If a system is busy doing garbage collection when a signal appears, there is no guarantee that the garbage collection will finish in time to handle the signal. That might cause the system to miss a deadline. The garbage collection stop-time can therefore be a problem in real-time software and needs some extra attention in this environment.

ERLANG is often used in high-availability large-scale embedded systems (e.g., telephone centers) and these systems typically require a high level of responsiveness. Even though the garbage collection technique used in the private heap architecture will not give any real-time guarantees, in practice the stop-times are short which is enough for so called *soft real-time* applications. With this in mind, the long stop-times of the garbage collector for the message area made us reconsider the choice of garbage collector algorithm used to collect that area.

In the hybrid architecture, the process-local heaps share the same properties as the heaps in the private heap architecture. Garbage collection of the process-local heaps is a private business and is performed without any interference with other processes. Also, as mentioned before, in a multi-threaded environment the garbage collection of a process-local heap can be performed in a separate thread. For these reasons the major bottleneck of the hybrid architecture is the garbage collection of the message area.

Since reducing the stop-times for the private heap areas is an orthogonal issue, we chose to concentrate on the message area and decided to make two major changes to the design of the memory manager. The first was to implement a non-moving (mark-sweep) garbage collector to be used in the old generation of the message area. This garbage collector has the advantage that it avoids repeated copying of old data. The design of the new memory manager, with a copying collector for the young generation and a non-moving collector for the old generation, is presented in **Paper C**. The algorithm we present for the message area is of course applicable to the private heaps as well. Our second change was to make the garbage collector for the message area incremental.

3 Incremental Garbage Collection

As mentioned before, tracing collectors interrupt the mutator to perform garbage collection. One advantage of this is that while the mutator executes, no time is spent on memory management. This usually means no overhead for the mutator on memory operations such as allocation and pointer assignment. On the negative side there is the stop-time, a sometimes long pause in mutator execution when the garbage collection takes place.

In techniques like reference counting, the garbage collection is performed continuously while the mutator executes. To manage this, reference counting needs *read* or *write barriers* that trap memory reads or writes to perform the record update along with the memory operations. This usually has a non-negligible overhead. On the other hand, it also means that the garbage collection work is evenly spread through out the execution of the application. This property, to spread the collection work throughout the application, is a desired feature and several techniques have been developed to perform *incremental garbage collection*.

Incremental garbage collectors divide the garbage collection work of a tracing collector into several smaller increments which are interleaved with mutator execution. By constraining these increments to some small enough time limit or work effort, it is possible to guarantee that the mutator will execute often enough to keep its deadlines. A measure used to determine the efficiency of the garbage collector in a real-time environment is the *Minimum Mutator Utilization* (MMU) [15]. It is defined as the minimum fraction of time that the mutator executes in any time window.

3.1 Reference counting in ERLANG

Since reference counting will perform a small part of the garbage collection at each memory update it is by nature incremental and it might seem tempting to use reference counting in the message area of the hybrid architecture. However, reference counting has drawbacks as well. It is for instance unable to reclaim cyclic data structures. Even if there is no reference from the set of live objects to the cyclic structure, the objects in the cycle will refer to each other and the records for the objects will never be empty in such structures. Since ERLANG does not allow destructive assignments, cyclic references cannot be created so this is not a concern in our case. The record itself is a bigger problem. As mentioned, a record field has to be associated to each data object in the heap. For environments using mainly small objects, this might become a problem. Take for instance lists built from cons cells; the data object (cons cell) occupies only two words of memory. With an extra word to keep the record, the object size would increase by 50%, increasing memory requirements of the program drastically. Lists are frequently used in functional languages and ERLANG is no exception.

The objects we find in an ERLANG heap are atoms, numbers (floats and arbitrary precision integers), process identifiers, function descriptors, cons cells (lists), and tuples. Each individual object is expected to be small. ERLANG also includes a binary data type (a vector of byte-sized data). These *binaries* are in general large objects and Erlang/OTP will not allocate them in the ERLANG heap but in a separate memory area. The ERLANG heap will only contain a small header-object for each binary. We have examined the heap contents of a number of large commercial ERLANG applications and found that almost three quarters of the heap objects are cons cells. Out of the last quarter, more than 99% were objects smaller than 8 words. Reference counting is therefore not an option in our case, and for the ERLANG heap we decided to stick to a tracing collector. Reference counting is however suitable (and used) for the binary-area.

Paper C describes a number of optimizations implemented to reduce the size of the root-set for the message area and, more importantly, how we use an incremental garbage collector to split the garbage collection into smaller stages to avoid the long stop-times.

3.2 A closer look at the algorithm used in the incremental collector

A *garbage collection cycle* starts when the young generation of the message area overflows, that is when the mutator wants to allocate more memory than there is free space left in the young generation. The cycle will continue through a number of *collection stages* until all live objects in the young generation have been copied to the old generation. Between each collection stage, the mutator is allowed to perform some work. This work is measured in number of words allocated by the mutator and the garbage collector will set a limit on how much work the mutator is allowed to perform before it is time to start the next garbage collection stage.

When a garbage collection cycle is initiated we optimistically assume that the collection will concern the young generation only. The calculation of mutator work allowed between collection stages is therefore based on the size of the young generation. The mutator is given a new, empty memory area to allocate in during the collection cycle (the *nursery*). The nursery has the same size as the young generation we are about to garbage collect which means that as long as we can rescue live data in the same speed as the mutator is allocating, we can guarantee that the collection cycle will end before the mutator runs out of memory in the nursery.

The copying garbage collector of the young generation will move live data into the memory area managed by the non-moving collector, which is used in the old generation. If this area overflows during the collection, a garbage collection of the old generation is required. The allowed mutator work now has to be calculated based on the size of the whole message area since we still have to guarantee that the collection cycle (that now also includes a collection of the old generation) is done before the mutator runs out of memory.

The incremental garbage collector in **Paper C** is presented in two different versions, a time-based and a work-based one. The time-based collector splits the garbage collection cycle into stages of a given (user-defined) time interval. The allowed mutator work is then calculated using a worst-case approximation, described in **Paper C**, based on the collection speed. The work-based collector rescues at least a constant (again user-defined) number of live words during each increment and the allowed mutator work is based on this number.

With small increments the work-based collector interleaves the garbage collection work with the mutator with stop-times in the order of a few micro seconds. The incremental versions of the collector do not impose any noticeable overhead on the mutator and require no costly read or write barriers. The only barrier enforced by the incremental garbage collector traps messages that have not been processed by the garbage collector yet if they are sent to a process that has already been scanned for roots. This is a cheap write barrier since it only affects the send operation and not common memory operations such as reads or writes. Its overhead is so small, it is not noticeable in runtime performance.

Paper C also reports on the mutator utilization of the hybrid architecture. In all real-world benchmarks, the mutator gets to work on average over 80% of the time in the work-based collector and about 75-80% in the time-based collector.

4 Contributions

Paper A describes the implementation of the shared heap and the hybrid heap memory architectures. It compares them to the private heap architecture and discusses pros and cons of all three schemes. This paper presents the first detailed characterization of the advantages and disadvantages of different memory architectures in a concurrent language where inter-process communication occurs through message passing.

Paper B gives some more in-depth information about the hybrid architecture and presents the details of the message analysis. The effectiveness of the analysis and the performance of the hybrid architecture with and without the analysis is reported. The novel characteristics of the analysis are that it does not rely on the presence of type information and it does not sacrifice precision when handling list types. We also describe the technique used to copy messages on the fly with support for garbage collection at any point in time.

Paper C describes the garbage collection algorithm used for the message area and compares an incremental and a non-incremental version of the same algorithm. The collector imposes no noticeable overhead on the mutator, requires no costly barrier mechanisms, and has a relatively small space overhead. The incremental collector obtains short stop-times and achieves high mutator utilization.

All the above memory architectures and garbage collection algorithms are implemented and evaluated in the industrial-strength implementation of Erlang/OTP and the work has also been included in the open source release of this system. For each of the three papers, all comparisons of memory architectures, garbage collection techniques etc. are made within the same version of Erlang/OTP. Only the properties we investigate differ between the systems we benchmark. In all three papers, we use real-world applications for benchmarking.

Related work is discussed in each of the three papers.

Comments on my participation

I started this work by implementing the shared heap and the hybrid heap architectures. I also designed and implemented the algorithm to copy parts of messages on the fly with support for garbage collection at any time. Once this was done and we realized that the garbage collector was not suitable for the shared message area, I designed and implemented the incremental garbage collector for the message area with a copying young generation and a non-moving old generation, including optimizations. To be able to obtain hard measurements on time performance I have also implemented the benchmarking capabilities of Erlang/OTP, first using the built-in Solaris timers and later using x86-linux hardware performance counters [38]. The benchmark support also includes counters and statistics for various system activities and memory usage. Finally, I have performed the benchmarking in all three papers and written the memory manager implementation-specific parts of **Paper B** and **Paper C**.

My contribution to the work presented in **Paper B** has been to enable the runtime system to allocate messages directly in the message area (as opposed to being copied there in the send operation) and perform the benchmarking. I have not been involved in the design of the message analysis.

Paper A

Heap Architectures for Concurrent Languages using Message Passing

Published in Proceedings of ISMM'2002: ACM SIGPLAN
International Symposium on Memory Management

June 2002

Heap Architectures for Concurrent Languages using Message Passing

Erik Johansson, Konstantinos Sagonas, and Jesper Wilhelmsson
Computing Science Department
Uppsala University, Sweden
{happi,kostis,jesperw}@it.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.

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 [32, 47] 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 [29]), 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 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 and 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 [44], 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-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 [21] who combine a fast, asynchronous copying collector for the thread-specific young generations with a non-disruptive concurrent mark-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 [24] 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 [27, 36, 15], and recently for Java; see for example [4, 26].

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 architecture 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]

[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.

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 proposed by Steensgaard [41] 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 [23] argued the case for an 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 [23], 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 an 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 [14] 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 [46].

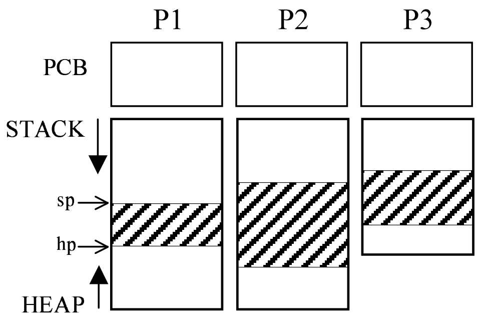


Figure 1: Memory architecture with private heaps.

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, for example 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 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

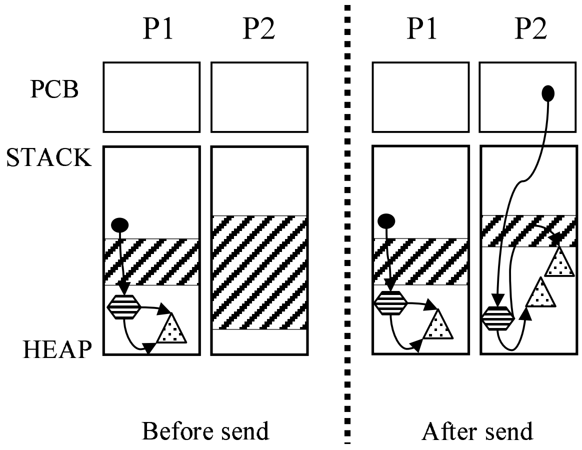


Figure 2: Message passing in a private heap system.

PCB; see Figure 2. As shown in the figure, a local data structure might share the same copy of a sub-term, but when that data structure is sent to another process each sub-term 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 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.

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

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.

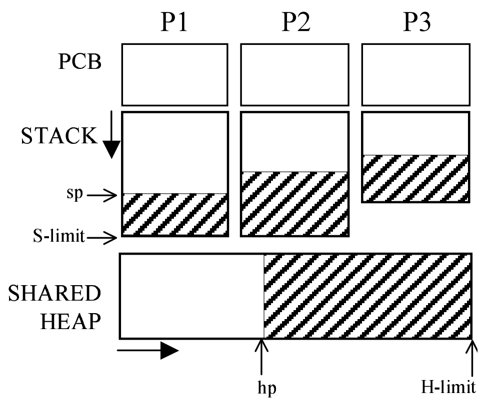


Figure 3: Memory architecture with shared heap.

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.

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.

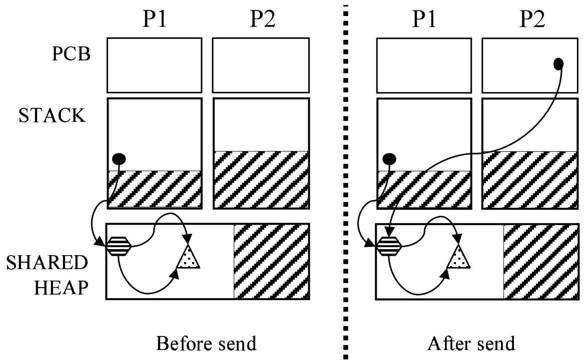


Figure 4: Message passing in a shared heap system.

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 sub-terms 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 [16] 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 and 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

[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.

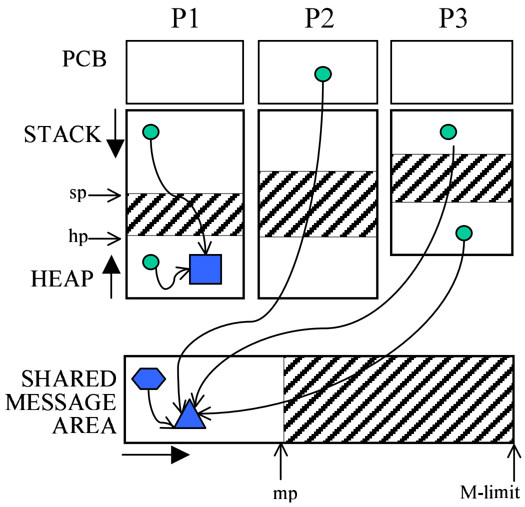


Figure 5: A hybrid memory architecture.

(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.

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 [37] and more recently for synchronization removal from Java programs [7, 17, 39]. 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

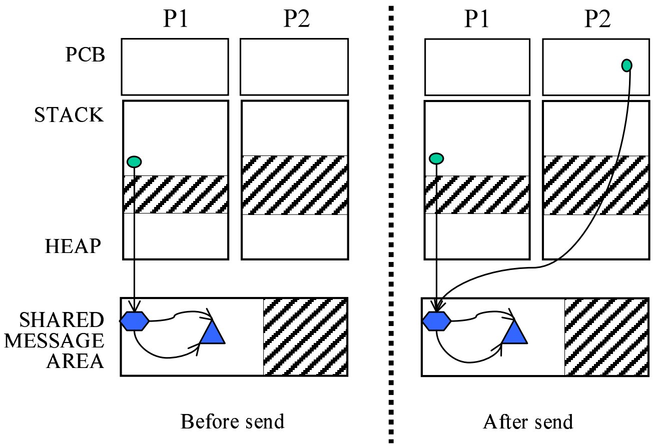


Figure 6: Message passing in a hybrid architecture.

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 [41] 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-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 local 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 [46]. Due to space limitations, we only present a small subset of these measurements here; the interested reader should also look at [46]. 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:

[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.

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 byte code.

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 [46].

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

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

Table 1: Number of processes and messages.

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 [46].

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 versus 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,

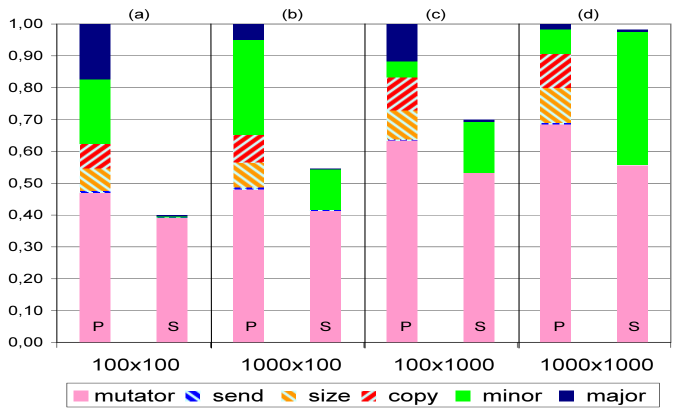


Figure 7: Normalized times for the **procs** benchmark.

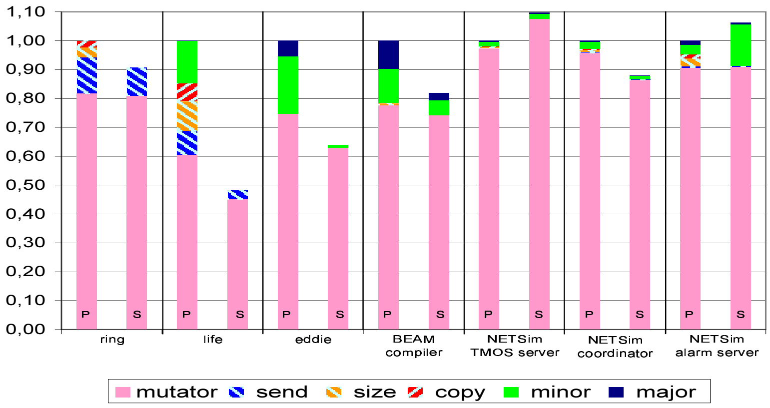


Figure 8: Normalized execution times.

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. [46]. 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

	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

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

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 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.

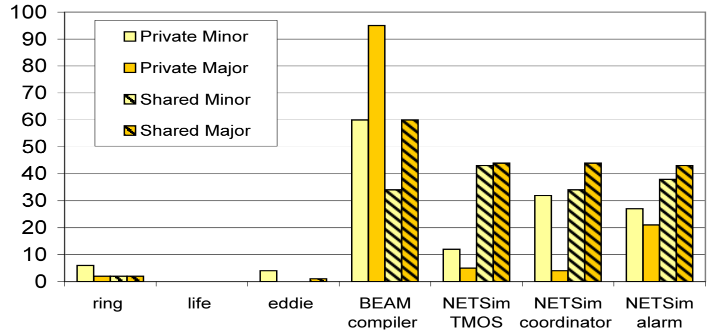


Figure 9: Max garbage collection stop-times (ms).

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. [46].

Notice that the difference in maximum stop-times between 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 versus 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.

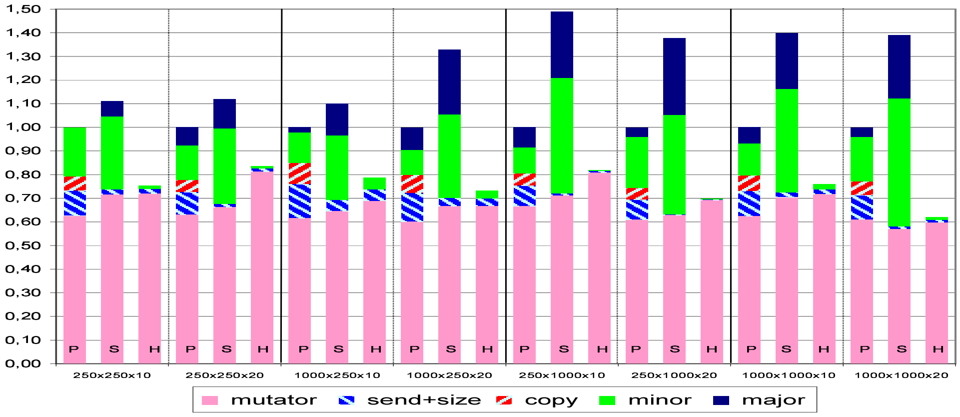


Figure 10: Performance of the **keeplive** benchmark.

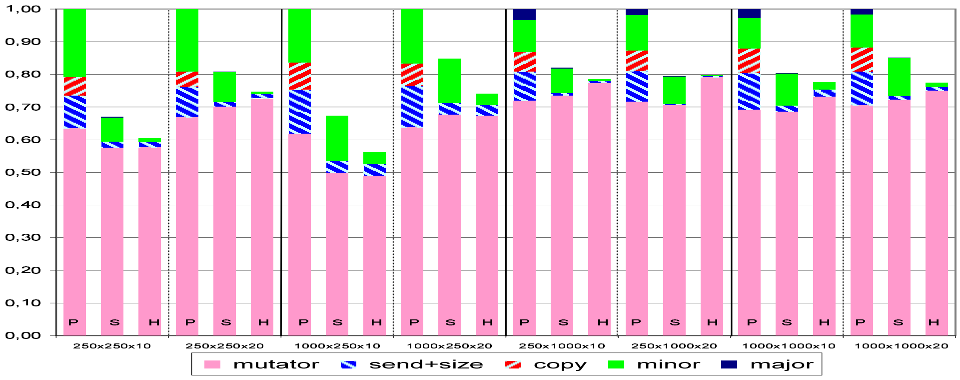


Figure 11: Performance of the **garbage** benchmark.

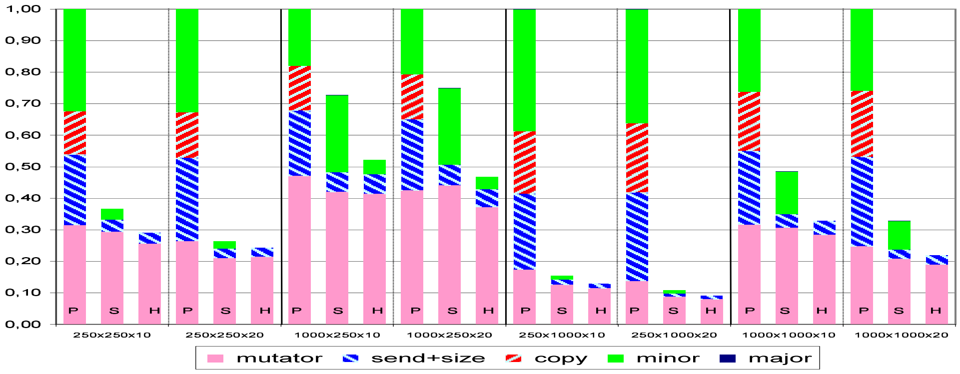


Figure 12: Performance of the **sendsame** benchmark.

6.3 Comparison of all three 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 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 [28]. 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.

Paper B

Message Analysis for Concurrent Languages

Published in Static Analysis: Proceedings of the 10:th International Symposium (SAS'03)

June 2003

Message Analysis for Concurrent Languages

Richard Carlsson, Konstantinos Sagonas, and Jesper Wilhelmsson
Computing Science Department
Uppsala University, Sweden
{richardc,kostis,jesperw}@it.uu.se

Abstract

We describe an analysis-driven storage allocation scheme for concurrent languages that use message passing with copying semantics. The basic principle is that in such a language, data which is not part of any message does not need to be allocated in a shared data area. This allows for deallocation of thread-specific data without requiring global synchronization and often without even triggering garbage collection. On the other hand, data that is part of a message should preferably be allocated on a shared area, which allows for fast ($O(1)$) interprocess communication that does not require actual copying. In the context of a dynamically typed, higher-order, concurrent functional language, we present a static message analysis which guides the allocation. As shown by our performance evaluation, conducted using an industrial-strength language implementation, the analysis is effective enough to discover most data which is to be used as a message, and to allow the allocation scheme to combine the best performance characteristics of both a process-centric and a shared-heap memory architecture.

1 Introduction

Many programming languages nowadays 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 through synchronized shared structures (as for example in Java), using synchronous message passing on typed channels (as for example in Concurrent ML), or using asynchronous message passing (as for example in ERLANG). Most of these languages typically also require support for automatic memory management, usually implemented using a garbage collector. So far, research has largely focused on the memory reclamation aspects of these concurrent systems. As a result, by now, many different garbage collection techniques have been proposed and their characteristics are well-known; see for example [32].

A less treated, albeit key issue in the design of a concurrent language implementation is that of memory allocation. It is clear that, regardless of the concurrency model of the language, there exist several different ways of structuring the memory architecture, each having its pros and cons. Perhaps surprisingly, till recently, there has not been any in-depth investigation of the performance tradeoffs that are involved in the choice between these alternative architectures. In [30], we provided the first detailed characterization of the advantages and disadvantages of different memory architectures in a language where communication occurs through message passing.

The reasons for focusing on this type of languages are both principled and pragmatic. Pragmatic because we are involved in the development of a production-quality system of this type, the Erlang/OTP system, which is heavily used as a platform for the development of highly concurrent (thousands of processes) commercial applications. Principled because, despite current common practice, we hold that concurrency through (asynchronous) message passing with copying semantics is fundamentally superior to concurrency through shared data structures. Considerably less locking is required, and consequently the method has better performance and scales better. Furthermore, the copying semantics makes distribution transparent.

Our contributions Our first contribution, which motivates our analysis, is in the area of runtime system organization. Based on the pros and cons of different memory architectures described in [30], we describe two different variants of a runtime system architecture that has process-specific areas for allocation of local data, and a common area for data that is shared between communicating processes (i.e., is part of some message). In doing so, it allows interprocess communication to occur without actual copying, uses less overall space due to avoiding data replication, and allows for the frequent process-local heap collections to take place without a need for global synchronization of processes, reducing the level of system irresponsiveness due to garbage collection.

Our second and main contribution is to present in detail a static analysis, called *message analysis*, whose aim is to discover which data is to be used as message, and which can guide the allocation in such a runtime system architecture. Novel characteristics of the analysis are that it does not rely on the presence of type information and does not sacrifice precision when handling list types.

Finally, we have implemented these schemes in the context of an industrial-strength implementation used for highly concurrent time-critical applications, and report on the effectiveness of the analysis, the overhead it incurs on compilation times, and the performance of the resulting system.

Summary of contents We begin by introducing ERLANG and reviewing our prior work on heap architectures for concurrent languages. Section 3 goes into more detail about implementation choices in the hybrid architecture. Section 4 describes the escape analysis and message analysis, and Sect. 5 explains how the information is used to rewrite the program. Section 6 contains experimental results measuring both the effectiveness of the analysis and the effect that the use of the analysis has on improving execution performance. Finally, Sect. 7 discusses related work and Sect. 8 concludes.

2 Preliminaries and Prior Work

2.1 ERLANG and Core Erlang

ERLANG [2] is a strict, dynamically typed functional programming language with support for concurrency, distribution, communication, fault-tolerance, on-the-fly code replacement, and automatic memory management. ERLANG was designed to ease the programming of large soft real-time control systems like those commonly developed in the telecommunications industry. It has so far been used quite successfully both by Ericsson and other companies around the world to construct large (several hundred thousand lines of code) commercial applications.

ERLANG's basic data types are atoms (symbols), numbers (floats and arbitrary precision integers), and process identifiers; compound data types are lists and tuples. Programs consist of function definitions organized in *modules*. There is no destructive assignment of variables or data. Because 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 can be large (in some cases up to 50,000 processes on a single node), 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 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 all messages sent to the process will arrive. Message selection from the mailbox is done by pattern matching. In send operations, the receiver is specified by its process identifier, regardless of where it is located, making distribution all but invisible. To support robust systems, a process can register to receive a message if some other process 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, ERLANG systems support upgrading code while the system is running, a mechanism known as *dynamic code replacement*.

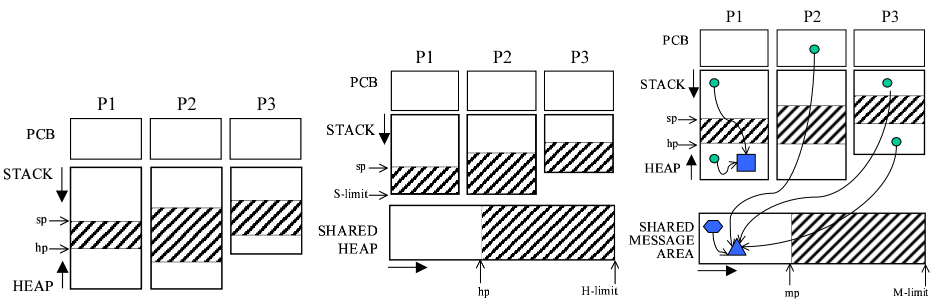
Core Erlang [12, 11] is the official core language for ERLANG, developed to facilitate compilation, analysis, verification and semantics-preserving transformations of ERLANG programs. When compiling a module, the compiler reduces the ERLANG code to Core Erlangas an intermediate form on which static analyses and optimizations may be performed before low level code is produced. While ERLANG has unusual and complicated variable scoping rules, fixed-order evaluation, and only allows top-level function definitions, Core Erlangis similar to the untyped lambda calculus with `let`- and `letrec`-bindings, and imposes no restrictions on the evaluation order of arguments.

2.2 Heap architectures for concurrent languages using message passing

In [30] we examined three different runtime system architectures for concurrent language implementations: One *process-centric* where each process allocates and manages its private memory area and all messages have to be copied between processes, one which is *communal* and all processes get to share the same heap, and finally we proposed a *hybrid* runtime system architecture where each process has a private heap for local data but where a shared heap is used for data sent as messages. Figure 1 depicts memory areas of these architectures when three processes are currently in the system; shaded areas show currently unused memory; the filled shapes and arrows in Figure 1(c) represent messages and pointers.

For each architecture, we discussed its pros and cons focusing on the architectural impact on the speed of interprocess communication and garbage collection (GC). We briefly review them below:

Process-centric. This is currently the default configuration of Erlang/OTP. Interprocess communication requires copying of messages, thus is an $O(n)$ operation where n is the message size. Also, memory fragmentation is high. Pros are that the garbage collection times and pauses are expected to be small (as the root set need only consist of the stack of the process requiring collection), and upon termination of a process, its allocated memory area can be



(a) Process-centric (b) Communal (c) Hybrid architecture

Figure 1: Different runtime system architectures for concurrent languages.

reclaimed without GC. This property in turn encourages the use of processes as a form of *programmer-controlled regions*: a computation that requires a lot of auxiliary space can be performed in a separate process that sends its result as a message to its consumer and then dies. This memory architecture has recently also been exploited in the context of Java; see [22].

Communal (shared heap). The biggest advantage is very fast ($O(1)$) interprocess communication, simply consisting of passing a pointer to the receiving process, and low memory requirements due to message sharing. Disadvantages include having to consider the stacks of all processes as root set (expected higher GC latency) and possibly poor cache performance due to processes' data being interleaved on the shared heap.

Hybrid. Tries to combine the advantages of the above two architectures: interprocess communication is fast and GC latency for the frequent collections of the per-process heaps is expected to be small. Also, this architecture allows for reclamation of data of short-lived, memory-intensive processes to happen without GC, but simply by attaching the process-local heap to a free list. However, to take advantage of this architecture, the system should be able to distinguish between data that is process-local and data which is to be shared and used as messages. This can be achieved by user annotations on the source code, by dynamically monitoring the creation of data as recently proposed in [22], or by a static analysis as we describe in Sect. 4.

Note that these runtime system architectures are applicable to all message passing concurrent languages. They are *generic*: their advantages and disadvantages in no way depend on characteristics of the ERLANG language or the current ERLANG implementation.

3 The Hybrid Architecture

A key point in the hybrid architecture is to be able to garbage collect the process-local heaps individually and without looking at the shared heap. In a multi-threaded system this allows collection of process-local heaps without any locking or synchronization. If, on the other hand, pointers from the shared area to the local heaps are allowed, these must then be traced so that what they point to is regarded as live during a local collection. This could be achieved by a read or write barrier, which typically incurs a relatively large overhead on the overall runtime. The alternative, which is our choice, is to maintain as an invariant that there are no pointers from the shared area to the local heaps, nor from one process-local heap to another; cf. Figure 1(c).

There are two possible strategies for the implementation of allocation and message passing in the hybrid architecture:

Local allocation of non-messages. Here, only data that is known to *not* be part of a message may be allocated on the process-local heap, while all other data is allocated on the shared heap. This gives $O(1)$ process communication for processes residing on the same node, since all possible messages are guaranteed to already be in the shared area, but utilization of the local heaps depends on the ability to decide through program analysis which data is definitely not shared. This approach is used by [41]. Because it is not possible in general to determine what will become part of a message, underapproximation is necessary. In the worst case, nothing is allocated in the process-local heaps, and the behavior of the hybrid architecture with this allocation strategy reduces to that of the shared heap architecture.

Shared allocation of possible messages. In this case, data that is likely to be part of a message is allocated speculatively on the shared heap, and all other data on the process-local heaps. This requires that the message operands of all send-operations are wrapped with a copy-on-demand operation, which verifies that the message resides in the shared area, and otherwise copies the locally allocated parts to the shared heap. If program analysis can determine that a message operand must already be on the shared heap, the copy operation can be statically eliminated. Without such analysis, the behavior will be similar to the process-centric architecture, except that data which is repeatedly passed as message from one process to another will only be copied once. If the analysis over-approximates too much, most of the data will be allocated on the shared heap, and we will not benefit from the process-local heaps; on the contrary, we may introduce unnecessary copying.

Copying of messages.

If the second strategy is used, as is the case in our implementation of the hybrid system, we must be prepared to copy (parts of) messages as necessary to ensure the pointer directionality invariant. Since we do not know how much of a message needs to be copied and how much already resides in the shared area, we can not ensure that the space available on the shared heap will be sufficient before we begin to copy data.

At the start of the copying, we only know the size of the topmost constructor of the message. We allocate space in the message area for this constructor. Non pointer data are simply copied to the allocated space, and all pointer fields are initialized to Nil. This is necessary because the object might be scanned as part of a garbage collection before all its children have been copied. The copying routine is then executed again for each child. When space for a child has been allocated and initialized, the child will update the corresponding pointer field of the parent, before proceeding to copy its own children.

If there is not enough memory on the shared heap for a constructor at some point, the garbage collector is called on-the-fly to make room. If a copying garbage collector is used, as is the case in our system, it will move those parts of the message that have already been copied, including the parent constructor. Furthermore, in a global collection, both source and destination will be moved. Since garbage collection might occur at any time, all local pointer variables have to be updated after a child has been copied. To keep the pointers up to date, two stacks are used during message copying: one for storing all destination pointers, and one for the source pointers. The source stack is updated when the sending process is garbage collected (in a global collection), and the destination stack is used as a root set (and is thus updated) in the collection of the shared heap.

4 Message Analysis

To use the hybrid architecture without user annotations on what is to be allocated on the local and shared heap respectively, program analysis is necessary. If data is allocated on the shared heap by default, we need to single out the data which is guaranteed to not be included in any message, so it can be allocated on the per-process heap. This amounts to escape analysis of process-local data [7, 9, 17].

$c \in$ *Const*	Constants (atoms, integers, pids and *nil*)
$x \in$ *Var*	Variables
$e \in$ *Expr*	Expressions
$l \in$ *Label*	Labels, including *xcall* and *xlambda*
$o \in$ *Primops*	Primitive operations (==, >,`is_nil`, `is_cons`, `is_tuple`, ...)

$$
\begin{array}{lcl}
v & ::= & c \mid x \\
e & ::= & v \mid (v_1\, v_2)^l \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x = b\ \texttt{in}\ e \\
b & ::= & v \mid (v_1\, v_2)^l \mid (\lambda x'.e')^l \mid \texttt{fix}\ (\lambda x'.e')^l \mid v_1 \texttt{:}^l v_2 \mid \texttt{\{}v_1, \ldots, v_n\texttt{\}}^l \mid \texttt{hd}\ v \mid \\
 & & \texttt{tl}\ v \mid \texttt{element}_k\ v \mid v_1\, \texttt{!}\, v_2 \mid \texttt{receive} \mid \texttt{spawn}\ (v_1\, v_2)^l \mid \\
 & & \texttt{primop}\ o(v_1, ..., v_n)
\end{array}
$$

Figure 2: A mini-ERLANG language

If data is by default allocated on the local heaps, we instead want to identify data that is sure to be part of a message, so it can be directly allocated in the shared area in order to avoid the copying operation when the message is eventually passed. We will refer to this special case of escape analysis as *message analysis*. Note that since copying will be performed if necessary whenever some part of a message could be residing on a process-local heap, both under- and over-approximation of the set of run-time message constructors is safe.

4.1 The analyzed language

Although our analyses have been implemented for the complete Core Erlanglanguage, for the purposes of this paper, the details of Core Erlangare unimportant. To keep the exposition simple, we instead define a sufficiently powerful language of A-normal forms [25], shown in Figure 2, with the relevant semantics of the core language (strict, higher-order, dynamically typed and without destructive updates), and with operators for asynchronous send, blocking receive, and process spawning. We also make the simplifying assumption that all primitive operations return atomic values and do not cause escapement; however, our actual implementation does not rely on that assumption.

Since the language is dynamically typed, the second argument of a list constructor $v_1 \texttt{:} v_2$ might not always be a list, but in typical ERLANG programs all lists are proper. Tuple constructors are written $\texttt{\{}v_1, \ldots, v_n\texttt{\}}$, for all $n \geq 1$. Each constructor expression in the program, as well as each call site and lambda expression, is given an unique label l. All variables in the program are assumed to be uniquely named.

Recursion is introduced with the explicit fix-point operator `fix` $(\lambda x'.e')^l$. Operators `hd` and `tl` select the first (head) and second (tail) element, respectively, of a list constructor. The operator $\texttt{element}_k$ selects the k:th element of a tuple, if the tuple has at least k elements.

The `spawn` operator starts evaluation of the application $(v_1\ v_2)$ as a separate process, then immediately continues returning a new unique process identifier ("pid"). When evaluation of a process terminates, the final result is discarded. The send operator $v_1\ !\ v_2$ sends message v_2 asynchronously to the process identified by pid v_1, yielding v_2 as result. Each process is assumed to have an unbounded queue where incoming messages are stored until extracted. The `receive` operator extracts the oldest message from the queue, or blocks if the queue is empty. This is a simple model of the concurrent semantics of ERLANG.

4.2 General framework

The analyses we have this far implemented are first-order data-flow analyses, and are best understood as extensions of Shivers' closure analysis [40]. Indeed, we assume that closure analysis has been done, so that:

- The label *xcall* represents all call sites external to the program, and the label *xlambda* represents all possible external lambdas.
- There is a mapping *calls*: *Label* $\rightarrow \mathcal{P}(\textit{Label})$ from each call site label (including *xcall*) to the corresponding set of possible lambda expression labels (which may include *xlambda*).

The domain V is defined as follows:

$$
\begin{aligned}
&V_0 = \mathcal{P}(\textit{Label}) \times \{\langle\rangle, \top\}\\
&V_i = V_{i-1} \cup \mathcal{P}(\textit{Label}) \times \textstyle\bigcup_{n\geq 0}\{\langle v_1,\ldots,v_n\rangle \mid v_1,\ldots,v_n \in V_{i-1}\} \text{ for all } i > 0\\
&V = \textstyle\bigcup_{i\geq 0} V_i
\end{aligned}
$$

Let R^* denote the reflexive and transitive closure of a relation R, and define $\sqsubseteq$ to be the smallest relation on V such that:

$$
\begin{aligned}
&(s_1, w) \sqsubseteq_i (s_2, \top) \text{ if } s_1 \subseteq s_2, \text{ for all } i \geq 0\\
&(s_1, \langle u_1,\ldots,u_n\rangle) \sqsubseteq_i (s_2, \langle v_1,\ldots,v_m\rangle)\\
&\qquad\qquad \text{if } s_1 \subseteq s_2 \wedge n \leq m \wedge \forall j \in [1,n] : u_j \sqsubseteq_{i-1} v_j, \text{ for all } i \geq 0\\
&v_1 \sqsubseteq_i v_2 \text{ if } v_1 \sqsubseteq_{i-1} v_2, \text{ for all } i > 0\\
&\\
&\sqsubseteq = \textstyle\bigcup_{i\geq 0} \sqsubseteq_i^*
\end{aligned}
$$

It is then easy to see that $\langle V, \sqsubseteq \rangle$ is a complete lattice.

Intuitively, our abstract values represent sets of constructor trees, where each node in a tree is annotated with the set of source code labels that could possibly be the origin of an actual constructor at that point. A node $(S, \top)$ represents the set of all possible subtrees where each node is annotated with set S. We identify $\bot$ with the pair $(\emptyset, \langle\rangle)$.

We define the expression analysis function $\mathcal{V}_e[\![e]\!]$ as:

$$\begin{array}{rcl}
\mathcal{V}_v[\![c]\!] & = & \bot \\
\mathcal{V}_v[\![x]\!] & = & \mathit{Val}(x) \\
\\
\mathcal{V}_e[\![v]\!] & = & \mathcal{V}_v[\![v]\!] \\
\mathcal{V}_e[\![(v_1\ v_2)^l]\!] & = & \mathit{In}(l) \\
\mathcal{V}_e[\![\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] & = & \mathcal{V}_e[\![e_1]\!] \sqcup \mathcal{V}_e[\![e_2]\!] \\
\mathcal{V}_e[\![\texttt{let}\ x = b\ \texttt{in}\ e]\!] & = & \mathcal{V}_e[\![e]\!]
\end{array}$$

and the bound-value analysis function $\mathcal{V}_b[\![b]\!]$ as:

$$\begin{array}{rcl}
\mathcal{V}_b[\![v]\!] & = & \mathcal{V}_v[\![v]\!] \\
\mathcal{V}_b[\![(v_1\ v_2)^l]\!] & = & \mathit{In}(l) \\
\mathcal{V}_b[\![(\lambda x'.e')^l]\!] & = & (\{l\}, \langle\rangle) \\
\mathcal{V}_b[\![\texttt{fix}\ (\lambda x'.e')^l]\!] & = & (\{l\}, \langle\rangle) \\
\mathcal{V}_b[\![v_1 \,\texttt{:}^l v_2]\!] & = & \mathit{cons}\ l\ \mathcal{V}_v[\![v_1]\!]\ \mathcal{V}_v[\![v_2]\!] \\
\mathcal{V}_b[\![\{v_1, \ldots, v_n\}^l]\!] & = & \mathit{tuple}\ l\ \langle \mathcal{V}_v[\![v_1]\!], \ldots, \mathcal{V}_v[\![v_n]\!] \rangle \\
\mathcal{V}_b[\![\texttt{hd}\ v]\!] & = & \mathit{head}(\mathcal{V}_v[\![v]\!]) \\
\mathcal{V}_b[\![\texttt{tl}\ v]\!] & = & \mathit{tail}(\mathcal{V}_v[\![v]\!]) \\
\mathcal{V}_b[\![\texttt{element}_k\ v]\!] & = & \mathit{elem}\ k\ \mathcal{V}_v[\![v]\!] \\
\mathcal{V}_b[\![v_1\ \texttt{!}\ v_2]\!] & = & \mathcal{V}_v[\![v_2]\!] \\
\mathcal{V}_b[\![\texttt{receive}]\!] & = & \bot \\
\mathcal{V}_b[\![\texttt{spawn}\ (v_1\ v_2)^l]\!] & = & \bot \\
\mathcal{V}_b[\![\texttt{primop}\ o(v_1, ..., v_n)]\!] & = & \bot
\end{array}$$

where

$$\begin{array}{rcl}
\mathit{cons}\ l\ x\ y & = & (\{l\}, \langle x \rangle) \sqcup y \\
\mathit{tuple}\ l\ \langle x_1, \ldots, x_n \rangle & = & (\{l\}, \langle x_1, \ldots, x_n \rangle)
\end{array}$$

and

$$
\begin{array}{rcl}
\textit{head}\ (s,w) & = & \begin{cases} (s,\top) & \text{if } w=\top \\ v_1 & \text{if } w=\langle v_1,\dots v_n\rangle, n\geq 1 \\ \bot & \text{otherwise} \end{cases} \\
\textit{tail}\ (s,w) & = & \begin{cases} (s,\top) & \text{if } w=\top \\ (s,w) & \text{if } w=\langle v_1,\dots v_n\rangle, n\geq 1 \\ \bot & \text{otherwise} \end{cases} \\
\textit{elem}\ k\ (s,w) & = & \begin{cases} (s,\top) & \text{if } w=\top \\ v_k & \text{if } w=\langle v_1,\dots v_n\rangle, k\in[1,n] \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

Because lists are typically much more common than other recursive data structures, we give them a nonstandard treatment in order to achieve decent precision by simple means. We make the assumption that in all or most programs, cons cells are used exclusively for constructing proper lists, so the loss of precision for non-proper lists is not an issue.

Suppose $z = \textit{cons}\ l\ x\ y$. If y is $(s, \langle v, \dots\rangle)$, then the set of top-level constructors of z is $s \cup \{l\}$. Furthermore, *head* z will yield $x \sqcup v$, and *tail* z yields z itself. Thus even if a list is of constant length, such as `[A, B, C]`, we will not be able to make distinctions between individual elements. The approximation is safe; in the above example, $x \sqsubseteq \textit{head}\ z$ and $y \sqsubseteq \textit{tail}\ z$.

For each label l of a lambda expression $(\lambda x.e)^l$ in the program, define $\textit{Out}(l) = \mathcal{V}_e[\![e]\!]$. Then for all call sites $(v_1\ v_2)^l$ in the program, including spawns and the dummy external call labeled *xcall*, we have $\forall l' \in \textit{calls}(l) : \textit{Out}(l') \sqsubseteq \textit{In}(l)$, and also $\forall l' \in \textit{calls}(l) : \mathcal{V}_v[\![v_2]\!] \sqsubseteq \textit{Val}(x')$, when l' is the label of $(\lambda x'.e')$. Furthermore, for each expression `let` $x = b$ `in` e' we have $\mathcal{V}_b[\![b]\!] \sqsubseteq \textit{Val}(x)$.

4.3 Termination

Finding the least solution for *Val*, *In*, and *Out* to the above constraint system for some program by fix-point iteration will however not terminate, because of infinite chains such as $(\{l\}, \langle\rangle) \sqsubset (\{l\}, \langle\ (\{l\}, \langle\rangle)\ \rangle) \sqsubset \dots$ To ensure termination, we use a variant of depth-k limiting.

We define the limiting operator θ_k as:

$$
\begin{array}{rcl}
\theta_k\ (s,\top) & = & (s,\top) \\
\theta_k\ (s,\langle\rangle) & = & (s,\langle\rangle) \\
\theta_k\ (s,\langle v_1,\dots,v_n\rangle) & = & (s,\langle \theta_{k-1}v_1,\dots,\theta_{k-1}v_n\rangle), \text{if } k>0 \\
\theta_k\ (s,w) & = & (\textit{labels}\ (s,w),\top), \text{if } k\leq 0
\end{array}
$$

where

$$\begin{array}{rcl} \mathit{labels}\ (s, \top) & = & s \\ \mathit{labels}\ (s, \langle\rangle) & = & s \\ \mathit{labels}\ (s, \langle v_1, \ldots, v_n \rangle) & = & \bigcup_{i=1}^{n} \mathit{labels}\ v_i\ \cup s \end{array}$$

The rules given in Sect. 4.2 are modified as follows: For all call sites $(v_1\ v_2)^l$, $\forall l' \in \mathit{calls}(l) \colon \theta_k \mathit{Out}(l') \sqsubseteq \mathit{In}(l)$, and $\forall l' \in \mathit{calls}(l) \colon \theta_k \mathcal{V}_v[\![v_2]\!] \sqsubseteq \mathit{Val}(x')$, when l' is the label of $(\lambda x'.e')^l$.

Note that without the special treatment of list constructors, this form of approximation would generally lose too much information; in particular, recursion over a list would confuse the spine constructors with the elements of the same list. In essence, we have a "poor man's escape analysis on lists" [37] for a dynamically typed language.

4.4 Escape analysis

As mentioned, in the scheme where data is allocated on the shared heap by default, the analysis needs to determine which heap-allocated data cannot escape the creating process, or reversely, which data can possibly escape. Following [40], we let *Escaped* represent the set of all escaping values, and add the following straightforward rules:

1. $\mathit{In}(\mathit{xcall}) \sqsubseteq \mathit{Escaped}$
2. $\mathcal{V}_v[\![v_2]\!] \sqsubseteq \mathit{Escaped}$ for all call sites $(v_1\ v_2)^l$ such that $\mathit{xlambda} \in \mathit{calls}(l)$
3. $\mathcal{V}_v[\![v_2]\!] \sqsubseteq \mathit{Escaped}$ for all send operators $v_1\,!\ v_2$
4. $\mathcal{V}_v[\![v_1]\!] \sqsubseteq \mathit{Escaped}$ and $\mathcal{V}_v[\![v_2]\!] \sqsubseteq \mathit{Escaped}$ for every `spawn` $(v_1\ v_2)$ in the program

After the fix-point iteration converges, if the label of a data constructor operation (including lambdas) in the program is not in *labels*(*Escaped*), the result produced by that operation does not escape the process.

It is easy to extend this escape analysis to simultaneously perform a more precise closure analysis than [40], which only uses sets, but doing so here would cloud the issues of this paper. Also, ERLANG programs tend to use fewer higher-order functions, in comparison with typical programs in for example Scheme or ML, so we expect that the improvements to the determined call graphs would not be significant in practice. Note that although our analysis is not in itself higher-order, we are able to handle the full higher-order language with generally sufficient precision.

4.5 Message analysis

If we instead choose to allocate data on the local heap by default, we want the analysis to tell us which data could be part of a message, or reversely, which data cannot (or is not likely to). Furthermore, we need to be able to see whether or not a value could be a data constructor passed from outside the program.

For this purpose, we let the label *unknown* denote any such external constructor, and let *Message* represent the set of all possible messages.

We have the following rules:

1. $(\{\mathit{unknown}\}, \top) \sqsubseteq \mathit{In}(l)$ for all call sites $(v_1\ v_2)^l$ such that $\mathit{xlambda} \in \mathit{calls}(l)$

2. $\mathcal{V}_v[\![v_2]\!] \sqsubseteq \mathit{Message}$ for every $v_1\ !\ v_2$ in the program

3. $\mathcal{V}_v[\![v_1]\!] \sqsubseteq \mathit{Message}$ and $\mathcal{V}_v[\![v_2]\!] \sqsubseteq \mathit{Message}$ for every `spawn` $(v_1\ v_2)$ in the program

The main difference from the escape analysis, apart from also tracking unknown inputs, is that in this case we do not care about values that leave the current process except through explicit message passing. (The closure and argument used in a `spawn` can be viewed as being "sent" to the new process.) Indeed, we want to find only those values that may be passed from the constructor point to a send operation without leaving the current process.

If the label of a data constructor is not in $\mathit{labels}(\mathit{Message})$ when a fix-point is reached, the value constructed at that point is not part of any message. Furthermore, for each argument v_i to any constructor, if $\mathit{unknown} \notin \mathit{labels}(\mathcal{V}_v[\![v_i]\!])$, the argument value cannot be the result of a constructor outside the analyzed program. Note that since the result of a `receive` is necessarily a message, we know that it already is located in the shared area, and therefore not "unknown".

5 Using the Analysis Information

Depending on the selected scheme for allocation and message passing, the gathered escape information is used as follows in the compiler for the hybrid architecture:

5.1 Local allocation of non-messages

In this case, each data constructor in the program such that a value constructed at that point is known to *not* be part of any message, is rewritten so that the allocation will be performed on the local heap. No other modifications are needed. Note that with this scheme, unless the analysis is able to report some constructors as non-escaping, the process-local heaps will not be used at all.

5.2 Shared allocation of possible messages

This requires two things:

1. Each data constructor in the program such that a value constructed at that point is likely to be a part of a message, is rewritten so that the allocation will be done on the shared heap.

2. For each argument of those message constructors, and for the message argument of each send-operation, if the passed value is not guaranteed to already be allocated on the shared heap, the argument is wrapped in a call to `copy`, in order to maintain the pointer directionality requirement.

In effect, with this scheme, we attempt to push the run-time copying operations backwards past as many allocation points as possible or suitable. It may then occur that because of over-approximation, some constructors will be made globally allocated although they will in fact not be part of any message. It follows that if an argument to such a constructor might be of unknown origin, it could be unnecessarily copied from the private heap to the shared area at runtime.

5.3 Example

In Figure 3, we show an example of an ERLANG program using two processes. The `main` function takes three equal-length lists, combines them into a single list of nested tuples, filters that list using a boolean function `test` defined in some other module `mod`, and sends the second component of each element in the resulting list to the spawned child process, which echoes the received values to the standard output.

The corresponding Core Erlangcode looks rather similar. Translation to the language of this paper is straightforward, and mainly consists of expanding pattern matching, currying functions and identifying applications of primitives such as `hd`, `tl`, `!`, $\texttt{element}_k$, `receive`, etc., and primitive operations like >, `is_nil` and `is_cons`. Because of separate compilation, functions residing in other modules, as in the calls to `mod:test(X)` and `io:fwrite(...)`, are treated as unknown program parameters.

For this example, our escape analysis determines that only the list constructors in the functions `zipwith3` and `filter` (lines 13 and 18, respectively) are guaranteed to not escape the executing process, and can be locally allocated. Since the

```
1  -module(test).
2  -export([main/3]).
3
4  main(Xs, Ys, Zs) ->
5     P = spawn(fun receiver/0),
6     mapsend(P, fun (X) -> element(2, X) end,
7             filter(fun (X) -> mod:test(X) end,
8                    zipwith3(fun (X, Y, Z) -> {X, {Y, Z}} end,
9                             Xs, Ys, Zs))),
10    P ! stop.
11
12 zipwith3(F, [X | Xs], [Y | Ys], [Z | Zs]) ->
13    [F(X, Y, Z) | zipwith3(F, Xs, Ys, Zs)];
14 zipwith3(F, [], [], []) -> [].
15
16 filter(F, [X | Xs]) ->
17    case F(X) of
18       true -> [X | filter(F, Xs)];
19       false -> filter(F, Xs)
20    end;
21 filter(F, []) -> [].
22
23 mapsend(P, F, [X | Xs]) ->
24    P ! F(X), mapsend(P, F, Xs);
26 mapsend(P, F, []) -> ok.
27
28 receiver() ->
29    receive
30       stop -> ok;
31       {X, Y} -> io:fwrite("~w: ~w.\n", [X, Y]), receiver()
33    end.
```

Figure 3: ERLANG program example.

actual elements of the list, created by the lambda passed to `zipwith3` (line 8), are being passed to an unknown function via `filter`, they must be conservatively viewed as escaping.

On the other hand, the message analysis recognizes that only the innermost tuple constructor in the lambda body in line 8, plus the closure `fun receiver/0` (line 5), can possibly be messages. If the strategy is to allocate locally by default, then placing that tuple constructor directly on the shared heap could reduce copying. However, the arguments `Y` and `Z` could both be created externally, and could thus need to be copied to maintain the pointer directionality invariant. The lambda body then becomes

```
{X, shared_2_tuple(copy(Y), copy(Z))}
```

where the outer tuple is locally allocated. (Note that the `copy` wrappers will not copy data that already resides on the shared heap; cf. Sect. 3.)

6 Performance Evaluation

The default runtime system architecture of Erlang/OTP R9 (Release 9)[1] is the process-centric one. Based on R9, we have also implemented the modifications needed for the hybrid architecture using the local-by-default allocation strategy, and included the above analyses and transformations as a final stage on the Core Erlangrepresentation in the Erlang/OTP compiler. By default, the compiler generates byte code from which, on SPARC or x86-based machines, native code can also be generated. We expect that the hybrid architecture will be included as an option in Erlang/OTP R10.

6.1 The benchmarks

The performance evaluation was based on the following benchmarks:

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

eddie A medium-sized ERLANG application implementing an HTTP parser which handles http-get requests. This benchmark consists of a number of ERLANG modules and tests the effectiveness of our analyses under separate (i.e., modular) compilation.

nag A synthetic benchmark which creates a ring of processes. Each process creates one message which will be passed on 100 steps in the ring. **nag** is designed to test the behavior of the memory architectures under different program characteristics. The arguments are the number of processes to create and the size of the data passed in each message. It comes in two flavors: **same** and **keep**. The **same** variant creates one *single* message which is wrapped in a tuple together with a counter and is then continously forwarded. The **keep** variant creates a new message at every step, but keeps received messages live by storing them in a list.

[1] Available commercially at `www.erlang.com` and as open-source at `www.erlang.org`.

Benchmark	Messages sent	Messages copied: No analysis	Messages copied: Analysis
life	8,000,404	100%	0.0%
eddie	20,050	100%	0.3%
nag - same 1000x250	103,006	100%	1.0%
nag - keep 1000x250	103,006	100%	1.0%
	Words sent	Words copied: No analysis	Words copied: Analysis
life	32,002,806	100%	0.0%
eddie	211,700	81%	34%
nag - same 1000x250	50,829,185	1.6%	$<$ 0.02%
nag - keep 1000x250	50,329,185	100%	$<$ 0.02%

Table 1: Numbers of messages sent and (partially) copied in the hybrid system.

6.2 Effectiveness of the message analysis

Table 1 shows numbers of messages and words copied between the process-local heaps and the message area in the hybrid system, both when the message analysis is not used[2] and when it is.

In the **life** benchmark, we see that while there is hardly any reuse of message data, so that the plain hybrid system cannot avoid copying data from the local heaps to the shared area, when the analysis is used the amount of copying shrinks to zero. This is expected, since the messages are simple and are typically built just before the send operations. The **eddie** benchmark, which is a real-world concurrent program, reuses about one fifth of the message data, but with the analysis enabled, the amount of copying shrinks from 81% to 34%. That this figure is not even lower is likely due to the separate compilation of its component modules, which limits the effectiveness of the analysis. In the **same** benchmark, we see that the hybrid system can be effective even without analysis when message data is heavily reused (only the top level message wrapper is copied at each send), but the analysis still offers an improvement. The **keep** version, on the other hand, creates new message data each time, and needs the analysis to avoid copying. It is clear from the table that, especially when large amounts of data are being sent, using message analysis can avoid much of the copying by identifying data that can be preallocated on the shared heap.

[2]The number of messages partially copied when no analysis is used can in principle be less than 100%, but only if messages are being forwarded exactly as is, which is rare.

Benchmark	Lines	Byte code compilation Size (bytes)	Byte code compilation Time (s)	Byte code compilation Analysis part	Native code compilation Time (s)	Native code compilation Analysis part
life	201	2,744	0.7	6%	2.3	2%
eddie	2500	86,184	10.5	9%	76.4	1%
nag	149	2,764	0.7	5%	2.2	1%
prettyprint	1081	10,892	0.9	30%	13.1	2%
pseudoknot	3310	83,092	4.2	30%	12.7	9%
inline	2700	36,412	4.0	49%	19.3	7%

Table 2: Compilation and analysis times.

6.3 Compilation overhead due to the analysis

In the byte code compiler, the analysis takes on average 19% of the compilation time, with a minimum of 3%. However, the byte code compiler is fast and relatively simplistic; for example, it does not in itself perform any global data flow analyses. Including the message analysis as a stage in the more advanced HiPE native code compiler [29], its portion of the compilation time is below 10% in all benchmarks. ERLANG modules are separately compiled, and most source code files are small (less than 1000 lines). The numbers for **eddie** show the total code size and compilation times for all its modules. We have included the non-concurrent programs **prettyprint**, **pseudoknot**, and **inline** to show the overhead of the analysis on the compilation of larger single-module applications.

6.4 Runtime performance

All benchmarks were ran on a dual processor Intel Xeon 2.4 GHz machine with 1 GB of RAM and 512 KB of cache per processor, running Linux. Times reported are the minimum of three runs and are presented excluding garbage collection times and normalized w.r.t. the process-centric memory architecture. Execution is divided into four parts: calculating message size (only in the process-centric architecture), copying of messages, bookkeeping overhead for sending messages, and mutator time (this includes normal process execution and scheduling, data allocation and initialization, and time spent in built-in functions).

In the figures, the columns marked P represent the process-centric (private heap) system, which is the current baseline implementation of Erlang/OTP. Those marked H represent the hybrid system *without* any analysis to guide it (i.e., all data is originally allocated on the process-local heaps), and the columns marked A are

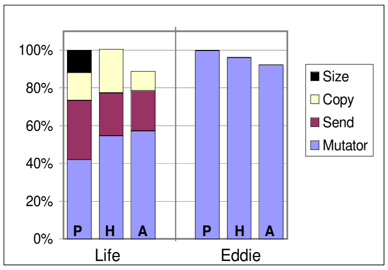


Figure 4: Performance of non-synthetic programs.

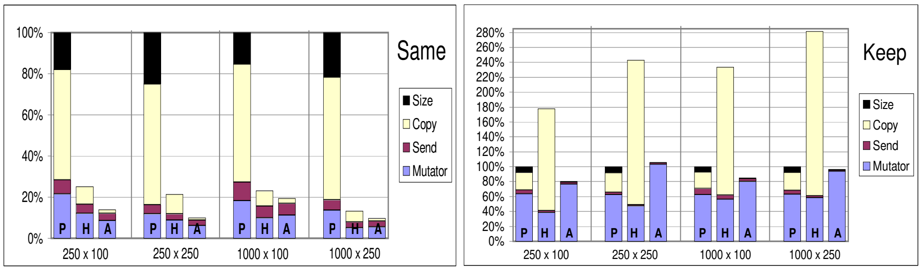


Figure 5: Performance of the **same** and **keep** variants of the **nag** benchmark.

those representing the hybrid system *with* the message analysis enabled.

In Figure 4, the **life** benchmark shows the behavior when a large number of small messages are being passed. The hybrid system with analysis is about 10% faster than the process-centric system, but we can see that although enabling the analysis removes the need for actual copying of message data (cf. Table 1), we still have a small overhead for the runtime safety check performed at each send operation (this could in principle be removed), which is comparable to the total copying time in the process-centric system when messages are very small. We can also see how the slightly more complicated bookkeeping for sending messages is noticeable in the process-centric system, and how on the other hand the mutator time can be larger in the hybrid system. (One reason is that allocation on the shared heap is more expensive.) In **eddie**, the message passing time is just a small fraction of the total runtime, and we suspect that the slightly better performance of the hybrid system is due to better locality because of message sharing (cf. Table 1).

Figure 5 shows the performance of the **nag** benchmark. Here, the hybrid system shows its advantages compared to the process-centric system when messages are larger, especially in the **same** program where most of the message data is reused. (Naturally, the speedup can be made arbitrarily large by increasing the message size, but we think that we have used reasonable sizes in our benchmarks, and that forwarding of data is not an atypical task in concurrent applications.) In the **keep** case, we see that the hybrid system with message analysis enabled is usually faster than the process-centric system also when there is no reuse. The excessive copying times in the hybrid system without the analysis show a weakness of the current copying routine, which uses the C call stack for recursion (the messages in this benchmark are lists).

7 Related Work

Our message analysis is in many respects similar to escape analysis. Escape analysis was introduced by Park and Goldberg [37], and further refined by Deutsch [20] and Blanchet [6]. So far, its main application has been to permit stack allocation of data in functional languages. In [7], Blanchet extended his analysis to handle assignments and applied it to the Java language, allocating objects on the stack and also eliminating synchronization on objects that do not escape their creating thread. Concurrently with Blanchet's work, Bogda and Hölzle [9] used a variant of escape analysis to similarly remove unnecessary synchronization in Java programs by finding objects that are reachable only by a single thread and Choi *et al.* [17] used a reachability graph based escape analysis for the same purposes. Ruf [39] focuses on synchronization removal by regarding only properties over the whole lifetimes of objects, tracking the flow of values through global state but sacrificing precision within methods and especially in the presence of recursion. It should be noted that with the exception of [17], all these escape analyses rely heavily on static type information, and in general sacrifice precision in the presence of recursive data structures. Recursive data structures are extremely common in ERLANG and type information is not available in our context.

Our hybrid memory model is inspired in part by a runtime system architecture described by Doligez and Leroy in [21] that uses thread-specific areas for young generations and a shared data area for the old generation. It also shares characteristics with the architecture of KaffeOS [3], an operating system for executing Java programs. Using escape analysis to guide a memory management system with thread-specific heaps was described by Steensgaard [41].

Notice that it is also possible to view the hybrid model as a runtime system architecture with a shared heap and separate *regions* for each process. Region-based memory management, introduced by Tofte and Talpin [43], typically allocates objects in separate areas according to their lifetimes. The compiler, guided by a static analysis called *region inference*, is responsible to generate code that deallocates these areas. The simplest form of region inference places objects in areas whose lifetimes coincide with that of their creating functions. In this respect, one can view the process-specific heaps of the hybrid model as regions whose lifetime coincides with that of the top-level function invocation of each process, and see our message analysis as a simple region inference algorithm for discovering data which outlives their creating processes.

8 Concluding Remarks

Aiming to employ a runtime system architecture which is tailored to the intended use of data in high-level concurrent languages, we have devised a powerful and practical static analysis, called *message analysis*, that can be used to guide the allocation process. Notable characteristics of our analysis are that it is tailored to its context, a dynamically typed, higher-order, concurrent language employing asynchronous message passing, and the fact that it does not sacrifice precision in the presence of recursion over lists. As shown in our performance evaluation, the analysis is in practice fast, effective enough to discover most data which is to be used as a message, and allows the resulting system to combine the best performance characteristics of both a process-centric and a shared-heap architecture and achieve (often significantly) better performance.

Paper C

Message Analysis-Guided Allocation and Low-Pause Incremental Garbage Collection in a Concurrent Language

Published in Proceedings of ISMM’2004: ACM SIGPLAN International Symposium on Memory Management

October 2004

Message Analysis-Guided Allocation and Low-Pause Incremental Garbage Collection in a Concurrent Language

Konstantinos Sagonas and Jesper Wilhelmsson
Computing Science Department
Uppsala University, Sweden
{kostis,jesperw}@it.uu.se

Abstract

We present a memory management scheme for a concurrent programming language where communication occurs using message passing with copying semantics. The runtime system is built around process-local heaps, which frees the memory manager from redundant synchronization in a multi-threaded implementation and allows the memory reclamation of process-local heaps to be a private business and to often take place without garbage collection. The allocator is guided by a static analysis which speculatively allocates data possibly used as messages in a shared memory area. To respect the (soft) real-time requirements of the language, we develop a generational, incremental garbage collection scheme tailored to the characteristics of this runtime system. The collector imposes no overhead on the mutator, requires no costly barrier mechanisms, and has a relatively small space overhead. We have implemented these schemes in the context of an industrial-strength implementation of a concurrent functional language used to develop large-scale, highly concurrent, embedded applications. Our measurements across a range of applications indicate that the incremental collector substantially reduces pause times, imposes only very small overhead on the total runtime, and achieves a high degree of mutator utilization.

1 Introduction

Concurrent, real-time programming languages with automatic memory management present new challenges to programming language implementors. One of them is how to structure the runtime system of such a language so that it is tailored to the intended use of data, so that performance does not degrade for highly concurrent (i.e., thousands of processes/threads) applications, and so that the implementation scales well in a multi-threaded or a multiprocessor setting. Another challenge is to achieve the high level of responsiveness that is required by applications from domains such as embedded control and telecommunication systems.

Taking up the latter challenge becomes tricky when automatic memory management is performed using garbage collection (GC). The naïve "stop-the-world" approach, where threads repeatedly interrupt execution of an user's program in order to perform garbage collection, is clearly inappropriate for applications with real-time requirements. It is also problematic on principle: it introduces a point of global synchronization between otherwise independent threads — and possibly also tasks — and provides no guarantees for bounds on the length of the individual pauses or for sufficient progress by the application; see [15] for a discussion of the issues that are involved.

Despite the significant progress in developing automatic memory reclamation techniques with real-time characteristics (see e.g., [5, 10, 15, 32, 36]), each technique relies on a number of (often implicit) assumptions about the architecture of the runtime system that might not be the most appropriate ones to follow in a different context. Furthermore, languages have their own characteristics which influence the trade-offs associated with each technique. For example, many collectors for object-oriented languages such as Java assume that allocating an extra header word for each object does not penalize execution times by much and does not impose a significant space overhead. Similarly, the semantics of a language may favor the use of a read rather than a write barrier, or may allow for more liberal forms of incremental collection (e.g., based on replication of objects). Finally, it is clear that the type of GC which is employed interacts with and is influenced by the allocation which is used. It is very difficult to come up with techniques that are well-suited for all runtime environments.

Our contributions Our first contribution is in the area of runtime systems architectures for highly concurrent languages where communication occurs using message passing. We present the details of a runtime system whose memory manager splits the allocated memory into areas based on the intended use of data. Its memory allocator is guided by a static analysis, which speculatively allocates data possibly used as messages in a shared memory area. Based on the characteristics of each memory area, we discuss the various types of garbage collection methods which are employed.

Our second, and main contribution is to develop a generational, incremental garbage collection scheme for this runtime system. Notable characteristics are that the collector imposes no noticeable overhead on the mutator, requires no costly barrier mechanisms, and has a relatively small space overhead.

Finally, we have implemented this scheme in the context of an industrial-strength implementation of a concurrent functional language, and we report on its performance across a range of "real-world" applications. When using the incremental collector, through various optimizations which we discuss in the paper, we are able to sustain the overall performance of the system, obtain extremely small pause times, and achieve a high degree of mutator utilization.

2 The Context of our Work

The work reported in this paper is part of an ongoing research project at Uppsala University in collaboration with the development team of the Erlang/OTP system at Ericsson. Prior work has resulted in the development of the HiPE(High Performance ERLANG) native code compiler [29], which nowadays is fully integrated in the Erlang/OTP system, and in investigation of the pros and cons of alternative runtime system architectures for concurrent languages using message passing (work reported in [30] and briefly reviewed in Sect. 2.2). Chief among the current goals of the project are to implement static analyses which determine the intended use of data in highly concurrent languages in order to guide the memory allocator, and to improve the responsiveness of the resulting system by incorporating garbage collectors with real-time characteristics and a high rate of mutator utilization.

To set our context, we briefly review the ERLANG language and the runtime system architectures of the Erlang/OTP system.

2.1 ERLANG and Erlang/OTP

ERLANG [2] is a strict, dynamically typed functional programming language with support for concurrency, communication, distribution and fault-tolerance. It has automatic memory management and supports multiple platforms. ERLANG was designed to ease the programming of soft real-time control systems commonly developed by the data- and tele-communications industry. Its implementation, the Erlang/OTP system, has so far been used quite successfully both by Ericsson and by other companies around the world (e.g., T-Mobile, Nortel Networks) to develop large (several hundred thousand lines of code) 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 the same as tuples. To allow efficient implementation of telecommunication protocols, ERLANG also includes a *binary* data type (a vector of byte-sized data) and a notation to perform pattern matching on binaries. There is no destructive assignment of variables or data and consequently cyclic references cannot be created. Because recursion is the only means to express iteration, tail call optimization is a required feature of ERLANG implementations.

Processes in ERLANG are extremely light-weight (significantly lighter than OS threads) and their number in typical applications is quite large (in some cases up to 100,000 processes on a single node). ERLANG's concurrency primitives — `spawn`, "`!`" (send), and `receive` — allow a process to spawn new processes and communicate with other processes through asynchronous message passing with *copying semantics*. Any data value can be sent as a message and the recipient may be located on any machine on the network. 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. In send operations, the receiver is specified by its process identifier, regardless of where it is located, making distribution all but invisible. To support robust systems, a process can register to receive a message if another one terminates. ERLANG also provides a mechanism that allows a process to timeout while waiting for messages and a try/catch-style exception mechanism for error handling.

ERLANG is often used in high-availability large-scale embedded systems (e.g., telephone centers), 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 their availability requirement. Consequently, an ERLANG system comes with support for upgrading code while the system is running, a mechanism known as *dynamic code replacement*. Moreover, these systems typically also require a high-level of responsiveness, and the soft real-time concerns of the language call for fast garbage collection techniques.

The ERLANG language is small, but its implementation comes with a big set of libraries. With the *Open Telecom Platform* (OTP) middleware, ERLANG is further extended with standard components for telecommunication applications (an ASN.1 compiler, the Mnesia distributed database, servers, state machines, process monitors, tools for load balancing, etc.), standard interfaces such as CORBA and XML, and a variety of communication protocols (e.g., HTTP, FTP, SMTP, etc.).

2.2 The three runtime systems of Erlang/OTP

Until quite recently, the Erlang/OTP runtime system was based on a *process-centric* architecture; that is, an architecture where each process allocates and manages its private memory area. The main reason why this memory allocation scheme was chosen was that it was believed it results in lower garbage collection latency.

Wanting to investigate the validity of this belief, in [30] we examined two alternative runtime system architectures for implementing concurrency through message passing: one which is *communal* and all processes get to share the same heap, and a *hybrid* scheme where each process has a private heap for process-local data but where a shared heap is used for data sent as messages and thus shared between processes. Nowadays, all three architectures are included in the Erlang/OTP release. We briefly review their characteristics.

Process-centric In this architecture, interprocess communication requires copying of messages and thus is an $O(n)$ operation where n is the message size. Memory fragmentation tends to be high. Pros are that the garbage collection times and pauses are expected to be small (as the root set need only consist of the stack of the process requiring collection), and upon termination of a process, its allocated memory area can be reclaimed in constant time (i.e., without garbage collection).

Communal The biggest advantage is very fast ($O(1)$) interprocess communication, simply consisting of passing a pointer to the receiving process, reduced memory requirements due to message sharing, and low external fragmentation. Disadvantages include having to consider the stacks of *all* processes as part of the root set (resulting in increased GC latency) and possibly poor cache performance due to processes' data being interleaved on the shared heap. Furthermore, the communal architecture does not scale well to a multi-threaded or multi-processor implementation, since locking would be required in order to allocate in and collect the shared memory area in a parallel setting; see [15] for an excellent recent treatment of the subject of parallel real-time GC.

Hybrid An architecture that tries to combine the advantages of the above two architectures: interprocess communication can be fast and GC latency for the frequent collections of the process-local heaps is expected to be small. No locking is required for the garbage collection of the process-local heaps, and the pressure on the shared heap is reduced so that it does not need to be garbage collected as often. Also, as in the process-centric architecture, when a process terminates, its local memory can be reclaimed by simply attaching it to a free-list.

Note that these runtime system architectures are applicable to all concurrent systems that use message passing. Their advantages and disadvantages do not depend in any way on characteristics of the ERLANG language or its current implementation.

In this paper we concentrate on the hybrid architecture. The reasons are both pragmatic and principled: Pragmatic because this architecture behaves best in practice, and principled because it combines the best performance characteristics of the other two runtime system architectures. Also, the garbage collection techniques developed in its context are applicable to the other architectures with only minor adjustments.

Assumptions Throughout the paper, for simplicity of presentation, we make the assumption that the system is running on an uniprocessor, and that message passing and garbage collection, although incremental operations, have control over their preemption (i.e., although they have to respect their work- or time-based quanta, they cannot be interrupted by the scheduler at arbitrary points when collecting).

3 Organization of the Hybrid Architecture

Figure 1 shows an abstraction of the memory organization in the hybrid architecture. In the figure, areas with lines and stripes show currently unused memory; the shapes in heaps and arrows represent objects and pointers. In the shown snapshot, three processes (P1, P2, and P3) are present. Each process has a process control block (PCB) and a contiguous private memory area with a stack and a process-local heap growing toward each other. The size of this memory area is either specified as an argument to the `spawn` primitive, set globally by the user for all processes, or defaults to a small system constant (currently 233 words). Besides the private areas, there are two shared memory areas in the system; one used for binaries above a certain size (i.e., a big object area), and a shared heap area, intended to be used for data sent between processes in the form of messages. We refer to the latter area as the *message area*.

3.1 The pointer directionality invariants

A key point in the hybrid architecture is to be able to garbage collect the process-local heaps individually, without looking at the shared areas. In a multi-threaded system, this allows collection of local heaps without any locking or synchronization. If pointers from the shared areas to the local heaps were allowed, these would then have to be traced so that what they point to would be considered live during a local collection. This could be achieved by a write barrier, but we want to avoid the overhead that this incurs. The alternative, which is our choice, is to maintain as an invariant of the runtime system that there are no pointers from the shared areas to the local heaps, nor from one process-local area to another. Figure 1 shows all types of pointers that can exist in the system. In particular:

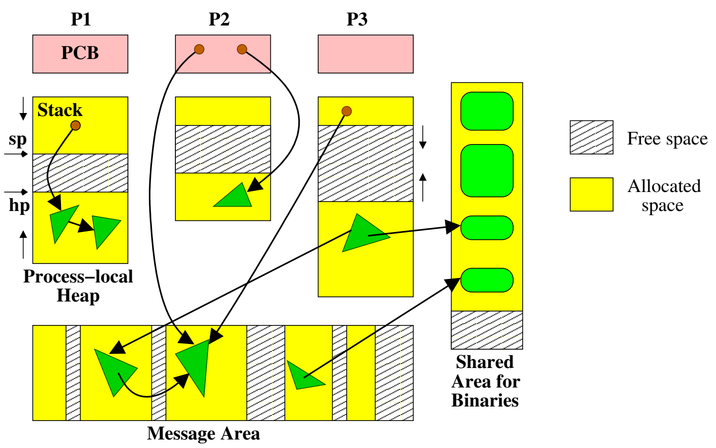


Figure 1: References allowed in the hybrid architecture.

- The area for binaries contains very few references and these are only from the header of a binary object to the start of the actual binary; these are shown in the figure. Note that these pointers will not be seen by the garbage collector.

- The message area only contains references to the shared area for binaries or to objects within the message area itself.

- Neither the shared area for binaries nor the message area contains any cyclic data.

The pointer directionality property for the message area is also crucial for our choice of memory allocation strategy, since it makes it easy to test at runtime whether or not a piece of data resides in the message area by making a simple $O(1)$ pointer comparison. (There are several possible implementations with this complexity, the simplest being mapping the message area to a single contiguous block of memory.)

3.2 Allocation in the hybrid architecture

To take full advantage of the organization of the hybrid architecture, the system needs to be able to distinguish between data which is process-local and data which is to be shared, that is, used as messages. This can be achieved by user annotations on the source code, by dynamically monitoring the creation of data as proposed in [22], or by the static *message analysis* that we have described in [13] and integrated in the hybrid runtime system configuration of Erlang/OTP.

For the purposes of this paper, the details of the message analysis are unimportant and the interested reader is referred to [13]. Instead, it suffices to understand how the analysis guides allocation of data in the compiler. The allocation can be described as *allocation by default on the local heap and shared allocation of possible messages*. More specifically, data that is *likely* to be part of a message is allocated speculatively on the shared heap, and all other data on the process-local heaps. To maintain the pointer directionality invariants, this in turn requires that the message operands of all send operations are wrapped with a copy-on-demand operation, which verifies that the message resides in the shared area (as noted above, this can be an $O(1)$ operation), and otherwise copies the locally allocated parts to the shared heap. However, if the message analysis can determine that a message operand *must* already be on the shared heap, the test can be statically eliminated.

Note that the copying semantics of message passing in ERLANG and the absence of destructive updates allows the message analysis to safely both under-approximate and over-approximate use of data as messages. With under-approximation, the data will be copied to the message area in the send operation and the behavior of the hybrid architecture will be similar to the process-centric architecture, except that data which is repeatedly passed from one process to another will only be copied once. On the other hand, if the analysis over-approximates too much, most of the data will be allocated on the shared heap, and we will not benefit from the process-local heaps; that is, data will need to be reclaimed by global garbage collection.

3.3 Allocation characteristics of ERLANG programs

In the eyes of the memory manager, the ERLANG heap only contains two kinds of objects: cons cells and boxed objects. Boxed objects are tuples, arbitrary precision integers, floats, binaries, and function closures. Boxed objects contain a header word which directly or indirectly includes information about the object's size. In contrast, there is no header word for cons cells. Regarding heap allocation, we have run a wide range of ERLANG programs and commercial applications we have access to, and have discovered that nearly three quarters (73%) of all heap-allocated objects are cons cells (of size two words). Out of the remaining ones, less than 1% is larger than eight words. Although these numbers have to be taken with a grain of salt, since each application has its own memory allocation characteristics, it is quite safe to conclude that, in contrast to, for example a Java implementation, there is a significant number of heap-allocated objects which are small in size and do not contain a header word. Adding an extra word to every object significantly penalizes execution and space consumption and is therefore not an option we consider. How this constraint influences the design of the incremental garbage collector is discussed in Section 5.

4 Garbage Collection in the Hybrid Architecture

We discuss the garbage collection schemes that are employed based on the characteristics and intended use of each memory area.

4.1 Garbage collection of process-local heaps

As mentioned, when a process dies, its allocated memory area can be reclaimed directly without the need for garbage collection. This property in turn encourages the use of processes as a form of *programmer-controlled regions*: a computation that requires a lot of auxiliary space can be performed in a separate process that sends its result as a message to its consumer and then dies. In fact, because the default runtime system architecture has for many years been the process-centric one, a lot of ERLANG applications have been written and fine-tuned with this memory management model in mind.[1]

When a process does run out of its allocated memory, the runtime system garbage collects its heap using a generational Cheney-style semi-space stop-and-copy collector [14]. (Data has to survive two garbage collections to be promoted to the old generation.) Also when running native code instead of byte code, the collector is guided by *stack descriptors* (also known as *stack maps*) and the root set is further reduced by employing *generational stack scanning* [16], an optimization which reduces the cost of scanning the root set by reusing information from previous GC scans. Although this collector cannot give any real-time guarantees, pause times when collecting process-local heaps are typically not a problem in practice. This is because most collections are minor and therefore quite fast, and as explained above many ERLANG applications have been programmed to use processes for specific, fine-grained tasks that require a relatively small amount of memory. Moreover, because process-local heaps can be collected independently, in a multi-threaded implementation, pauses due to collecting process-local heaps do not jeopardize the responsiveness of the entire system as the mutator can service other processes which are in the ready queue.

4.2 Garbage collection of binaries

The shared area for (large) binaries is collected using *reference counting* [19]. The count is stored in the header of binaries and increased whenever a new reference

[1] In this respect, process-local heaps are very much like *arenas* used by the Apache Web server [42] to deallocate all the memory allocated by a Web script once the script has terminated.

to a binary is created (when a binary is for example copied to the message area in the send operation). Each process maintains a *remembered list* of such pointers to binaries stored in the binary area. When a process dies, the reference counts of binaries in this remembered list are decreased. A similar action happens for references which are removed from the remembered list as part of garbage collection. Since cycles in binaries are not possible, cycle collection is not needed and garbage collection of binaries is effectively real-time.

4.3 Garbage collection of the message area

Since the message area is shared between processes, its garbage collection requires global synchronization. The root set is typically large since it consists of both the stacks and the process-local heaps of all processes in the system. As a result, pause times for collecting the message area can be quite high.

This situation can be ameliorated as follows:

- By splitting the message area into generations and performing generational collection on this area. In fact, one can employ a *non-moving* collector (such as *mark-sweep*) for the old generation to avoid the cost of repeatedly having to copy long-lived objects. (We still prefer to manage the young generation by a copying collector, because allocation is faster in compacted spaces.)
- By performing an optimization, called *generational process scanning*, which is the natural extension of generational root scanning from the sequential to the concurrent setting. More specifically, similarly to how generational stack scanning tries to reduce the root set which has to be considered during a process-local GC to only the “new” part of the stack, generational process scanning tries to reduce the number of processes whose memory areas are considered part of the root set. In implementation terms, the runtime system maintains information about which processes have been active (or received a message) since the last garbage collection and considers only those processes as part of the root set during the frequent minor collections.

All these techniques are used in the hybrid architecture and are quite effective. However, they can of course not provide any real-time guarantees — not even soft real-time ones — and cannot prevent GC of the message area becoming a bottleneck in highly concurrent applications. For the message area, we need a GC method that is guaranteed to result in low pause times.

Note that reference counting is *not* the most appropriate such method. The main reason is that one cannot wait until a process dies to decrease reference counts of messages that a process has sent to or received from other processes;

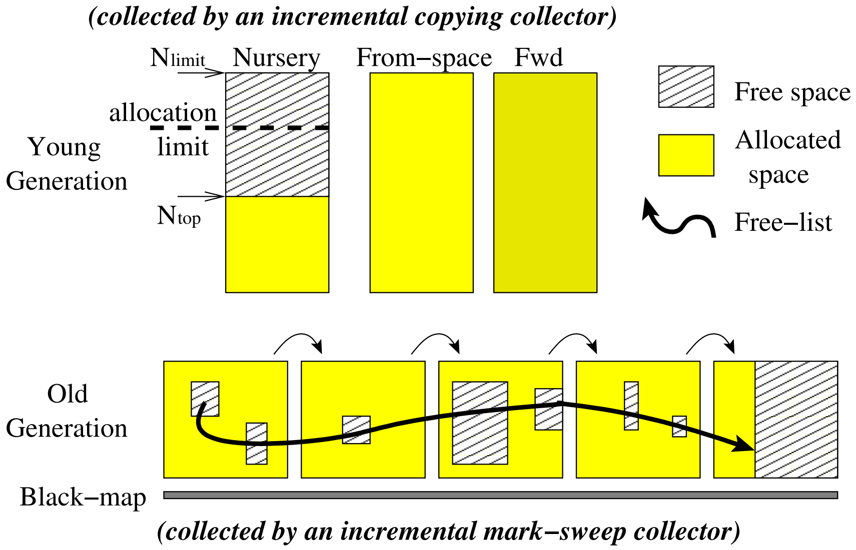


Figure 2: Organization of the message area.

consider for example the case of a Web server servicing requests. Furthermore reference counting typically imposes a non-negligible overhead. A different real-time or incremental GC is called for. We describe the one we designed and chose to implement in the next section.

5 Incremental Collection in the Message Area

Organization of the message area Figure 2 shows the organization of the message area when performing incremental GC of the young generation.

- The old generation, which is collected by a mark-sweep collector, consists of n pages (each page being $32Kwords$ in size). Allocation uses first-fit in the free-list. If there is not a large enough free area in this list, a garbage collection of the old generation is triggered. If, after a non-moving collection cycle, there is less than 25% free in the old generation, we allocate a new page in order to reduce the risk of triggering another collection soon.

- The young generation consists of two equal-sized parts, the *nursery* and the *from-space*. The size of each part, Σ, is constant and in our implementation we have chosen $\Sigma = 100K\ words$. The nursery is used by the mutator as the allocation area during a collection cycle. The from-space is used in the incremental copying collection; the *to-space* is the old generation.

- We also use an area (currently an array of size Σ) of *forwarding pointers* (denoted as Fwd in Figure 2). The reason is that the mutator does not expect to find forwarding pointers in the place of objects. Since the mutator can access objects in the from-space during a collection cycle, forwarding pointers cannot be stored in this area. This would require either making the mutator perform a test on each heap pointer dereferencing and paying the corresponding cost, or the systematic use of *indirection* (as in [10]) and employing a *read barrier* mechanism to maintain the to-space invariant (as for example in [5]), which also has a non-trivial associated cost.

  In our implementation, the size of the area for the forwarding pointers is *constant*. It could be further reduced if a different (resizeable) data structure is used; however, we prefer the simplicity of implementation and constant access time that an array provides.

- Finally, we also use a bit array (the *black-map*) and a pointer into the nursery (the *allocation limit*), whose purposes and uses we describe below.

Terminology We use the term *collection stage* to refer to a contiguous period of incremental garbage collection, and the term *collection cycle* to refer to a complete collection of the young generation. After a collection cycle has finished, all live data has been rescued from the nursery and moved to the old generation. A collection cycle may include a *non-moving collection cycle*, since it is the garbage collector of the young generation that allocates in the old generation and is thus the one to trigger its collection.

5.1 The incremental collection algorithm

A new collection cycle begins with the from-space and the nursery switching roles and with all forwarding pointers being reset.

All processes are then marked as active (i.e., are placed in the *active queue*), the first process from this queue is picked up and a *snapshot* of its root set is taken. (The process does not need to be suspended to have its snapshot taken.) When all roots for this process have been rescued, the process is removed from the queue. During a collection cycle, inactive processes may become active again *only* by receiving a message from another active process. This effectively acts as a *write barrier*, albeit one with an extremely low cost; namely, one extra test for each entire send operation. (Note that if a sender process is not active, then either the message has been allocated in the message area after the collection has started, and thereby placed in the nursery, or the message has already been copied to the old generation.) The collection cycle will scan the root set as long as there are active processes that contain "new" live objects (i.e., objects in the from-space not already copied to the to-space). During a collection cycle, the collector might of course yield to the mutator as described below.

When a live object is found, and this object has not yet been forwarded, it is copied to the old generation and added to a stack of gray objects. A forwarding pointer for this object is placed in the forwarding pointer array. If the object has been previously forwarded, we update its reference in the root set to point to the new location for the object. When the active queue is empty, the collection cycle continues to process all the gray objects, in order to rescue their children. This in turn possibly puts more objects on the gray stack.

If during collection of the young generation, the old generation overflows, its non-moving incremental garbage collector is triggered. This collector uses its own tricolor scheme [32] implemented as follows. We use a stack of references to keep track of gray objects. We also use a bit array (the *black-map*) to mark objects as black (i.e., fully processed). The black-map is needed since there is no room for any mark-bits in the actual objects.

At the end of the collection cycle we also have to look through the objects in the nursery to update references to data which has been moved from the from-space by the collection (or possibly copy these objects). This is because the mutator can create references from objects in the nursery to objects in the from-space during a collection cycle.

5.1.1 Interplay between the mutator and the collector

In incremental tracing garbage collectors, the amount of work to be done in one collection cycle depends on the amount of live data when a *snapshot* of the root set is taken. Since we can not know this quantity, we have to devise a mechanism that allows us to control how much allocation the mutator is allowed to do between two collection stages. (Relying on user-annotations to specify such a quantity is neither safe nor a "user-friendly" option in the typical multi-thousand line application domain of ERLANG.)

As with all incremental collectors, a crucial issue is to decide how and when the switch between the mutator and the collector will occur. We use an *allocation limit* to interrupt the mutator (cf. Figure 2). When the mutator reaches this limit the collector is invoked. This is a cheap way to control the interleaving and furthermore imposes no additional overhead on the mutator. This is because, even in a non-incremental environment, the mutator checks against a limit anyway (the end of the nursery, N_{limit}). The allocation limit is updated in the end of each collection stage based on a calculated estimate as described below. To influence the interaction between the mutator and the collector, the user can choose between a *work-based* and a *time-based* approach, which update the allocation limit in different ways.

5.1.2 The work-based collector

The underlying idea is simple. In order for the mutator to allocate w_M words of heap, the collector must rescue w words of live data, where $w_M \leq w$. In our implementation, the value of w is user-specified. (However, regardless of the user setting, we ensure that $w_M \leq w$ in all collection stages.) The choice of w naturally affects the pause times of the collector; see Section 6.2. After each collection stage the allocation limit is updated to $N_{top} + w$, where N_{top} denotes the top of the nursery (i.e., its first free word; cf. Figure 2). Note that this is exact, rather than an estimate as in the case of the time-based collector below.

Since the area we collect, the from-space, has the same size as the nursery we can guarantee that the collection cycle ends before the nursery overflows and the mutator cannot allocate further. In fact, since this is a young generation and most of its data tends to die young, the collection cycle will most often be able to collect the from-space before significant allocation takes place in the nursery.

5.1.3 The time-based collector

In the time-based collector, the *collector time quantum*, denoted t, determines the time interval of each collection stage. After this quantum expires, the collector is interrupted and the mutator is resumed. In our implementation, t is specified (in $\mu secs$) by the user based on the demands of the application.[2]

To dynamically adjust the allocation limit, we keep track of the amount of work done during a collection stage. We denote this by ΔGC and since this is a tracing collector it is expressed in number of live words rescued, that is,

$$\Delta GC = rescued\ after\ GC - rescued\ before\ GC$$

Assuming the worst case scenario (that the entire from-space of size Σ is live), at the end of a collection stage we (conservatively) estimate how much of the total collection we managed to do. Then we, again conservatively, estimate how many more collection stages it will take to complete the collection cycle, provided we are able to continue to rescue live data at the same rate.

$$GC\_stages = \frac{\Sigma - rescued\ after\ GC}{\Delta GC}$$

[2]When needed, the collector is allowed some "free" extension, in order to update the reference counts of binaries and possibly clean up after itself. This deadline extension is typically very small; cf. Section 6.2.

We now get:

$$w_M = \frac{f}{GC\_stages}$$

where f is the amount of free memory in the nursery. Thus, we can now update the allocation limit to $N_{top} + w_M$.

5.2 Some optimizations

In the beginning of the collection cycle, all processes in the system are put in the active queue, in a more or less random order.[3] However, each time an active process receives a message, it is moved last in the queue (as if it were reborn). This way, we keep the busiest processes last in the queue and scan them as late as possible. The rationale for wanting to postpone their processing is three-fold:

1. avoid repeated re-activation of message-exchanging processes;
2. allow processes to execute long enough for their data to become garbage;
3. give processes a chance to die before we take a snapshot of their root set; in this way, we might actually avoid considering these processes.

Another way of postponing processing members of the active queue is to process the stack of gray objects after we are finished with each process (instead of processing all processes in the active queue first and then processing the complete gray stack).

In minor collections of the shared message area, we remember the top of the heap for each process and only consider as part of their root set data that has been created since the process was taken off the active-queue.

Finally, a very important optimization is to have process-local garbage collections record pointers into the message area in a remembered set. This way we avoid scanning the old generation of their local heaps.

5.3 Characteristics of the collector

First of all note that the collector does not require any header word in the objects in order to perform incremental copying collection in the young generation. Therefore, it imposes no overhead to allocation. The collector instead uses an extra space, namely the forwarding area, whose size is bounded by Σ. Recall that Σ does not increase during GC and is not affected by the allocation characteristics of the program which is being executed. In the old generation, the only extra overhead is one bit per word for the black-map. A dynamically resizeable stack is used for

[3] The queue order is actually determined by the age of the processes; oldest first.

Benchmark	Processes	Messages
worker	403	1,650
msort_q	16,383	49,193
adhoc	137	246,021
yaws	420	2,275,467
mnesia	1,109	2,892,855

Table 1: Concurrency characteristics of benchmarks.

the gray objects. Note that for the frequent collections of the young generation, the size of this gray stack is bounded by $\Sigma/2$. The space overhead of the incremental collector is quite low.

Without incrementality, the collector behaves as a *snapshot-at-the-beginning* algorithm [47, Section 3.3.1]. As explained above, in the incremental collector we postpone taking the snapshot of processes in the active queue as long as possible. By incrementally taking partial snapshots of the root set, that is, only one process at a time, we allow the remaining processes to create more garbage as we collect the process at the head of the queue. This means that we will most likely collect more garbage than a pure snapshot-at-the-beginning collector.

An unfortunate side-effect of the root set minimization effort described above is that since we do not actually scan the old generation of process-local heaps during root-scanning, but only the set of references to the message area recorded during process-local garbage collection, some of the rescued objects might actually be already dead at the start of the collection. An object may therefore be kept in the message area for a number of collection cycles until a major process-local garbage collection updates the remembered set of objects (or the process dies) and triggers the next collection cycle of the message area to finally remove the object. This however is an inherent drawback of all generational schemes.

6 Measurements

The benchmarks For the performance evaluation we used two synthetic benchmarks and three ERLANG applications with a high degree of concurrency from different domains:

worker Spawns a number of worker processes and waits for them to return their results. Each worker builds a data structure in several steps, generating a large amount of local, temporary data. The final data structure is sent to the parent process. This is an allocation-intensive program whose adversarial nature is a challenge for the incremental garbage collector.

msort_q A distributed implementation of merge sort. Each process receives a list, implicitly splits it into two sublists by indexing into the original list, and spawns two new processes for sorting these lists (which are passed to the processes as messages). Although this program takes a very small time to complete, we use it as a benchmark because it spawns a large number of simultaneously live processes (cf. Table 1) and thus its root set is quite large.

adhoc A framework for genetic algorithms. It solves deceptive problems while simulating a population of chromosomes using processes and applies crossovers and mutations. The AdHOC program[4] consists of about 8,000 lines of ERLANG code.

yaws A high-performance multi-threaded HTTP Web server where each client is handled by a separate ERLANG process. Yaws[5] contains about 4,000 lines of code (excluding calls to functions in Erlang/OTP libraries such as HTTP, SSL, etc.). We used `httperf` [35] to generate requests for Yaws.

mnesia The standard TPC-B database benchmark for the Mnesia distributed database system. Mnesia consists of about 22,000 lines of ERLANG code. The benchmark tries to complete as many transactions as possible in a given time quantum.

Some more information on these benchmarks (number of processes spawned and messages sent between them) is shown in Table 1.

The performance evaluation was conducted on a dual processor Intel Xeon 2.4 GHz machine with 1 GB of RAM and 512 KB of cache per processor, running Linux. The kernel has been enhanced with the `perfctr` driver [38], which provides access to high-resolution performance monitoring counters on Linux and allows us to measure GC pause times in μs.

[4] ADHOC: Adaptation of Hyper Objects for Classification.

[5] YAWS : Yet Another Web Server; see `yaws.hyber.org`.

Benchmark	Local GCs	Message area GCs			
		$w = 2$	$w = 100$	$w = 1000$	$t = 1000$
worker	6.7 K	2.5 M	98.7 K	10 K	—
msort_q	357	79,190	1,716	174	222
adhoc	1.1 M	54,934	3,737	390	—
yaws	2.1 M	32,204	1,393	290	1,551
mnesia	892 K	12,581	671	219	775

Table 2: Number of GCs when using the two incremental collectors.

Benchmark	Mutator	Local GC	MA GC
worker	3,591	2,756	1,146
msort_q	174	3	29
adhoc	61,578	7,848	27
yaws	240,985	11,359	153
mnesia	53,276	4,487	88

Table 3: Mutator and total GC times (in ms) using the non-incremental collector.

6.1 Runtime and collector performance

To provide a base line for our measurements, Table 3 and Table 4 show time spent in the mutator, garbage collection times, and GC pause times for all benchmarks when using the non-incremental collector for the message area. Observe that the times in Table 3 are in ms while the times in Table 4 are in μs. Table 5 confirms that the time spent in the mutator and in performing garbage collection of process-local heaps is not affected by using the incremental collector for the message area. Depending on the configuration, the overhead for the incremental collector compared to the non-incremental collector ranges from a few percent to 2.5–3 times for most programs. The overhead is higher (5.6 times) for **worker** which is a program that was constructed to spend a significant part of its time allocating in (and garbage collecting) the message area.

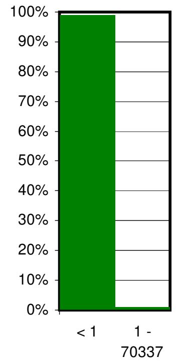
Considering total execution time, the performance of applications is practically unaffected by the extra overhead of performing incremental GC in the message area. Even for the extreme case of **worker**, which performs 2.5 million incremental garbage collections of the message area when $w = 2$ (cf. Table 2), its total execution time is 1.7 times that with non-incremental GC.

	Local GC (μs)			Message area GC (μs)		
Benchmark	Max	Mean	G.Mean	Max	Mean	G.Mean
worker	7,673	395	68	178,916	89,811	77,634
msort_q	577	9	4	16,263	9,807	11,646
adhoc	88	6	7	1,650	1,242	1,174
yaws	370	8	7	1,088	649	636
mnesia	4,722	4	5	1,413	485	458

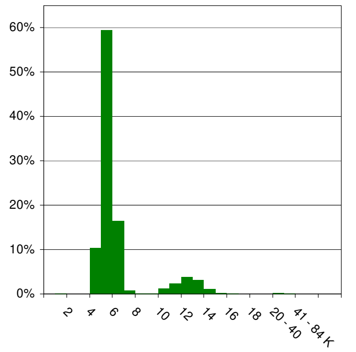
Table 4: Pause times (in μs) using the non-incremental collector.

			Message area (MA) GC		
Benchmark	Mutator	Local GC	$w = 2$	$w = 100$	$w = 1000$
worker	3,560	2,798	6,445	6,296	6,341
msort_q	164	3	54	34	33
adhoc	61,045	8,194	244	203	78
yaws	237,629	11,728	373	374	242
mnesia	52,906	4,439	182	164	156

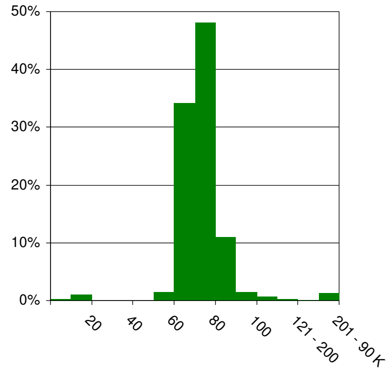
Table 5: Mutator times and total GC times (in ms) using the incremental (work-based) collector.



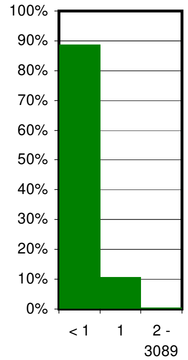
(a) **worker** ($w = 2$)



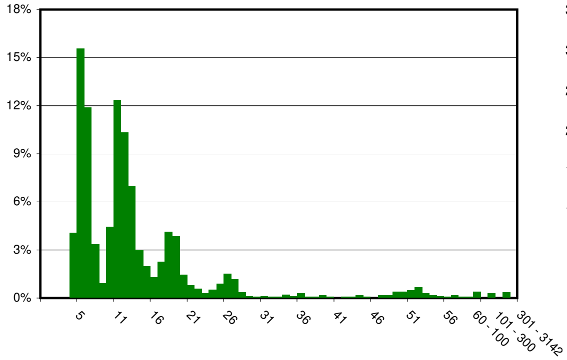
(b) **worker** ($w = 100$)



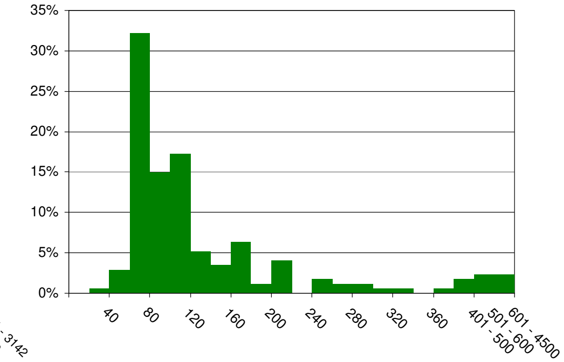
(c) **worker** ($w = 1000$)



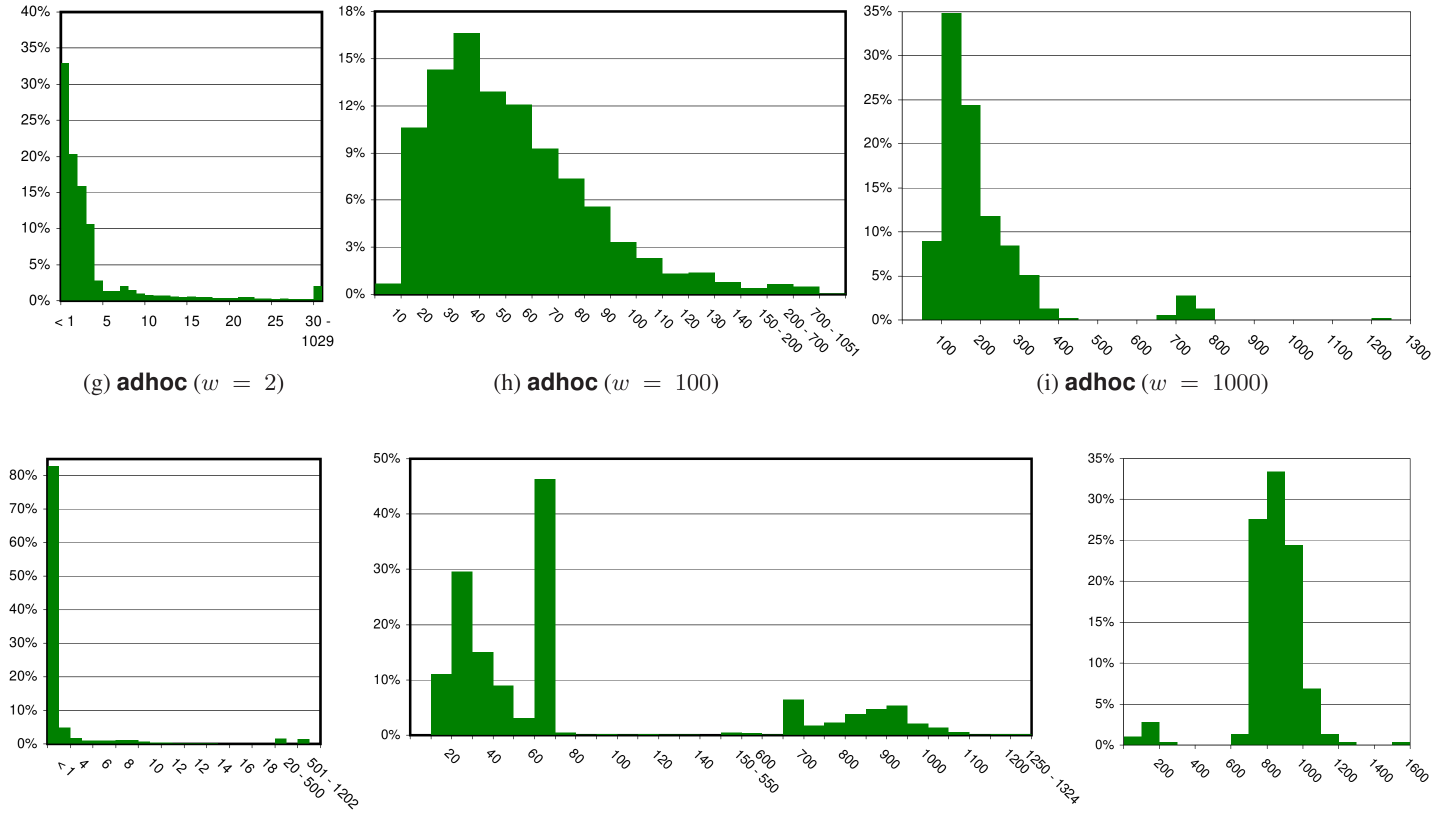
(d) **msort_q** ($w = 2$)



(e) **msort_q** ($w = 100$)



(f) **msort_q** ($w = 1000$)



(g) **adhoc** ($w = 2$) (h) **adhoc** ($w = 100$) (i) **adhoc** ($w = 1000$)

(j) **yaws** ($w = 2$) (k) **yaws** ($w = 100$) (l) **yaws** ($w = 1000$)

Figure 3: Distribution of pause times (in μs) for the work-based incremental collector.

	Local GC (μs)				Message area GC (μs)		
Benchmark	Max	Mean	G.Mean	w	Max	Mean	G.Mean
worker	6,891	390	68	2	70,337	2	0
				100	83,450	63	7
				1000	96,450	635	72
msort_q	611	8	4	2	3,089	0	0
				100	3,142	19	11
				1000	4,511	204	110
adhoc	125	6	7	2	1,029	3	2
				100	1,051	53	46
				1000	1,233	202	158
yaws	266	8	8	2	1,202	9	1
				100	1,324	268	36
				1000	1,586	836	853
mnesia	4,751	4	5	2	1,014	14	1
				100	1,027	244	43
				1000	1,212	714	787

Table 6: Pause times (in μs) for the incremental (work-based) collector with different values of w.

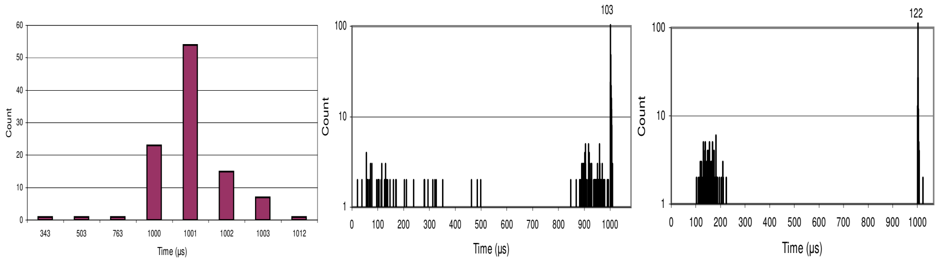
6.2 Garbage collection pause times

Table 6 shows pause times for the incremental work-based collector using three different choices of w, collecting $2, 100$, and 1000 words, respectively. As expected, for most benchmarks, the incremental garbage collector significantly lowers GC pause times, both their maximum and mean values (the columns titled G.mean show the geometric mean of all pause times) compared with the non-incremental collector (cf. the last three columns of Table 4). The maximum pause times of **yaws** (for $w = 100$ and 1000) are the only slight exception to this rule, and the only explanation we can offer for this behavior is that perhaps message live data is hard to come by in this benchmark.

The mean GC pause time values, in particular the geometric means, show a more consistent behavior. In fact, one can see a correlation between the value of w and the order of pause times in μs.

The distribution of pause times (in μs) for the benchmarks using the work-based incremental collector is shown in Figure 3.[6] The majority of collection stages are very fast, and only a very small percentage of the collections might be a problem for a real-time application. On the other hand, a work-based collector whose

[6] **mnesia** is not included in Figure 3 as its pause times do not show anything interesting.



(a) **msort_q** ($t = 1000\mu s$) (b) **yaws** ($t = 1000\mu s$) (c) **mnesia** ($t = 1000\mu s$)

Figure 4: Counts of pause times (in μs) for the time-based incremental collector.

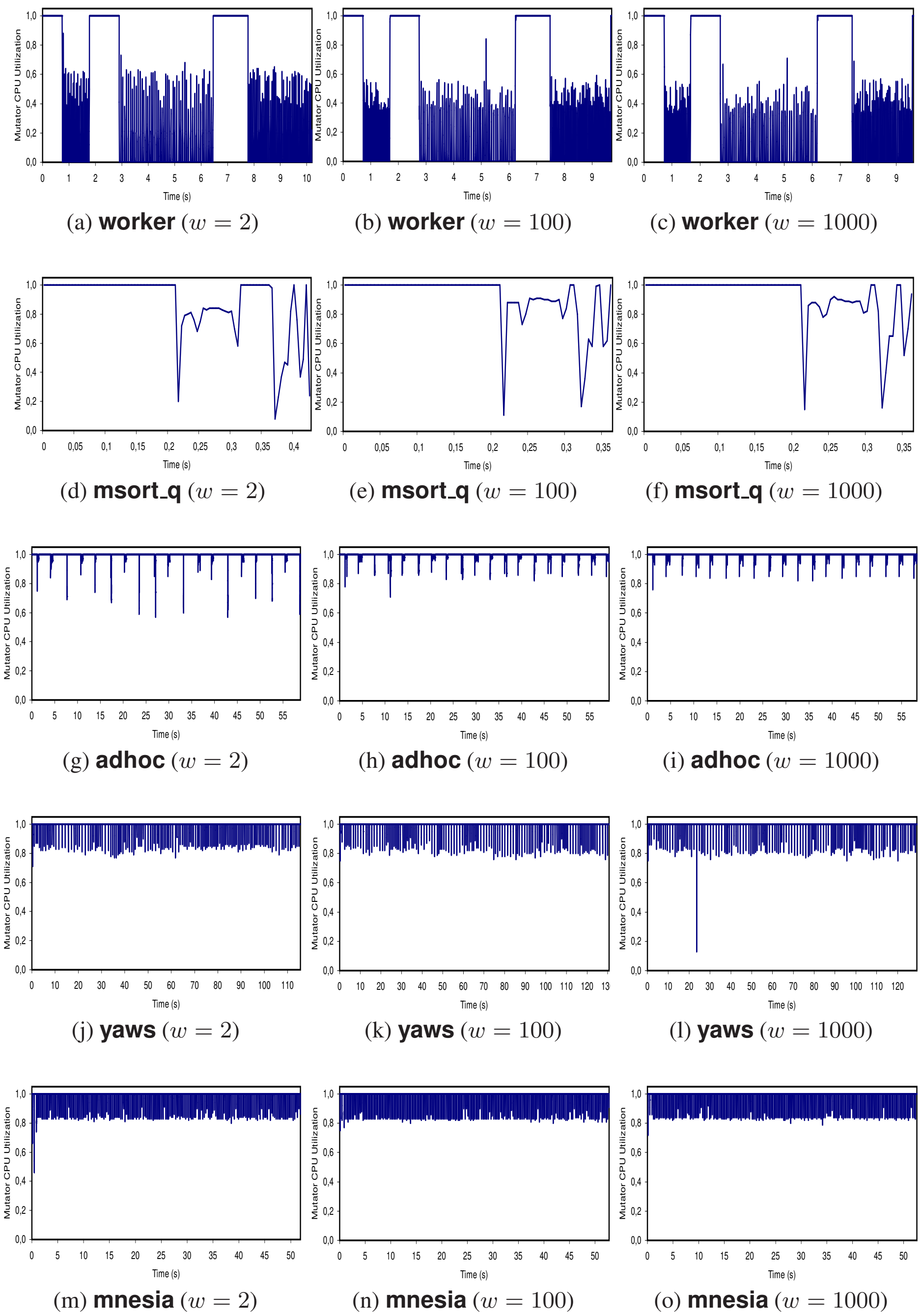
notion of work is defined in terms of “words rescued” naturally cannot guarantee an upper limit on pause times, as live data to rescue might be hard to come by.

A time-based incremental collector can in principle avoid this problem; see [5]. Care of course must be taken to detect the case when the mutator is allocating faster than the collector can reclaim, and take an appropriate action. Figure 4 (cf. also Table 2) shows counts of GC pauses when running three of the benchmarks programs using the time-based incremental garbage collector with a t value of $1ms$ ($1000\mu s$). As mentioned in Footnote 2, when needed, the collector is allowed some small deadline extension, in order to possibly clean up after itself. This explains why there is a small number of values above $1000\mu s$. Note that in Figures 4(c) and 4(b) the number of GCs (the Y axis) is in logarithmic scale.

6.3 Mutator utilization

In any time window, the notion of *mutator utilization* is defined as the fraction of time that the mutator executes; see [15].

Figure 5 shows mutator utilization for the programs we used as benchmarks when using the work-based incremental collector for different values of w. The two synthetic benchmarks exhibit interesting patterns of utilization. As expected, the **worker** benchmark suffers from poor mutator utilization since it is designed to be allocation-demanding and be a serious challenge for the incremental collector. (The first interval of high utilization is the time before the first collection is triggered and the remaining two are periods after a collection cycle has finished and there is free space left in the nursery that the mutator can use for its allocation needs.) Similarly, the mutator utilization of **msort_q** drops significantly when live data in the message area is hard to come by. On the other hand, the mutator utilization of the three “real” programs is good — even for $w = 2$, although for **yaws** and **mnesia** this is apparent only with the time axis stretched out; Figure 6 shows the same data as Figure 5(k) but only for a portion of the total time needed to run



(a) **worker** ($w = 2$) (b) **worker** ($w = 100$) (c) **worker** ($w = 1000$)

(d) **msort_q** ($w = 2$) (e) **msort_q** ($w = 100$) (f) **msort_q** ($w = 1000$)

(g) **adhoc** ($w = 2$) (h) **adhoc** ($w = 100$) (i) **adhoc** ($w = 1000$)

(j) **yaws** ($w = 2$) (k) **yaws** ($w = 100$) (l) **yaws** ($w = 1000$)

(m) **mnesia** ($w = 2$) (n) **mnesia** ($w = 100$) (o) **mnesia** ($w = 1000$)

Figure 5: Mutator utilization for the work-based incremental collector.

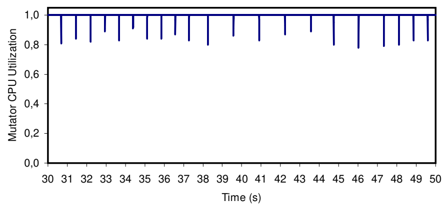
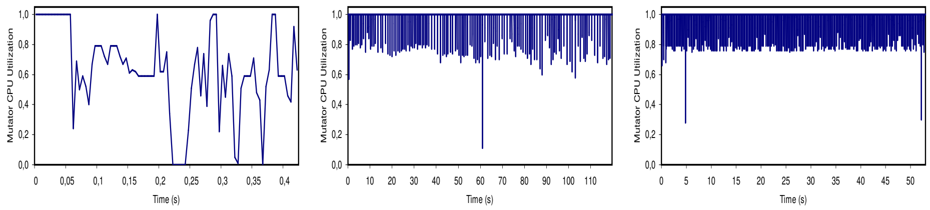


Figure 6: Mutator utilization of **yaws** ($w = 100$) for the work-based incremental collector shown in detail.



(a) **msort_q** (b) **yaws** (c) **mnesia**

Figure 7: Mutator utilization for the time-based ($t = 1000\mu s$) incremental collector.

the benchmark.

Mutator utilization for the time-based incremental collector is shown in Figure 7. For both **yaws** (mainly) and **mnesia** the utilization using the time-based collector is slightly worse than that when using the work-based one. The choice of an otherwise small, but compared with the total execution time relatively high in this case, t value ($1ms$) jeopardizes the mutator utilization of **msort_q**.

7 Related Work

Runtime system organization By now, several works have suggested detecting thread-local objects via static *escape analysis*, mainly of Java programs; notable among them are [8, 18, 39]. The goal has been to identify, conservatively and at compile time, the objects that are only going to be accessed by their creating thread and allocate them on the thread-local stack, thereby avoiding synchronization for these objects. In fact, the analysis of [39] is exploited in [41] by suggesting the use of thread-local heap chunks for non-escaping objects and a shared (portion of the) heap for all other data. Thread-local heaps for Java have also been advocated in [22], this time guided by information gathered by a profiler rather than by static analysis.

Note that, mainly because of the differences in the semantics of Java and ERLANG, all the above works attack the problem of memory allocation in the opposite direction than we do. Rather than allocating in thread-local heaps by default and using analysis to determine which objects are possibly shared, they try to determine objects that will *only* be accessed by their creating thread and allocate them in a thread-local memory area. In contrast, the message analysis that guides our allocator, identifies data that will probably be used in a message, enabling a speculative optimization that allocates data in the shared message area, thereby eliminating the need for copying at send time and making it possible to remove run-time checks altogether. The closest relative of our work is the memory architecture described in [21] which uses thread-local allocation for immutable objects in Caml programs.

Memory management of ERLANG programs The soft real-time concerns of the ERLANG language call for bounded-time GC techniques. One such technique, based on a mark-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 described in [1], but imposes a significant overhead and was never fully implemented. Similarly, [24] describes 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 never made it to an Etos distribution.

Incremental and real-time GC techniques In the context of other (strict, concurrent) functional language implementations, the challenge has been to achieve low GC latency without paying the full price in performance that a guaranteed real-time garbage collector usually requires. Notable among them is the quasi real-time collector of Concurrent Caml Light [21] which combines a fast, asynchronous copying collector for the thread-specific young generations with a non-disruptive concurrent mark-sweep collector for the old generation (which is shared among all threads).

Many concurrent (real-time) garbage collectors for functional languages have also been proposed, either based on incremental copying [10, 27], or on *replication* [36] (see also [15] for a multi-processor version of one such collector). The main difference between them is that incremental collectors based on copying require a read barrier, while collectors based on replication do not. Instead, they capitalize on the copying semantics of (pure) functional programs, and incrementally replicate all accessible objects using a mutation log to bring the replicas up-to-date with changes made by the mutator.

An excellent discussion and analysis of the trade-offs between work-based and time-based incremental collectors appears in [5]. Our work, although done independently and in a very different context than that of [5], is quite heavily influenced by it, presentation-wise. Given the different semantics (copying versus sharing) of concurrency in ERLANG and Java, and the different compiler and runtime system implementation technologies involved in Erlang/OTP and in Jikes RVM, it is very difficult to do a fair comparison between the Metronome (the collector of [5]) and our incremental collector. As a rather philosophical difference, we do not ask the user to guide the incremental collector by specifying the maximum amount of simultaneously live data or the peak allocation rate over the time interval of a garbage collection. More importantly, it appears that our system is able to achieve significantly lower pause times and better mutator utilization than the Metronome. We believe this can mostly be attributed to the memory allocation strategy of the hybrid runtime system architecture which is local-by-default. On the other hand, the utilization of our collector is not as consistent as that of [5] for adversarial, synthetic programs,[7] but then again we are interleaving the collector and the mutator in a much finer-grained manner (e.g., collecting just 2 words) or we are forcing our collector to run in a considerably smaller collector quantum ($1ms$ versus $22.2ms$ which [5] uses).

8 Acknowledgments

This research has been supported in part by a grant from the Swedish Research Council (Vetenskapsrådet) and by the ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson and T-Mobile.

[7] Of course, this very much depends on the choice of these programs!

REFERENCES

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 Europe, Herfordshire, Great Britain, 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*, pages 333–346. USENIX Association, 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, New York, N.Y., June 2001. ACM Press.

[5] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–298, New York, N.Y., Jan. 2003. ACM Press.

[6] B. Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Conference Record of the 25th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'98)*, pages 25–37, New York, N.Y., Jan. 1998. ACM Press.

[7] B. Blanchet. Escape analysis for object oriented languages. Application to Java^{TM}. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 20–34, New York, N.Y., Nov. 1999. ACM Press.

[8] B. Blanchet. Escape analysis for Java^{TM}: Theory and practice. *ACM Trans. Prog. Lang. Syst.*, 25(6):713–775, Nov. 2003.

[9] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Conference on Object-Oriented Programming Sys-*

tems, Languages and Applications (OOPSLA'99), pages 35–46, New York, N.Y., Nov. 1999. ACM Press.

[10] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In G. L. Steele, editor, *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 256–262, New York, N.Y., 1984. ACM Press.

[11] R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, Sept. 2001.

[12] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 030, Information Technology Department, Uppsala University, Nov. 2000.

[13] R. Carlsson, K. Sagonas, and J. Wilhelmsson. Message analysis for concurrent languages. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, number 2694 in LNCS, pages 73–90, Berlin, Germany, June 2003. Springer.

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

[15] 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, New York, N.Y., June 2001. ACM Press.

[16] 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, New York, N.Y., 1998. ACM Press.

[17] J.-D. Choi, M. Gupta, M. Serrano, V. C. Shreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 1–19, New York, N.Y., Nov. 1999. ACM Press.

[18] J.-D. Choi, M. Gupta, M. Serrano, V. C. Shreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Prog. Lang. Syst.*, 25(6):876–910, Nov. 2003.

[19] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960.

[20] A. Deutsch. On the complexity of escape analysis. In *Conference Record of the 24th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 358–371, New York, N.Y., Jan. 1997. ACM Press.

[21] 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, New York, N.Y., Jan. 1993. ACM Press.

[22] T. Domani, G. Goldshtein, E. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In D. Detlefs, editor, *Proceedings of ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, pages 76–87, New York, N.Y., June 2002. ACM Press.

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

[24] 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, New York, N.Y., Oct. 1998. ACM Press.

[25] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–247, New York, N.Y, June 1993. ACM Press.

[26] 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.

[27] L. Huelsbergen and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming*, pages 73–82, New York, N.Y., May 1993. ACM Press.

[28] 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.

[29] 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, New York, NY, Sept. 2000. ACM Press.

[30] E. Johansson, K. Sagonas, and J. Wilhelmsson. Heap architectures for concurrent languages using message passing. In D. Detlefs, editor, *Proceedings of ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, pages 88–99, New York, N.Y., June 2002. ACM Press.

[31] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: solved? In *ISMM '98: Proceedings of the First International Symposium on Memory Management*, pages 26–36. ACM Press, 1998.

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

[33] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, 1983.

[34] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[35] D. Mosberger and T. Jin. httperf—a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, Dec. 1998.

[36] 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, New York, N.Y, June 1993. ACM Press.

[37] 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, New York, N.Y., July 1992. ACM Press.

[38] M. Pettersson. Linux x86 performance-monitoring counters driver. http://user.it.uu.se/~mikpe/linux/perfctr/.

[39] E. 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.

[40] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, New York, N.Y., June 1988. ACM Press.

[41] B. 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., Oct. 2000. ACM Press.

[42] L. Stein and D. MacEachern. *Writing Apache Modules with Perl and C*. O'Reilly & Associates, 1999.

[43] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, Feb. 1997.

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

[45] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167. ACM Press, 1984.

[46] 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.

[47] 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, Berlin, Germany, Sept. 1992. Springer-Verlag. See also expanded version as Univ. of Texas Austin technical report submitted to ACM Computing Surveys.

Recent licentiate theses from the Department of Information Technology

2003-010	Dan Wallin: *Exploiting Data Locality in Adaptive Architectures*
2003-011	Tobias Amnell: *Code Synthesis for Timed Automata*
2003-012	Olivier Amoignon: *Adjoint-Based Aerodynamic Shape Optimization*
2003-013	Stina Nylander: *The Ubiquitous Interactor - Mobile Services with Multiple User Interfaces*
2003-014	Kajsa Ljungberg: *Numerical Methods for Mapping of Multiple QTL*
2003-015	Erik Berg: *Methods for Run Time Analysis of Data Locality*
2004-001	Niclas Sandgren: *Parametric Methods for Frequency-Selective MR Spectroscopy*
2004-002	Markus Nordén: *Parallel PDE Solvers on cc-NUMA Systems*
2004-003	Yngve Selén: *Model Selection*
2004-004	Mohammed El Shobaki: *On-Chip Monitoring for Non-Intrusive Hardware/Software Observability*
2004-005	Henrik Löf: *Parallelizing the Method of Conjugate Gradients for Shared Memory Architectures*
2004-006	Stefan Johansson: *High Order Difference Approximations for the Linearized Euler Equations*
2005-001	Jesper Wilhelmsson: *Efficient Memory Management for Message-Passing Concurrency*

UPPSALA
UNIVERSITET

Department of Information Technology, Uppsala University, Sweden