Message Analysis for Concurrent Programs Using Message Passing

RICHARD CARLSSON, KONSTANTINOS SAGONAS, and JESPER WILHELMSSON
Uppsala University

We describe an analysis-driven storage allocation scheme for concurrent systems that use message passing with copying semantics. The basic principle is that in such a system, 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 communal memory architecture.

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

General Terms: Languages, Performance, Measurement

Additional Key Words and Phrases: Static analysis, runtime systems, concurrent languages, message passing, Erlang

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 e.g. in Java, C#, and Modula-3; using synchronous message passing on (usually typed) channels as e.g. in Concurrent ML and Concurrent Haskell; via rendezvous as in Ada and Concurrent C; using asynchronous message passing as in Erlang; or through shared logical variables as in concurrent logic programming languages, including Mozart/Oz. Most of these languages typically also require support for automatic memory management, usually

Authors' address: Department of Information Technology, Uppsala University, Box 337, 75105 Uppsala, Sweden, email: {richardc,kostis,jesperw}@it.uu.se.
Corresponding author: Richard Carlsson, richardc@it.uu.se.
This is a significantly extended version of a paper that appeared in SAS'03: Proceedings of the Tenth Static Analysis Symposium.

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 e.g. [Jones and Lins 1996] or [Wilson 1992] for excellent surveys of this area.

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 of the runtime system. 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 [Johansson et al. 2002], 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 systems are both principled and pragmatic. Pragmatic because we are involved in the development of a production-quality system of this kind, the Erlang/OTP system, which is heavily used as a platform for the development of highly concurrent (thousands of light-weight 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, resulting in higher performance and much better scalability. Furthermore, from a software engineering perspective the copying semantics offers isolation between processes and makes distribution transparent, both important properties.

Our contributions. Our first contribution, which motivates our static program analysis, is in the area of runtime system organization. Based on the properties of the two different memory architectures investigated in [Johansson et al. 2002], we describe two variants of a *hybrid* 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). This hybrid architecture allows interprocess communication to occur without actual copying when shared memory is available, 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 what data will be used in messages, and which can guide the memory allocation in the hybrid architecture. One of the main advantages of the analysis is that it tends to perform well even when it is run on a single module at a time, rather than on the whole program (although this is also possible). We present a mini-Erlang language with a formal semantics and sketch a correctess proof for the message analysis in terms of this. The analysis does not rely on the presence of type information, and does not sacrifice precision when handling list types, despite its simplistic representation of data structures. We show that although the analysis has cubic worst-case time complexity, it tends to be fast enough in practice.

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

Summary of contents. We begin by introducing Erlang and briefly reviewing the pros and cons of alternative heap architectures for concurrent languages. Section 3 goes into more detail about implementation choices in the hybrid architecture. Section 4 describes the message analysis and its relation to escape analysis, and Section 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 and memory usage. Finally, Section 7 discusses related work and Section 8 concludes.

2. PRELIMINARIES

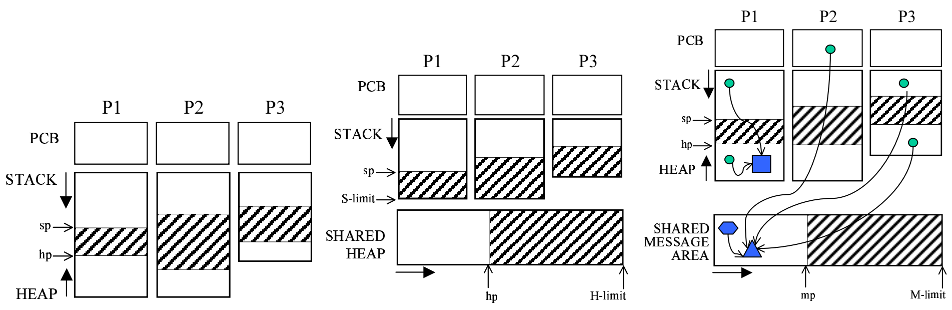
2.1 Erlang and Core Erlang

Erlang [Armstrong et al. 1996] 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 the recipient may be located on any machine in a network. 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*. In more detail, this means that any loaded module can at any time be replaced, by simply loading new code for that module. All calls to the module will then be redirected to the new version of the code. Processes executing code in the older version will remain alive, and will migrate to the new code as soon as they execute an inter-module tail call. For example, a typical server process will be executing an event loop which, af-



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

Fig. 1. Different runtime system architectures for concurrent languages

ter handling a single event, will make a self-recursive tail call qualified with the module name, ensuring a switch whenever new code is loaded.[1] While dynamic code replacement is considered an essential feature for real-world Erlang applications, it constitutes a major obstacle for whole-program and cross-module analyses; currently, modules are always compiled independently of one another.

Core Erlang [Carlsson et al. 2000; Carlsson 2001] 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 Erlang as an intermediate form on which static analyses and optimizations may be performed before low level code is produced. While Erlang has quite unusual and complicated variable scoping rules, fixed-order evaluation, and does not allow function definitions to be nested, Core Erlang is 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 [Johansson et al. 2002] 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:

[1] In order to cleanly migrate to new code, no return addresses to old code may remain on the stack. The Erlang run-time system contains support for killing processes that are still depending on old code, when necessary.

Process-centric. This is currently the default confi guration of Erlang/OTP. Interprocess communication requires copying of messages, i.e., 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 immediately. 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 [Domani et al. 2002].

Communal. The biggest advantage is very fast ($O(1)$) interprocess communication, simply consisting of passing a pointer to the receiving process, low memory requirements due to message sharing, and low fragmentation. 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. Furthermore, the communal architecture does not scale well to a multithreaded or multiprocessor implementation, since locking would be required in order to garbage collect the shared memory in a parallel setting; see e.g. [Cheng and Blelloch 2001] for a recent treatment of the subject.

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 garbage collection of the 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 heap can be reclaimed by simply attaching it 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, i.e., used as messages. This can be achieved by user annotations on the source code, by dynamically monitoring the creation of data as proposed by Domani et al. [2002], or by a static analysis as we describe in Section 4.

Note that these runtime system architectures are applicable to all concurrent systems that use message passing. Their advantages and disadvantages do not in any way depend on characteristics of the Erlang language or its current 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 multithreaded system, this allows collection of 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 of the runtime system 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).

This pointer directionality property is also crucial for our choice of memory allocation strategy, since it makes it easy to test at runtime whether or not a data structure resides in the shared area by making a simple $O(1)$ pointer comparison. (This requires that the shared message area is contiguous, e.g. using a compacting garbage collector.)

3.1 Allocation strategies

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

Local allocation of non-messages. 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 Steensgaard [2000]. 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 on the process-local heaps, and the behaviour of the hybrid architecture with this allocation strategy reduces to that of the communal 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 (as noted above, this is an $O(1)$ operation in our system), and otherwise copies the locally allocated parts to the shared heap. Furthermore, if program analysis can determine that a message operand *must* already be on the shared heap, the test can be statically eliminated. Without such analysis, the behaviour 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 overapproximates 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 could introduce unnecessary copying, as further explained in Section 5.2.

We have chosen to implement and evaluate the second strategy, which to the best of our knowledge has not been studied previously. Because Erlang modules are typically separately compiled, and any module can be replaced at any time during program execution, any data that might be passed across module boundaries will in general have to be regarded as escaping. Thus, the first strategy above is less likely to make good use of both the local heaps and the shared heap.

3.2 Copying of messages

If the second strategy is used, as in our implementation of the hybrid system, we must be prepared to copy (parts of) a message as necessary to ensure the pointer directionality invariant. Since we do not know how much of the message needs to be copied and how much already resides in the shared area, we cannot easily 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 is simply copied to the allocated space, and all pointer fields are initialized to Nil. The latter is necessary because the message 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 at some point, the garbage collector is called on-the-fly to make room. If a (mostly) copying garbage collector is used, as is currently 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.

An alternative algorithm first scans the message to find the size of the required memory. This simplifies the copying part, because garbage collection cannot occur in mid-copy. However, our measurements showed that this algorithm (which is used in the process-centric system) has very bad performance in the hybrid system. The reason is that in the hybrid system, each pointer needs to be tested in order to determine whether the object pointed to is already in the shared heap (in which case it will not be copied). When separating the size measurement from the copying, this test must be done twice for each pointer (once when measuring and once when copying), rather than only once.

4. MESSAGE ANALYSIS

To use the hybrid architecture without user annotations on what is to be allocated on the process-local heap and on the shared heap, respectively, program analysis is necessary. If data were to be allocated on the shared heap by default, we would need to single out the data which is guaranteed to not be included in any message, so it can be allocated on the local heap. This amounts to *escape analysis* of process-local data; see e.g. [Blanchet 2003; Bogda and Hölzle 1999; Choi et al. 2003].

However, if data is by default allocated on the local heaps, we instead want to identify data that is likely 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 form of analysis as *message analysis*. Note that because copying will always be invoked in the rewritten program whenever some part of a message might be residing on a process-local heap (cf. Section 5.2), both under- and overapproximation of the set of run-time message constructors is in itself safe. In our current implementation of message analysis, we usually overapproximate the set of constructors that could be messages, but this is not a requirement – underapproximation will have no ill effects apart from increased copying and unnecessary use of the local heaps for message data.

4.1 The analyzed language

Although our analyses have been implemented for the complete Core Erlang language, for the purposes of this article, the details of Core Erlang are unimportant. To keep the exposition simple, we instead define a sufficiently powerful language of A-normal forms [Flanagan et al. 1993], 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.

$$
\begin{array}{rcllll}
c & \in & \mathit{Const} & & \text{Constants (Atoms, Integers, Pids and } \mathit{nil}\text{)} \\
x & \in & \mathit{Var} & & \text{Variables} \\
e & \in & \mathit{Expr} & & \text{Expressions} \\
l & \in & \mathit{Label} & & \text{Labels, including } \mathit{xcall} \text{ and } \mathit{xlambda} \\
o & \in & \mathit{Primops} & & \text{Primitive operations } (\texttt{==}, \texttt{>}, \texttt{is\_nil}, \texttt{is\_cons}, \texttt{is\_tuple}, \ldots) \\
t & \in & \mathit{Term} & = & \mathit{Const} \cup \{t_1 : t_2\} \cup \{\texttt{\{}t_1, \ldots, t_n\texttt{\}} \mid n \geq 0\} \cup \mathit{Clos} \\
\rho & \in & \mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Term} \\
 & & \mathit{Clos} & = & \{\langle e, \rho, \rho' \rangle \mid e \in \mathit{Expr},\ \rho, \rho' \in \mathit{Env}\}
\end{array}
$$

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

Fig. 2. A mini-Erlang language

A program is any expression e that does not contain free variables. It is assumed that all variables in the program are uniquely named. 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 this assumption,[2] and the extension to handle general primops is straightforward.

Atoms (symbols) are written within single-quotes; the atoms `'true'` and `'false'` are used to represent boolean values. The empty list (*nil*) is written `[]`, and is not an atom. Each constructor in the program, as well as each call site and lambda expression, is given a unique label l. Tuple constructors are written $\texttt{\{}v_1, \ldots, v_n\texttt{\}}$, for all $n \geq 0$. Since the language is dynamically typed, the second argument of a list constructor $v_1 : v_2$ might not always be a list, but in typical Erlang programs all lists are proper.

Recursion is introduced with the explicit fixpoint operator $\texttt{fix}\ (\lambda x.e)$. Operators `hd` and `tl` select the first (head) and second (tail) element, respectively, of a list constructor. An 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 and then immediately returns, yielding a new unique process identifier ("pid"). When evaluation of a process terminates, the final result is discarded. The send operator $v_1\ \texttt{!}\ v_2$ sends message v_2 asynchronously to the process identified by pid v_1, returning 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 but sufficiently general model of the concurrent semantics of Erlang.

A big-step operational semantics for the analyzed language is shown in Figure 3. For simplicity, the state component σ has been left out where it is not affected. We write $\rho + \rho'$ for the extension of one mapping by another, and $\rho[x \mapsto y]$ denotes the extension of ρ by the function which maps x to y. Recursion is handled by finite unfolding in the vein of The Definition of Standard ML [Milner et al. 1997]: a closure is represented by a triple $\langle e, \rho, \rho' \rangle$

[2]In our actual implementation, we need to handle the fact that Erlang supports arbitrary-precision integers ("bignums") which are boxed and stored on the heap unless they fit into a single word including tag bits. Furthermore, on 32-bit machines, floating-point numbers are always boxed. As a consequence, the analysis has to conservatively assume that most arithmetic operations possibly return a heap allocated object. In our context, this somewhat limits the number of run-time copying checks that can be statically eliminated.

$$\rho \vdash c \rightarrow c \qquad \text{[Constant]}$$

$$\rho \vdash x \rightarrow \rho(x) \qquad \text{[Var]}$$

$$\frac{\rho \vdash v_1 \rightarrow t_1 \quad \rho \vdash v_2 \rightarrow t_2}{\rho \vdash v_1 \!:\! v_2 \rightarrow t_1 \!:\! t_2} \qquad \text{[Cons]}$$

$$\frac{\rho \vdash v_1 \rightarrow t_1 \quad \cdots \quad \rho \vdash v_n \rightarrow t_n}{\rho \vdash \{v_1, \ldots, v_n\} \rightarrow \{t_1, \ldots, t_n\}} \qquad \text{[Tuple]}$$

$$\frac{\rho \vdash v \rightarrow t_1 \!:\! t_2}{\rho \vdash \texttt{hd}\ v \rightarrow t_1} \qquad \text{[Head]}$$

$$\frac{\rho \vdash v \rightarrow t_1 \!:\! t_2}{\rho \vdash \texttt{tl}\ v \rightarrow t_2} \qquad \text{[Tail]}$$

$$\frac{\rho \vdash v \rightarrow \{t_1, \ldots, t_n\} \quad n \geq k}{\rho \vdash \texttt{element}_k\ v \rightarrow t_k} \qquad \text{[Element]}$$

$$\rho \vdash (\lambda x.e) \rightarrow \langle (\lambda x.e), \rho, [] \rangle \qquad \text{[Lambda]}$$

$$\frac{\mathit{rec}(\rho[x \mapsto \langle e, \rho, [] \rangle]) \vdash e, \sigma \rightarrow t, \sigma'}{\rho \vdash \texttt{fix}\ (\lambda x.e), \sigma \rightarrow t, \sigma'} \qquad \text{[Fix]}$$

$$\frac{c \in \mathit{Const}}{\rho \vdash \texttt{primop}\ o(v_1, ..., v_n) \rightarrow c} \qquad \text{[Primop]}$$

$$\frac{\rho \vdash v \rightarrow \texttt{'true'} \quad \rho \vdash e_1, \sigma \rightarrow t, \sigma'}{\rho \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2, \sigma \rightarrow t, \sigma'} \qquad \text{[If-True]}$$

$$\frac{\rho \vdash v \rightarrow \texttt{'false'} \quad \rho \vdash e_2, \sigma \rightarrow t, \sigma'}{\rho \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2, \sigma \rightarrow t, \sigma'} \qquad \text{[If-False]}$$

$$\frac{\rho \vdash b, \sigma \rightarrow t, \sigma' \quad \rho[x \mapsto t] \vdash e, \sigma' \rightarrow t', \sigma''}{\rho \vdash \texttt{let}\ x = b\ \texttt{in}\ e, \sigma \rightarrow t', \sigma''} \qquad \text{[Let]}$$

$$\frac{\rho \vdash v_1 \rightarrow \langle (\lambda x.e), \rho', \rho'' \rangle \quad \rho \vdash v_2 \rightarrow t \quad (\rho' + \mathit{rec}\ \rho'')[x \mapsto t] \vdash e, \sigma \rightarrow t', \sigma'}{\rho \vdash (v_1\ v_2), \sigma \rightarrow t', \sigma'} \qquad \text{[Call]}$$

$$\frac{p \in \mathit{Pid}}{\rho \vdash \texttt{spawn}\ (v_1\ v_2), \sigma \rightarrow p, \sigma \cup \{v_1, v_2\}} \qquad \text{[Spawn]}$$

$$\frac{\rho \vdash v_1 \rightarrow p \in \mathit{Pid} \quad \rho \vdash v_2 \rightarrow t}{\rho \vdash v_1 \,!\, v_2, \sigma \rightarrow t, \sigma \cup \{t\}} \qquad \text{[Send]}$$

$$\frac{t \in \mathit{Term}}{\rho \vdash \texttt{receive}, \sigma \rightarrow t, \sigma \cup \{t\}} \qquad \text{[Receive]}$$

$$\mathit{rec}\ \rho\ x = \begin{cases} \langle e, \rho', \rho \rangle, \text{if } \rho(x) = \langle e, \rho', \rho'' \rangle \\ \rho(x), \qquad \text{otherwise} \end{cases}$$

Fig. 3. Operational semantics

where e is the expression, ρ is the original environment, and ρ' is the recursive component which is unrolled by the function *rec* at least once before each lambda application; see the rules [Fix] and [Call].

For our purposes, it is sufficient to describe the behaviour of a single process at a time; therefore, the state σ is simply the set of shared terms. Sending a message adds the sent term to the shared area, and receiving a message yields an arbitrary term as result,[3] which is also placed in the shared area; see the [Send] and [Receive] rules. Spawning a new process adds both arguments to the shared area and yields a fresh process identifier; see the [Spawn] rule. (We leave out all details about how process identifiers are created; they are not heap allocated objects.)

4.2 General framework

The analyses we have this far implemented are first-order dataflow analyses, and are best understood as extensions of Shivers' 0-cfa control flow analysis [1988]. Indeed, we assume that control flow analysis has been done, so that:

—The label *xcall* represents all call sites external to the analyzed program, and the label *xlambda* represents all possible external lambdas.

—There is a mapping $\mathit{calls} : \mathit{Label} \rightarrow \mathcal{P}(\mathit{Label})$ from each call site label (including *xcall*) to the corresponding set of possible lambda expression labels (which may include *xlambda*), which constitutes an upper bound of the call graph of the program.

In particular, any lambda closure that may be part of the result of evaluating program e could be called from an external site. For $e = (\lambda x.x)^l$, this is the lambda labeled l; for $e = (\lambda x.\{(\lambda y.y)^{l'}, (\lambda z.z)^{l''}\})^l$, this is the set $\{l, l', l''\}$, and so on. Furthermore, an external lambda expression (i.e., not in e) could be called from a point in e if a closure is passed from an external site to a function in e, or is received in a message.

Although higher-order control flow analysis could be directly integrated with the dataflow analyses, the presentation is much simplified by the assumption that we have already determined the static call graph for the program.

The analysis domain V is defined as follows:

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

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

$$\sqsubseteq = \textstyle\bigcup_{i\geq 0} \sqsubseteq_i^*, \text{ where for all } i \geq 0 :$$

$$\begin{aligned}
&(s_1, w) \sqsubseteq_i (s_2, \top) \text{ if } s_1 \subseteq s_2 \\
&(s_1, \langle\rangle) \sqsubseteq_0 (s_2, \langle\rangle) \text{ if } s_1 \subseteq s_2 \\
&(s_1, \langle u_1, \ldots, u_n\rangle) \sqsubseteq_i (s_2, \langle v_1, \ldots, v_m\rangle) \\
&\qquad\qquad \text{if } i > 0 \wedge n \leq m \wedge s_1 \subseteq s_2 \wedge \forall j \in [1, n] : u_j \sqsubseteq_{i-1} v_j \\
&v_1 \sqsubseteq_i v_2 \text{ if } i > 0 \wedge v_1 \sqsubseteq_{i-1} v_2
\end{aligned}$$

[3] Since we only study a single process, other processes in the system can put arbitrary terms in the shared area at any time, but those terms are not of interest unless they are received by the current process.

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 $(\emptyset, \langle\rangle)$.

Let *Val* be a mapping from variables to abstract values, and *In* be a mapping from call site labels to abstract values. We defi ne the expression analysis functions $\mathcal{V}_v[\![\cdot]\!]$ and $\mathcal{V}_e[\![\cdot]\!]$ as follows:

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

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

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

where

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

and

$$\begin{aligned}
\mathit{head}\,(s, w) &= \begin{cases} (s, w) & \text{if } w = \top \\ v_1 & \text{if } w = \langle v_1, \ldots v_n \rangle, n \geq 1 \\ \bot & \text{otherwise} \end{cases} \\
\mathit{tail}\,(s, w) &= \begin{cases} (s, w) & \text{if } w = \top \vee w = \langle v_1, \ldots v_n \rangle, n \geq 1 \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

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

For each label l of a lambda expression $(\lambda x.e)^l$ in the program, define $\mathit{Out}(l) = \mathcal{V}_e[\![e]\!]$. Then for all call sites $(v_1 v_2)^l$ in the program, including spawns and the dummy external call labeled *xcall*, we have $\forall l' \in \mathit{calls}(l) : \mathit{Out}(l') \sqsubseteq \mathit{In}(l)$, and also $\forall l' \in \mathit{calls}(l) : \mathcal{V}_v[\![v_2]\!] \sqsubseteq \mathit{Val}(x)$, when l' is the label of $(\lambda x.e)^{l'}$. Finally, for each expression `let` $x = b$ `in` e we have $\mathcal{V}_b[\![b]\!] \sqsubseteq \mathit{Val}(x)$, and for each `fix` $(\lambda x.e)$, $\mathcal{V}_e[\![e]\!] \sqsubseteq \mathit{Val}(x)$. It is easy to verify from the definitions that the constraint system has a least fixpoint solution due to monotonicity.

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 $z = \mathit{cons}\ l\ x\ y$. If y is $(s, \langle v, \ldots \rangle)$, 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 for our purposes: in the above example, $x \sqsubseteq \mathit{head}\ z$ and $y \sqsubseteq \mathit{tail}\ z$; thus, as long as we are only interested in the labels occurring in the final set of abstract values at any particular point, and not of the actual substructure of those values, we may treat the tail of a list as we do here.

4.3 Termination, complexity and correctness

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

We define the limiting operator θ_k as:

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

where

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

The rules given in Section 4.2 are modified as follows, for some fixed k: 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" for a dynamically typed language.[4] Better precision could

[4]The escape analysis on lists of Park and Goldberg [1992] requires type information.

be achieved by a graph representation of data, as in the storage use analysis of Serrano and Feeley [1996]; our approximation was chosen to fit easily into our existing analysis framework and allow us to explore the usefulness of message analysis as a tool for guiding allocation of data.

It is straightforward to extend this framework to simultaneously perform a more precise control flow analysis than that of Shivers [1988] (which only uses sets of labels), building the call graph as we go, but doing so here would cloud the issues of this article. Also, Erlang programs tend to use fewer higher-order functions, in comparison with typical programs in e.g. Scheme or ML, so we expect that the improvements to the determined call graphs would be insignificant in practice.

Correctness of the analyses can be shown by structural induction over the derivations of $\vdash e, \sigma \rightarrow t, \sigma'$ for program e and initial state σ. (The semantics as specified in Figure 3 is nondeterministic, e.g. with respect to received terms, so there may be several derivations.) The concretization mapping γ from elements in V to sets of (labeled) terms should be obvious. The only difficult part is the nonstandard handling of lists: $\gamma(\textit{cons}\ l\ x\ y)$ does not include $\{t_1 :^l t_2 \mid t_1 \in \gamma(x), t_2 \in \gamma(y)\}$ as would be expected, but rather $\{t :^l c \mid t \in \gamma(x), c \in \textit{Const}\} \cup \gamma(y)$. However, as observed earlier, $x \sqsubseteq \textit{head}(\textit{cons}\ l\ x\ y)$ and $y \sqsubseteq \textit{tail}(\textit{cons}\ l\ x\ y)$, and furthermore, if $s = \textit{labels}(\textit{cons}\ l\ x\ y)$ then $\gamma((s, \top))$ is a safe approximation of $\textit{cons}\ l\ x\ y$. Hence, as long as we interpret the values v of a solution in terms of $\textit{labels}(v)$, our analyses are safe.

It is well known that control flow analysis has cubic worst-case time complexity (see e.g. [Heintze and McAllester 1997]). Since our analyses are based on the standard 0-cfa, and the terms of our domain have a fixed maximum depth imposed by the above limiting, we get the same cubic time worst-case complexity. Our experience, however, is that the analysis is in practice quite fast; see also Section 6.5.

Having established our general framework, we now show in the following two sections how it can be instantiated to obtain an escape analysis and a message analysis, respectively.

4.4 Escape analysis

As mentioned, in a scheme where data is allocated on the shared heap by default, the analysis would need to determine what heap-allocated data cannot escape the creating process, or reversely, what data could possibly escape. Following [Shivers 1988], this can be done in the above framework by letting *Escaped* represent the set of all escaping values, and adding the following straightforward rules to the system:

(1) $\textit{In}(\textit{xcall}) \sqsubseteq \textit{Escaped}$

(2) $\mathcal{V}_v[\![v_2]\!] \sqsubseteq \textit{Escaped}$ for all call sites $(v_1\ v_2)^l$ such that $\textit{xlambda} \in \textit{calls}(l)$

(3) $\mathcal{V}_v[\![v_2]\!] \sqsubseteq \textit{Escaped}$ for all send operators $v_1\,!\ v_2$

(4) $\mathcal{V}_v[\![v_1]\!] \sqsubseteq \textit{Escaped}$ and $\mathcal{V}_v[\![v_2]\!] \sqsubseteq \textit{Escaped}$ for every `spawn` $(v_1\ v_2)$

After the fixpoint iteration converges, if the label of a data constructor operation (including lambda expressions) in the program is not in $\textit{labels}(\textit{Escaped})$, the value produced by that operation does not escape the process. (A more common formulation of escape analysis is to discover data that does not escape a particular function invocation and might therefore be stack allocated. For the purposes of this article, however, we are only concerned with whether or not process-local storage can be used for the data.)

4.5 Message analysis

Since we have chosen to allocate data on the local heap by default, we instead want the analysis to tell us which constructors may become part of some message. Furthermore, we need to be able to see whether or not a value could contain locally allocated data constructed at a point outside the analyzed program.

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

For the message analysis, we need the following rules:

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

(2) $\forall l \in \textit{calls}(xcall) : (\{\textit{unknown}\}, \top) \sqsubseteq \textit{Val}(x)$, when l is the label of $(\lambda x.e)^l$

(3) $\mathcal{V}_v[\![v_2]\!] \sqsubseteq \textit{Message}$ for every $v_1\,!\;v_2$ in the program

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

The main difference from the above escape analysis, apart from also tracking unknown inputs, is that we do not care about values that escape the current process unless through explicit message passing. (The closure and argument used in a `spawn` can be viewed as being "sent" to the new process.) If a value escapes our scrutiny by being passed to some external function, we generally underapproximate by assuming that it will *not* be used as message. However, the rest of the analysis always overapproximates the set of possible message constructors. Due to the way the analysis information will be used (see Section 5.2) this is not a problem.

Upon reaching a fixpoint, if the label of a data constructor is not in *labels*(*Message*), the value constructed at that point is not likely to be part of any message. Furthermore, for each argument v_i to any constructor[5], if $\textit{unknown} \notin \textit{labels}(\mathcal{V}_v[\![v_i]\!])$, the value of that argument 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 is therefore not *unknown*.

5. USING THE ANALYSIS INFORMATION

Depending on the selected strategy for allocation and message passing (cf. Section 3.1), the information gathered by the corresponding program analysis (as described above) 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 guaranteed *not* to be part of any message, is rewritten so that the allocation will be performed on the local heap. All other data is allocated on the shared heap. No other modifications are needed. Note that with this scheme, unless escape 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:

[5]This also includes the free variables of any lambda expression, if closure conversion [Appel 1992] is done.

(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 performed on the shared heap. All other data is allocated on the local heap.

(2) For all arguments of the rewritten constructors, and for the message argument of each send-operation, if the value is not guaranteed to already be allocated on the shared heap (i.e., if it might consist of constructors other than those rewritten in the previous step – such as *unknown*), 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 overapproximation, some constructors are allocated in the shared area although they will not in fact be part of any message at run-time. As a consequence, if an argument to such a misplaced constructor was created on the local heap (e.g., by a function in some other module), that argument will need to be copied to the shared area in order to preserve the pointer directionality, but the work is wasted, because the constructed term is not in fact shared. (In comparison, the process-centric system will only copy exactly the data being passed in messages, but it can never avoid copying like the hybrid system, and it will repeat the copying if the data is further forwarded.) If such redundant copying becomes a problem in practice (see Section 6.2 for an example), probabilistic methods or profiling data could likely be used to improve the precision of the analysis.

5.3 Differences between escape analysis and message analysis: an example

Figure 4 shows an Erlang program using two processes. (The line numbers are not part of the program.) 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 Erlang code looks rather similar. Translation to the language of this article 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`. In the context of separate compilation of modules, and taking into account the Erlang requirement that the code of any individual module can be replaced at any time, functions residing in other modules, as in the calls to `mod:test(X)` and `io:fwrite(...)`, are conservatively treated as unknown program parameters by the analyses.

For this example, the 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 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

```
 1   -module(example).
 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.
```

Fig. 4. Erlang program example

pointer directionality invariant. The lambda body in line 8 then becomes

$$\{^{L}\texttt{X},\ \{^{S}\mathit{copy}(\texttt{Y}),\ \mathit{copy}(\texttt{Z})\}\,\}$$

where the outer tuple is locally allocated. (Note that the *copy* wrappers will not copy data that already resides on the shared heap; cf. Section 3.2.)

6. PERFORMANCE EVALUATION

The default runtime system architecture of Erlang/OTP R9 (Release 9)[6] is the process-centric one. The communal ("shared heap") architecture can be selected by specifying the `-shared` flag when the system is started. Based on R9, we have also implemented the modifi cations needed for the hybrid architecture using the local-by-default allocation strategy, and included the above message analysis and transformation as a fi nal stage on the Core Erlang representation in the Erlang/OTP compiler. By default, the compiler generates byte code, from which native code can also be generated.[7] A compiler option invokes the message analysis. We expect that the hybrid architecture will be included as an option

[6]Available as open source from `www.erlang.org` and commercially from `www.erlang.com`.
[7]Currently supported architectures are SPARC, x86, AMD-64 and PowerPC.

in Erlang/OTP R10; in fact, this is indeed the case in the pre-release of R10 where the architecture can be selected by specifying the `-hybrid` command line option at system start-up time.

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.

6.1 The benchmarks

The performance evaluation was based on the following benchmarks:

msort	A distributed implementation of merge sort. Each process receives a list, splits it into two sublists, and spawns two new processes for sorting the new lists. Since new lists are continously created, there is very little forwarding of data.
msort_q	An alternative implementation of distributed merge sort. Sublist splitting is not done by creating new lists, but by indexing into the original list. However, if message data cannot be shared (as in the process-centric system), the algorithm uses quadratic space.
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 fi nal data structure is sent to the parent process.
nag	A synthetic benchmark which creates a ring of 1000 processes. Each process creates a large message (about 200 words) which will be passed on 200 steps in the ring. **nag** is designed to test the behavior of the memory architectures under different program characteristics. It comes in two flavours: **same** and **keep**. The **same** variant creates one *single* message which is then continously forwarded. The **keep** variant creates a new message at every step, but each process keeps received messages live by storing them in a list.
eddie	A medium-sized application (≈2,500 lines of code in 8 modules) implementing an HTTP parser which handles http-get requests from a client.
eddie_m	The **eddie** modules merged into one single module.
mnesia	The standard TPC-B database benchmark for the Mnesia distributed database system [Mattsson et al. 1999]. Mnesia consists of about 22,000 lines of Erlang code, in 29 modules. The benchmark tries to complete as many transactions as possible in a given time quantum. Unlike the other programs, the performance measure is not the runtime, but the average throughput per second.

6.2 Effectiveness of the message analysis

Table I shows amounts of messages sent and words copied between the process-local heaps and the message area in the hybrid system, both when the message analysis is not used to guide allocation, and when it is. A message is counted as copied if at least some part of it needs to be copied to the shared heap at send time.[8] We also show the amount of words allocated directly on the shared heap when the analysis is enabled.

In the hybrid system, the number of words copied also includes forced copying when allocating message data (cf. Section 5.2), and can thus be nonzero even if no copying happens at send time. Note that in a process-centric system, the number of words copied

[8]When the analysis is not used, the number of messages sent without any copying can be nonzero only if some messages are forwarded exactly as they are (without any wrapper). This rarely happens in non-synthetic programs.

Table I. Effectiveness of message analysis in the hybrid system

	Messages	Messages copied		k words	k words	Words copied	
Benchmark	sent	w/o an.	with an.	prealloc.	sent	w/o an.	with an.
msort	49,149	100.0%	0.0%	1,169	1,202	98.6%	3.33%
msort_q	49,149	66.7%	0.0%	451	60,210	0.8%	0.07%
worker	802	100.0%	0.0%	19,995	19.984	100.0%	0.01%
nag (same)	203,000	100.0%	0.5%	606	40,827	2.0%	0.05%
nag (keep)	203,000	100.0%	0.5%	40,606	40,627	100.0%	0.05%
eddie	40,028	100.0%	0.1%	200	421	81.0%	33.47%
eddie_m	40,028	100.0%	0.1%	260	421	81.0%	19.22%
mnesia	1,061,290	100.0%	25.2%	5,461	11,203	77.8%	33.68%

between process heaps is exactly the number of words sent. Also, in the communal system, all data resides in the same shared area, so no copying is done per se (an analysis that trivially classifi es all data as shared will have the same effect for the hybrid system); however, storing data in shared memory can have higher cost, in a multithreaded implementation.

It is clear from Table I that, especially when large amounts of data are being sent, using message analysis can avoid much of the copying. Even in the real-world programs **eddie** and **mnesia**, the amount of copying is reduced from about 80% to 33% when modules are separately compiled, and when compiling **eddie** as a single module, only 19% of the sent data is copied. We can furthermore see that the message analysis typically causes a signifi cant portion of the message data to be preallocated on the shared heap (58–100%), with only a small amount of overapproximation (computed as the difference in copying without and with analysis, compared to the number of words preallocated), being in the range of 0–9%.

In the **mnesia** benchmark, however, we encountered for the fi rst (and so far only) time a problematic case of overapproximation. The numbers shown in Tables I and II were measured when only 28 out of 29 modules were compiled using the message analysis. When compiling also the fi nal module with the analysis enabled, the number of words copied almost doubled. Studying the code in question and the effects of the analysis, it turned out that while a small fraction of the overapproximation might be avoided by a more precise analysis (e.g. distingushing call contexts), the main problem was due to the style of programming: The rogue module was written so that whenever an error occurs, or when an “info query” is received, large parts of otherwise local data will be passed in a message to the outside world. This meant that many data structures that are normally used only for internal bookkeeping (and are thus constantly being updated), were considered “probable messages” by the analysis, and were therefore being created on the shared heap, sometimes also triggering further copying of subterms. The situation could be compared with how a C programmer unaware of pointer analysis and aliasing may write code that a compiler cannot optimize. In our case, a programmer cannot be completely oblivious of the memory model, and needs to keep local data separated from intended message data in order to get the best performance.

6.3 Memory utilization

Our main goal is however not to put all data in the shared area, but to also use the local heaps as much as possible for data that does not need to be shared, while avoiding the poor space behaviour that the process-centric system can display. Figure 5 shows the total heap usage in the different systems, in terms of the maximum heap sizes at garbage collection

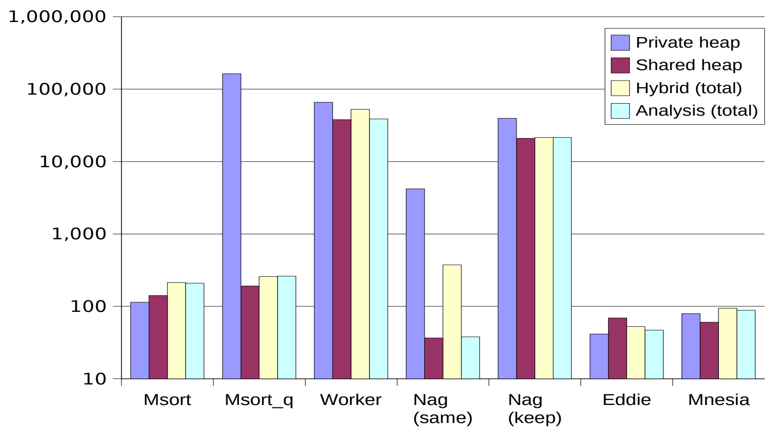


Fig. 5. Total heap usage (k words)

Table II. Heap utilization (k words)

Benchmark	Process-centric	Communal	Plain Hybrid		With Analysis		
			local	shared	local	shared	total
msort	114	142	87	126	87	122	208
msort_q	163,493	192	96	163	96	167	262
worker	69,815	38,041	36,452	16,200	25,033	13,774	38,808
nag (same)	4,207	37	31	343	8	30	38
nag (keep)	39.558	20,924	542	20,924	10	21,473	21,483
eddie	42	69	41	12	39	9	47
mnesia	79	60	32	62	32	56	88

time.[9] (Note that the Y axis is in logarithmic scale.) All heaps used the same initial size for these tests.

The heap utilization numbers can be studied in more detail in Table II. Since the hybrid system *without* message analysis only copies to the shared area the data that actually must be there, on demand, the "plain hybrid" columns give us an approximate lower bound on the utilization of the shared heap. They also show how reuse of message data can reduce the sizes of the local heaps compared to the process-centric system. In general, the table shows that the hybrid system can use significantly less local memory than the process-centric system, and less shared memory than the communal system, although the total amount of used memory can be larger than in the latter; this is a natural consequence of wanting to separate local memory from shared memory. Furthermore, using the message analysis can improve the memory behaviour of the hybrid system, sometimes even reducing the size of the shared area. (Figures showing heap usage of individual benchmarks can be found in Appendix A.)

A seemingly anomalous detail in Table II is that in the **nag (same)** benchmark, the use of the shared area is much higher without the analysis. The reason is that when a message is copied on demand, the original reference is not updated to point to the shared heap. We did not originally think this detail would be worthwhile to handle, but it turns out that in a program like this one, where effectively a multicast is performed which distributes a single

[9]Heap usage of **eddie_m** is not shown as it is identical to that of **eddie**.

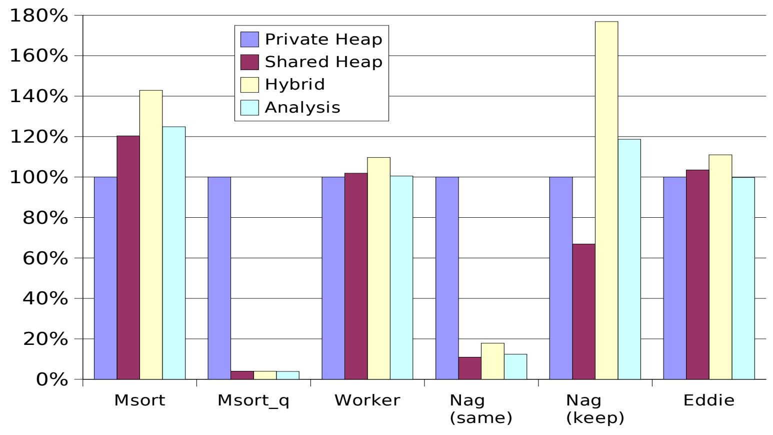


Fig. 6. Normalized runtimes

message to all processes, if the message was first created on the local heap it will then be copied *repeatedly* onto the shared heap - in this case yielding 1000 copies. (Enabling the message analysis eliminates the effect in this particular program, but does not make the hybrid system immune to it.) One way of avoiding such effects might be to cache the reference to the last sent message and its location on the shared heap; a more far-reaching change would be to add indirection pointers to the runtime system (see e.g. [Brooks 1984]).

Note that although we do not show those figures here, the process-centric system can suffer from high fragmentation, consisting of the unused space between the heap top and the stack top on each process – often up to 50% of the allocated memory area. With a large number of processes (e.g., the **msort** benchmark uses more than 16,000 processes), a lot of memory is allocated without being used. Using the hybrid system with message analysis can avoid fragmentation by reducing the local heap sizes. (In comparison, the communal system allows temporary data created by one process to be quickly garbage collected so the memory can be used by another process, keeping the total memory usage low; however, in a multithreaded implementation one does not want processes to share their scratch memory.)

6.4 Runtime performance

Figure 6 shows execution times for the benchmarks,[10] excluding time spent in garbage collection and normalized with respect to the process-centric architecture. Garbage collection times are excluded to avoid possible side-effects from the different garbage collection policies that the different systems employ. (Although for these benchmarks including GC times does not change the overall picture in any significant way; see Appendix B.) The hybrid system, when the analysis is enabled, tends to follow the behaviour of the communal system, avoiding the excessive copying times that the process-centric system sometimes suffers from.

A more detailed breakdown of the execution times for these benchmarks (including time spent in garbage collection) can be found in Appendix B.

[10]Times for **mnesia** are not shown as runtimes do not make sense for this benchmark.

Table III. Compilation times (secs) and percentage of time spent in analysis

	Benchmark sizes			To byte code		To native code	
Benchmark	Modules	Lines	Byte code	Time	Analysis	Time	Analysis
msort	1	76	2,216	0.2	5%	0.8	1%
worker	1	96	2,624	0.2	9%	1.0	2%
nag	1	157	3,596	0.2	9%	1.1	2%
eddie	8	3,310	63,224	1.2	26%	17.7	1.9%
eddie_m	1	3,417	55,152	1.5	33%	18.3	2.7%
mnesia	29	24,216	427,136	9.9	34%	163.0	2%
pseudoknot	1	3,315	72,696	2.4	67%	7.2	22%
inline	1	2,762	37,400	1.3	44%	11.2	4.9%

6.5 Compilation overhead due to the analysis

Table III shows sizes and compilation times for the benchmarks, both for compilation to byte code and to native code. Erlang modules are separately compiled, and most source code files are small (less than 1,000 lines). The numbers for **eddie** and **mnesia** show the total code and byte code size and compilation time for all their modules. We have also included the non-concurrent programs **pseudoknot** and **inline** to show the overhead of the analysis on the compilation of single-module applications which contain functions of quite large size.

In the byte code compiler, the analysis takes on average 25% of the compilation time, with a minimum of 5%. 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 [Johansson et al. 2000; Pettersson et al. 2002], its portion of the compilation time is below 5% in all benchmarks except **pseudoknot** (22%). More importantly, the analysis appears to scale well when performed on the whole program (**eddie_m**) rather than on a single module at a time (**eddie**).

7. RELATED WORK

We first discuss related work in the areas of runtime system organization and static analysis and then we try to hopefully shed some more insight on message analysis.

Runtime system organization. Our hybrid memory model is inspired in part by a runtime system architecture described by Doligez and Leroy [1993] 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 [Back et al. 2000], an operating system providing isolation, resource management, and sharing for the execution of Java programs. An approach using escape analysis to guide a memory management system with thread-specific heaps for Java programs was described by Steensgaard [2000].

Static analysis. As mentioned, our analysis framework can be best understood as an extension of Shivers' control flow analysis [1988] and it is closely related to the frameworks used by escape analyses. Escape analysis was introduced in 1992 by Park and Goldberg, and further refined by Deutsch [1997] and Blanchet [1998]. Till quite recently, its main application has been to permit stack allocation of data in functional languages. In 1999, 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; see the recent journal article [Blanchet 2003]. Concurrently with Blanchet's work, Bogda and Hölzle [1999] 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. used a reachability graph based escape analysis for the same purposes; see [Choi et al. 2003]. Ruf [2000] 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 [Choi et al. 2003], 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.

Message analysis vs. escape analysis. Although our message analysis is in some respects similar to escape analysis, note that it addresses the problem in its reverse direction. Rather than proving that a piece of data does not escape its context (which more often than not requires a whole-program analysis), it identifies data that will probably be used in a message, enabling a speculative optimization that allocates that data in the shared area of the hybrid system, eliminating the need for copying at send time and making it possible to remove some run-time checks altogether. While in our case it is the copying semantics of the Erlang language that allows us to use the message analysis to guide the memory allocator, we think that even languages with sharing semantics could benefit from such a memory architecture when the immutability of data structures can be established, e.g. by static analysis or a type system.

Message analysis vs. region inference. Notice that it is also possible to view the hybrid runtime system architecture as a system with a shared heap and separate *regions* for each process. Region-based memory management, introduced by Tofte and Talpin [1997], typically allocates objects in separate areas according to their lifetimes. The compiler, guided by a static analysis called *region inference*, is responsible for generating code that creates and 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 region inference algorithm for discovering data which potentially outlives its creating process.

8. CONCLUDING REMARKS

For the purpose of employing a hybrid runtime system architecture, which is tailored to the intended use of data in a high-level concurrent language using message passing, we have devised and formalized an effective and practical static analysis, called *message analysis*, that can be used to guide the allocation of data.

As shown in our performance evaluation, the analysis is in practice fast, precise enough to discover most of the data which will become part of some message, and allows the resulting system to combine the best performance characteristics of both a process-centric and a communal memory architecture.

Communication through message passing with copying semantics, even when the communicating processes or threads have access to shared memory (as on a single machine or in a cluster), has many advantages over the currently more common shared-datastructure approach; these include isolation, portability, scalability, and reduced complexity for the programmer. With the formidable explosion of network programming in recent years, many different but similar techniques based on message passing have become buzzwords, such as RMI, SOAP, and XML-RPC. We believe that message passing – regardless of acronym – is here to stay,[11] and that programming environments and languages with direct support for message passing will ultimately be common. Runtime systems will need to be adapted to this way of programming.

ACKNOWLEDGMENT

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.

REFERENCES

APPEL, A. W. 1992. *Compiling with Continuations*. Cambridge University Press, Cambridge, England.

ARMSTRONG, J., VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. 1996. *Concurrent Programming in Erlang*, Second ed. Prentice Hall Europe, Herfordshire, Great Britain.

BACK, G., HSIEH, W. C., AND LEPREAU, J. 2000. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 333–346. http://www.cs.utah.edu/flux/papers/.

BLANCHET, B. 1998. 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)*. ACM Press, New York, N.Y., 25–37.

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

BOGDA, J. AND HÖLZLE, U. 1999. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*. ACM Press, New York, N.Y., 35–46.

BROOKS, R. A. 1984. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. ACM Press, New York, N.Y., 256–262.

CARLSSON, R. 2001. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*.

CARLSSON, R., GUSTAVSSON, B., JOHANSSON, E., LINDGREN, T., NYSTRÖM, S.-O., PETTERSSON, M., AND VIRDING, R. 2000. Core Erlang 1.0 language specification. Tech. Rep. 030, Information Technology Department, Uppsala University. Nov.

CHENG, P. AND BLELLOCH, G. E. 2001. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, N.Y., 125–136.

CHOI, J.-D., GUPTA, M., SERRANO, M., SHREEDHAR, V. C., AND MIDKIFF, S. P. 2003. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Trans. Program. Lang. Syst. 25,* 6 (Nov.), 876–910.

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

DOLIGEZ, D. AND LEROY, X. 1993. 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*. ACM Press, New York, N.Y., 113–123.

DOMANI, T., GOLDSHTEIN, G., KOLODNER, E., LEWIS, E., PETRANK, E., AND SHEINWALD, D. 2002. Thread-local heaps for Java. In *Proceedings of ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, D. Detlefs, Ed. ACM Press, New York, N.Y., 76–87.

[11] Some might say "back with a vengeance".

FLANAGAN, C., SABRY, A., DUBA, B. F., AND FELLEISEN, M. 1993. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, N.Y, 237–247.

HEINTZE, N. AND MCALLESTER, D. A. 1997. On the cubic bottleneck in subtyping and flow analysis. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 342–351.

JOHANSSON, E., PETTERSSON, M., AND SAGONAS, K. 2000. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. ACM Press, New York, NY, 32–43.

JOHANSSON, E., SAGONAS, K., AND WILHELMSSON, J. 2002. Heap architectures for concurrent languages using message passing. In *Proceedings of ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, D. Detlefs, Ed. ACM Press, New York, N.Y., 88–99.

JONES, R. E. AND LINS, R. 1996. *Garbage Collection: Algorithms for automatic memory management*. John Wiley & Sons.

MATTSSON, H., NILSSON, H., AND WIKSTRÖM, C. 1999. Mnesia - a distributed robust DBMS for telecommunications applications. In *Practical Applications of Declarative Languages: Proceedings of the PADL'1999 Symposium*, G. Gupta, Ed. Number 1551 in LNCS. Springer, Berlin, Germany, 152–163.

MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts.

PARK, Y. G. AND GOLDBERG, B. 1992. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, N.Y., 116–127.

PETTERSSON, M., SAGONAS, K., AND JOHANSSON, E. 2002. The HiPE/x86 Erlang compiler: System description and performance evaluation. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, Z. Hu and M. Rodríguez-Artalejo, Eds. Number 2441 in LNCS. Springer, Berlin, Germany, 228–244.

RUF, E. 2000. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, N.Y., 208–218.

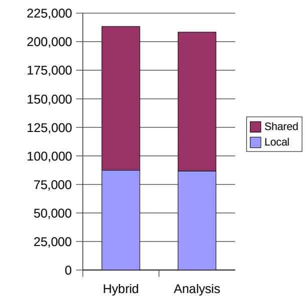
SERRANO, M. AND FEELEY, M. 1996. Storage use analysis and its applications. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*. ACM Press, New York, N.Y., 50–61.

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

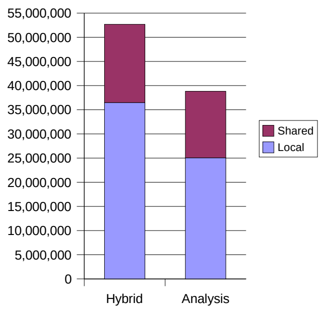
STEENSGAARD, B. 2000. Thread-specific heaps for multi-threaded programs. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*. ACM Press, New York, N.Y., 18–24.

TOFTE, M. AND TALPIN, J.-P. 1997. Region-based memory management. *Information and Computation 132,* 2 (Feb.), 109–176.

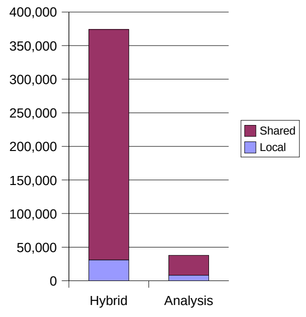
WILSON, P. R. 1992. Uniprocessor garbage collection techniques. In *Proceedings of IWMM'92: International Workshop on Memory Management*, Y. Bekkers and J. Cohen, Eds. Number 637 in LNCS. Springer-Verlag, Berlin, Germany, 1–42. See also expanded version as Univ. of Texas Austin technical report submitted to ACM Computing Surveys.



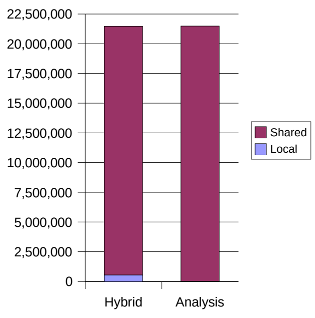
(a) **msort**



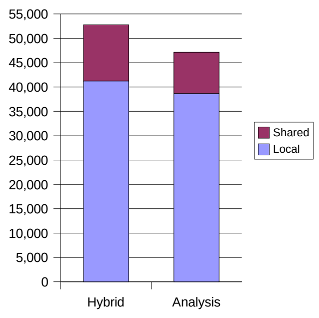
(b) **worker**



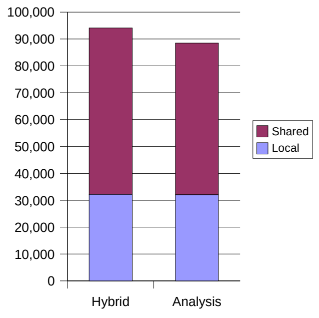
(c) **nag (same)**



(d) **nag (keep)**



(e) **eddie**



(f) **mnesia**

Fig. 7. Heap size improvement due to message analysis

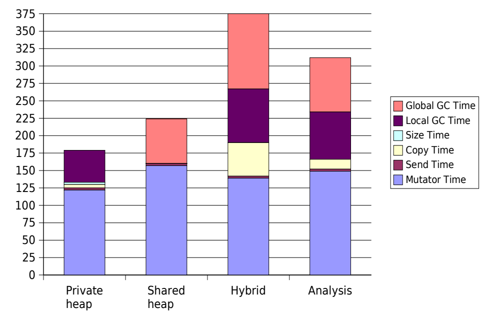
A. HEAP MEMORY USAGE

Figure 7 shows the effect of the message analysis-guided heap allocation on the sizes of the heaps in the hybrid system. While the changes in the allocation pattern is in some cases not big enough to affect the heap enlargement policy by much, as in **msort**, and is rarely as extreme as in **nag (same)**, the message analysis typically makes the hybrid system almost as memory-effi cient as the communal system (cf. Figure 5). We have left out **msort_q**, since it behaves very much like **msort** in this respect.

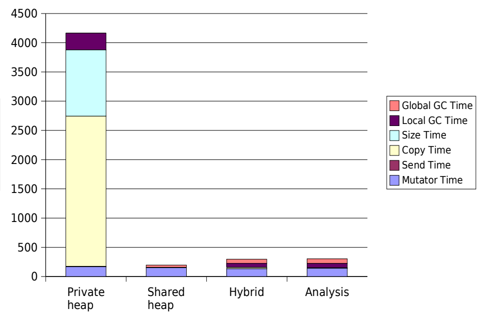
B. EXECUTION TIMES

Figure 8 shows execution time details for the benchmarks (see also Figure 6). It must be noted that the current (copying, 2-generational) garbage collector is tailored for the process-centric system only, and for instance does not work well with large amounts of live data. Work on better garbage collection for the hybrid system is under way, but is not expected to be ready any time soon.

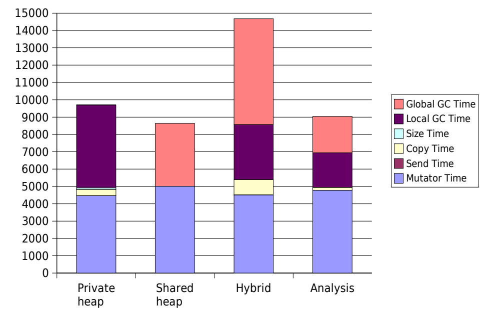
Figure 9 makes it easier to see for each benchmark where the time is spent, in the different systems. It is e.g. clear that the message analysis removes much of the copying overhead in the hybrid system,



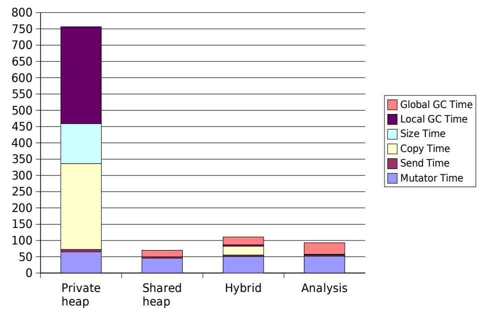
(a) **msort**



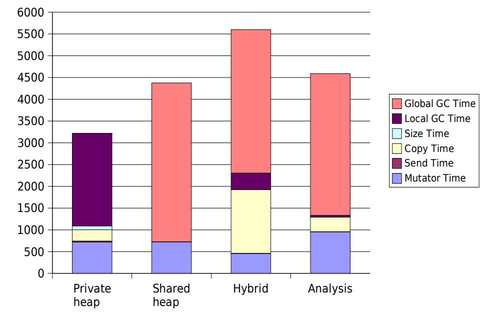
(b) **msort_q**



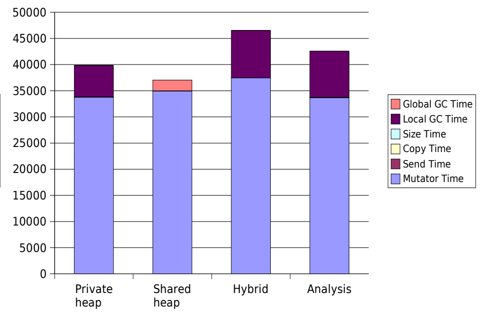
(c) **worker**



(d) **nag (same)**

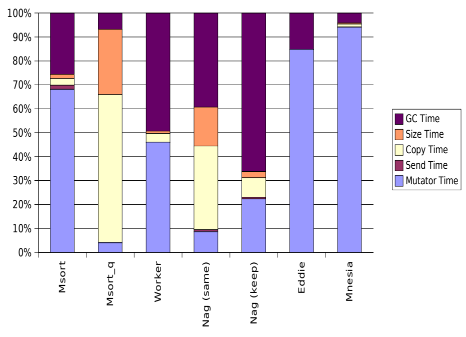


(e) **nag (keep)**

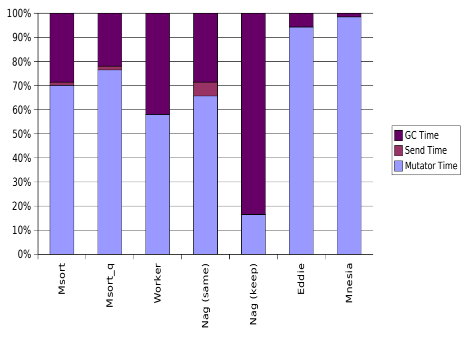


(f) **eddie**

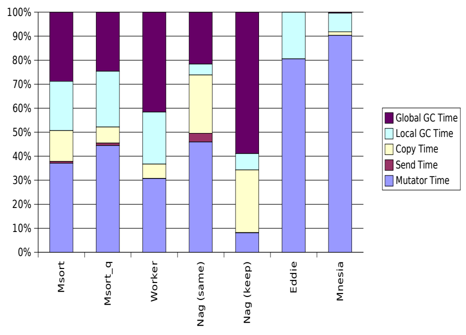
Fig. 8. Performance of individual benchmarks



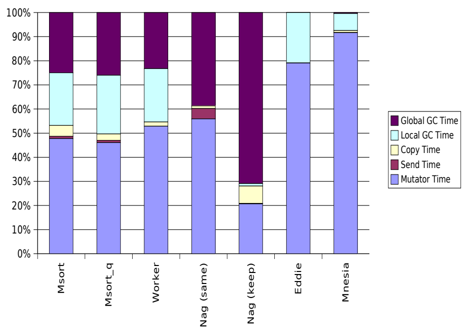
(a) Process-centric



(b) Communal



(c) Hybrid without analysis



(d) Hybrid with message analysis

Fig. 9. Execution time percentages for the different systems