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 the 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 since this 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 a production-quality 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: F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; 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, Measurement, Performance

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

This research was supported in part by Grant No. 621-2003-3442 from the Swedish Research Council (Vetenskapsrådet) and by the Vinnova ASTEC (Advanced Software Technology) Competence Center with matching funds by Ericsson and T-Mobile. This is a significantly extended version of an article that appeared in *SAS'03*: *Proceedings of the Tenth Static Analysis Symposium.*

Authors' addresses: R. Carlsson (corresponding author), K. Sagonas, J. Wilhelmsson, Department of Information Technology, Uppsala University, Box 337, 75105 Uppsala, Sweden; email: {richardc,kostis,jesperw}@it.uu.se.

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 through synchronized shared structures, as, for example, in Java, C#, and Modula-3; using synchronous message passing on (usually, typed) channels as, for example, 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 implemented using a garbage collector. So far, research in memory management has largely focused on the 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, Jones and Lins [1996] or Wilson [1992] for excellent surveys of algorithms and techniques used in 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, untill recently, there has not been any in-depth investigation of the performance tradeoffs involved 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 system 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 lightweight 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 runtime system architecture that has process-specific areas for the allocation of local data and a common area for data that is shared between communicating processes (i.e., is part of some message). Both variants of this *hybrid* architecture allow interprocess communication to occur without actual copying when shared memory is available, use less overall space due to avoiding data replication, and allow frequent process-local heap collections to take place without a need for global synchronization

of processes, thus 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*, the aim of which is to discover what data will be used in messages so as to 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, of course, this is also possible). We present a mini-Erlang language with a formal semantics and sketch proofs of correctness and safety 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. Our evaluation shows that although the analysis has cubic worst-case time complexity, it tends to be fast enough in practice.

Finally, we have implemented the aforementioned architecture and analysis in the context of an industrial-strength implementation used for highly concurrent time-critical applications, and we 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 its power in terms of 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 thus far been used quite successfully by Ericsson and other companies around the world to construct large (several hundred thousand lines of code) commercial applications.

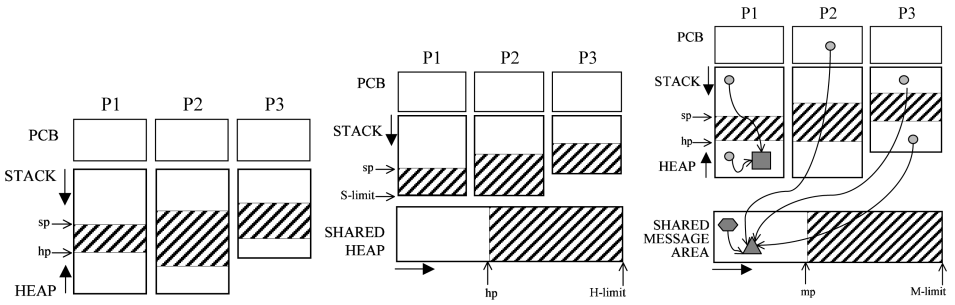
Erlang's basic data types are atoms (symbols), numbers (floats and arbitrary precision integers), and process identifiers; compound data types are lists and tuples. Programs consist of function definitions organized in *modules*. There is no destructive assignment of variables or data. Because recursion is the only means to express iteration in Erlang, tail call optimization is a required feature of Erlang implementations.

Processes in Erlang are extremely lightweight (lighter than operating system 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 not migrate to the new code until they execute an intermodule call to the module (possibly first having returned from the old code).[1] For example, a typical server process will be executing an event loop which, after handling a single event, will make a self-recursive tail call qualified with the module name, ensuring a switch whenever new code is loaded. While dynamic code replacement is considered an essential feature for real-world Erlang applications, it constitutes a major obstacle for whole-program and crossmodule 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.

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



(a) processcentric (b) communal (c) hybrid

Fig. 1. Different runtime system architectures for concurrent languages.

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 *communal*, where all processes get to share the same heap; and finally, one a *hybrid* runtime system architecture, where each process has a private heap for local data while a shared heap is used for data sent as messages. Figure 1 depicts the memory areas of these architectures when three processes are currently in the system: Shaded areas show currently unused memory; each process has exactly one process control block (PCB); and the filled shapes and arrows in Figure 1(c) represent messages and pointers.

For each architecture, we have already discussed its advantages and disadvantages, focusing on the impact of the architecture on the speed of interprocess communication and garbage collection (GC). We briefly review them now:

Process-centric. This is currently the default configuration of Erlang/OTP. Interprocess communication requires copying of messages, that is, it is an $O(n)$ operation where n is the message size. Memory fragmentation tends to be high. Advantages 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 latter 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), reduced memory requirements due to message sharing, and low fragmentation. Disadvantages include having to consider the stacks of all processes as root set (resulting in increased GC latency), and possibly, poor cache performance due to the processes' data being interleaved on the shared heap. Furthermore, 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, for example, Cheng and Blelloch [2001] for a recent treatment of the subject.

Hybrid. An architecture that tries to combine the advantages of the previous 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, that is, used as messages. This can be achieved through 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 property of the hybrid architecture is the ability to garbage collect process-local heaps individually without looking at the shared heap. In a multithreaded system, this allows the collection of local heaps without any locking or synchronization. If pointers from the shared area 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 read or write barrier, which typically incurs a nonnegligible 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 area to another; see Figure 1(c).

This pointer-directionality invariant 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 performing a simple $O(1)$ pointer comparison (there are several possible implementations which allow this test to be done in constant time; in ours, the message area is kept as a single contiguous block of memory, making the test very cheap).

3.1 Allocation Strategies

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

Local Allocation of Nonmessages. 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 generally possible 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. 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 having escaped. Thus, this strategy is less likely to make good use of both the local heaps and the shared heap.

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 the message is already on the shared heap, the test can be statically eliminated. Without such analysis, the behaviour will be similar to that of 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; we could even introduce unnecessary copying as further explained in Section 5.

We have chosen to implement and evaluate the second strategy, which, to the best of our knowledge, has not been studied previously.

Two other allocation strategies are also possible, both requiring that messages can be copied on demand. The first is to allocate data shared by default, but to allocate probable nonmessage data on the process-local heaps. However, if there are no indications at all as to whether a piece of data will become part of a message or not, it generally seems better to assume that it is local, as in our selected strategy, rather than shared since in most programs the majority of data is used exclusively by the process that creates it. The second variation is to allocate data on the shared heap only if it can be proved that it will always be used in a message, and on the local heap otherwise. This is more conservative than our selected strategy and as such, likely to miss opportunities to avoid copying. Whether the performance differences between this strategy and our selected one are significant in practice is not clear, and ideally, we would want to experimentally measure the performance tradeoffs of all these allocation alternatives. As doing so is not at all trivial, we have opted instead to implement the strategy that our intuition tells us is likely to perform best, given the constraints imposed by our setting (e.g., the requirement for separate compilation of modules).

3.2 Copying of Messages

In order to use a strategy that allocates on the local heap by default, 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 copying, we only know the size of the topmost constructor of the message. We allocate space in the message area for this constructor. Nonpointer 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 (it should be noted that our current implementation of copying was written mainly for speed and does not preserve sharing in the original message nor redirect pointers from the local heap to the shared heap).

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 copying because garbage collection cannot occur in midcopy. 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 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 guaranteed to not be included in any message, so that it could be allocated on the local heap. This amounts to *escape analysis* of process-local data; see, for example, Blanchet [2003], Bogda and Hölzle [1999], and 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 that it can be directly

$$
\begin{array}{rcll}
c & \in & \mathit{Const} & \mathit{Integer} \cup \mathit{Atom} \cup \mathit{Pid} \cup \{\mathit{nil}\} \\
p & \in & \mathit{Pid} & \text{Process identifiers} \\
x & \in & \mathit{Var} & \text{Variables} \\
e & \in & \mathit{Expr} & \text{Expressions} \\
v & \in & \mathit{SimpleExpr} & \text{Simple-value expressions} \\
b & \in & \mathit{BoundExpr} & \text{Bound-value expressions} \\
f & \in & \mathit{LambdaExpr} & \text{Lambda expressions} \\
o & \in & \mathit{Primops} & \text{Primitive operations (}\texttt{==}, \texttt{>}, \texttt{is\_nil}, \texttt{is\_cons}, \texttt{is\_tuple}, \ldots\text{)} \\
\beta & \in & \mathit{Label}_B & \text{Labels of call sites, including } \textit{xcall} \\
\lambda & \in & \mathit{Label}_\Lambda & \text{Labels of lambda expressions, including } \textit{xlambda} \\
\delta & \in & \mathit{Label}_\Delta & \text{Labels of data constructors, including } \textit{message} \text{ and } \textit{unsafe} \\
\eta & \in & \mathit{Label}_H & = \mathit{Label}_\Delta \cup \mathit{Label}_\Lambda \\
\sigma & \in & \mathit{State} & = \mathcal{P}(\mathit{Term}) \\
\rho & \in & \mathit{Env} & = \mathit{Var} \rightarrow \mathit{Term} \\
t & \in & \mathit{Term} & ::= \mathit{Const} \cup \{t_1 :^\delta t_2\} \cup \{\{t_1, \ldots, t_n\}^\delta \mid n \geq 0\} \cup \mathit{Clos} \\
 & & \mathit{Clos} & ::= \{\langle f, \rho, \rho' \rangle \mid f \in \mathit{LambdaExpr},\ \rho, \rho' \in \mathit{Env}\}
\end{array}
$$

$$
\begin{array}{rcl}
v & ::= & c \mid x \\
e & ::= & v \mid (v_1\ v_2)^\beta \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{let}\ x = b\ \texttt{in}\ e \mid \\
 & & \texttt{letrec}\ x_1 = f_1, \ldots, x_n = f_n\ \texttt{in}\ e \\
b & ::= & v \mid f \mid (v_1\ v_2)^\beta \mid v_1 :^\delta v_2 \mid \{v_1, \ldots, v_n\}^\delta \mid \texttt{hd}\ v \mid \texttt{tl}\ v \mid \texttt{element}_k\ v \mid \\
 & & v_1\ \texttt{!}\ v_2 \mid \texttt{receive} \mid \texttt{spawn}\ (v_1\ v_2)^\beta \mid \texttt{primop}\ o(v_1, \ldots, v_n) \\
f & ::= & (\texttt{fn}\ x.e)^\lambda
\end{array}
$$

Fig. 2. A mini-Erlang language.

allocated in the shared area in order to avoid the copying operation when the message is eventually sent. 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 (see Section 5), both under- and overapproximation of the set of runtime 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 the unnecessary use of 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 operators for asynchronous send (‘!’), blocking receive, and process spawning.

A program is any expression e that does not contain free variables. It is assumed that all variables in the program are uniquely named, that is, no variable name appears in more than one `let` or `letrec`. We also make the simplifying assumption that all primitive operations return atomic values and do not cause their parameters to escape the process; 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 `[]`. Tuple constructors are written $\{v_1, \ldots, v_n\}$ for all $n \geq 0$, and list constructors are written $v_1 : v_2$. Each constructor in the program, as well as each call site and lambda expression, is given a unique label; the special labels *xcall*, *xlambda*, *message*, and *unsafe* are used to represent program points external to the currently analyzed program. The *xcall* and *xlambda* labels are standard for control flow analysis [Shivers 1988], while *message* and *unsafe* are particular to our message analysis; their use is explained in the following. We generally leave out the labels when they can be deduced from context.

Operators `hd` and `tl` select the first (head) and second (tail) element, respectively, of a list constructor. 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 the vast majority of lists are proper. 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 the 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 slightly nonstandard big-step operational semantics for the analyzed language is shown in Figure 3, in which constructed data is labeled with its point of origin. Labels have no effect upon operations on data, such as equivalence comparisons (a standard semantics is achieved by disregarding the labels altogether). Note that for specifying the complete Erlang language (including concurrency), big-step semantics would not be suitable, but it is sufficient for this article and has several advantages, such as being concise and readable. On the other hand, big-step semantics does not let us reason formally about nonterminating programs.

For our purposes, it is only necessary to describe the behaviour of a single process at a time; therefore, the state σ is simply the set of shared terms (for simplicity, σ has been left out where it is not affected). We let $T{\downarrow_s}$ denote the subset T of *Term* restricted to constructors whose labels are in s. Sending a message adds the sent term to the shared area, and receiving a message yields

[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 runtime copying checks that can be statically eliminated.

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

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

$$\frac{\rho \vdash v_1 \to t_1 \quad \rho \vdash v_2 \to t_2}{\rho \vdash v_1 :^{\delta} v_2 \to t_1 :^{\delta} t_2} \qquad \text{[Cons]}$$

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

$$\frac{\rho \vdash v \to t_1 :^{\delta} t_2}{\rho \vdash \texttt{hd}\ v \to t_1} \qquad \text{[Head]}$$

$$\frac{\rho \vdash v \to t_1 :^{\delta} t_2}{\rho \vdash \texttt{tl}\ v \to t_2} \qquad \text{[Tail]}$$

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

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

$$\frac{\rho + rec[x_1 \mapsto \langle f_1, \rho, [] \rangle, \ldots, x_n \mapsto \langle f_n, \rho, [] \rangle] \vdash e, \sigma \to t, \sigma'}{\rho \vdash \texttt{letrec}\ x_1 = f_1, \ldots, x_n = f_n\ \texttt{in}\ e, \sigma \to t, \sigma'} \qquad \text{[Letrec]}$$

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

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

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

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

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

$$\frac{\rho \vdash v_1 \to t_1 \quad \rho \vdash v_2 \to t_2 \quad p \in \mathit{Pid}}{\rho \vdash \texttt{spawn}\ (v_1\ v_2), \sigma \to p, \sigma \cup \{t_1, t_2\}} \qquad \text{[Spawn]}$$

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

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

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

Fig. 3. A nonstandard operational semantics for the language of Figure. 2.

an arbitrary element in $\mathit{Term}\downarrow_{\{message\}}$ as result;[3] see the [Send] and [Receive] rules. Spawning a new process adds both arguments to the shared area and

[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.

yields a fresh process identifier; see the [Spawn] rule (we leave out all details about how process identifiers are created; they are constants and are not heap-allocated.)

We write $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ for the partial function which maps x_i to y_i for $i \in [1, n]$, and $\rho + \rho'$ for the extension of one mapping by another. 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 $\langle f, \rho, \rho' \rangle$, where f is a lambda 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 [Letrec] and [Call]. The free variables of an expression e are given by the function $free[\![e]\!]$.

4.2 General Framework

The analyses we have thus far implemented are first-order dataflow analyses, and are best understood as extensions of Shivers' 0CFA 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 $calls: Label_B \to \mathcal{P}(Label_\Lambda)$ that maps 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 might be accessed through the value returned by program e or that might be passed as an argument from within e to an external function "escapes" from e and could be called from an external site. 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 call site to a function in e or is received by e in a message.

Note that in the context of program analysis, we allow a *program* to be any arbitrary expression; if a program contains free variables, they may assume any values from *Term*. This is in contrast to the previous section, where programs could only be closed expressions.

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

The analysis domain V is defined as

$$\begin{aligned} v \in V \ ::= \ & (s, \top) \\ | \ & (s, \langle v_1, \ldots, v_n \rangle) \qquad n \geq 0, \end{aligned}$$

$$\text{where } s \in \mathcal{P}(Label_H).$$

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. An abstract node $(S, \top)$ represents the set of all possible subtrees where each node is annotated with set S.

$$
\begin{aligned}
t \in \gamma(v) \;&\text{ if }\; t \in \gamma_0(v) \\
t :^{\delta} t' \in \gamma(v) \;&\text{ if }\; \{t\}^{\delta} \in \gamma_0(v) \wedge t' \in \gamma(v)
\end{aligned}
$$

$$
\begin{aligned}
\gamma_0(s, \top) &= \mathit{Term}\downarrow_s \\
\gamma_0(s, \langle v_1, \ldots, v_n \rangle) &= \mathit{Const} \\
&\quad \cup \{\{t_1, \ldots, t_n\}^{\delta} \mid \delta \in s, \forall i \in [1,n] : t_i \in \gamma(v_i)\} \\
&\quad \cup \{\langle (\texttt{fn}\ x.e)^{\lambda}, [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n], [\ldots] \rangle \mid \lambda \in s, \\
&\qquad \langle x_1, \ldots, x_n \rangle = \mathit{free}[\![(\texttt{fn}\ x.e)^{\lambda}]\!], \forall i \in [1,n] : t_i \in \gamma(v_i)\} \\
&\quad \cup \gamma_0(s, \langle v_1, \ldots, v_{n-1} \rangle)
\end{aligned}
$$

Fig. 4. The concretization mapping $\gamma :: V \to \mathcal{P}(\mathit{Term})$.

Define $\sqsubseteq$ to be the smallest relation on V such that

$$
\begin{aligned}
(s_1, w) &\sqsubseteq (s_2, \top) && \text{if } \mathit{labels}\ (s_1, w) \subseteq s_2 \\
(s_1, \langle u_1, \ldots, u_n \rangle) &\sqsubseteq (s_2, \langle v_1, \ldots, v_m \rangle) && \text{if } n \le m \wedge s_1 \subseteq s_2 \wedge \forall j \in [1,n] : u_j \sqsubseteq v_j\ ,
\end{aligned}
$$

where

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

It is then easy to see that $\langle V, \sqsubseteq \rangle$ is an upper semilattice with top element $(\mathit{Label}_H, \top)$, and bottom element $(\emptyset, \langle\rangle)$.

The concretization function $\gamma :: V \to \mathcal{P}(\mathit{Term})$ is defined as the smallest mapping fulfilling the requirements shown in Figure 4 (typically, given program e and abstract value v, we would also restrict the set $\gamma(v)$ to such terms that are consistent with the labeled constructors and lambda expressions in e). For the sake of readability, we have omitted all details about the recursive components of closures from the definition.[4]

LEMMA. $v_1 \sqsubseteq v_2 \Rightarrow \gamma(v_1) \subseteq \gamma(v_2)$

The proof is easy and is left to the reader.

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; this is further explained next.

We now specify the main equations of the constraint system for the analyses. Let *Val* be a mapping from variables to abstract values, and *In* be a mapping from call site labels to abstract values. Figure 5 gives the definitions of the expression analysis functions $\mathcal{V}_v[\![\cdot]\!]$ and $\mathcal{V}_e[\![\cdot]\!]$, and the bound-value analysis function $\mathcal{V}_b[\![\cdot]\!]$ In particular, note that:

—The rule for `receive` simply returns $\bot$ since we do not need to track values that are guaranteed to reside in the shared area already.

[4]Like the second component of closures, these are also mappings from variable names to terms.

$$
\begin{aligned}
\mathit{Val} &:: \mathit{Var} \to V \\
\mathit{In} &:: \mathit{Label}_B \to V
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}_v[\![c]\!] &= \bot \\
\mathcal{V}_v[\![x]\!] &= \mathit{Val}(x)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}_e[\![v]\!] &= \mathcal{V}_v[\![v]\!] \\
\mathcal{V}_e[\![(v_1\ v_2)^\beta]\!] &= \mathit{In}(\beta) \\
\mathcal{V}_e[\![\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] &= \mathcal{V}_e[\![e_1]\!] \sqcup \mathcal{V}_e[\![e_2]\!] \\
\mathcal{V}_e[\![\texttt{let}\ x = b\ \texttt{in}\ e]\!] &= \mathcal{V}_e[\![e]\!] \\
\mathcal{V}_e[\![\texttt{letrec}\ x_1 = f_1, \ldots, x_n = f_n\ \texttt{in}\ e]\!] &= \mathcal{V}_e[\![e]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}_b[\![v]\!] &= \mathcal{V}_v[\![v]\!] \\
\mathcal{V}_b[\![(v_1\ v_2)^\beta]\!] &= \mathit{In}(\beta) \\
\mathcal{V}_b[\![(\texttt{fn}\ x.e)^\lambda]\!] &= \mathit{clos}\ \lambda\ \langle \mathcal{V}_v[\![x_1]\!], \ldots, \mathcal{V}_v[\![x_n]\!] \rangle \\
&\qquad \text{if } \langle x_1, \ldots, x_n \rangle = \mathit{free}[\![(\texttt{fn}\ x.e)]\!] \\
\mathcal{V}_b[\![v_1 :^\delta v_2]\!] &= \mathit{cons}\ \delta\ \mathcal{V}_v[\![v_1]\!]\ \mathcal{V}_v[\![v_2]\!] \\
\mathcal{V}_b[\![\{v_1, \ldots, v_n\}^\delta]\!] &= \mathit{tuple}\ \delta\ \langle \mathcal{V}_v[\![v_1]\!], \ldots, \mathcal{V}_v[\![v_n]\!] \rangle \\
\mathcal{V}_b[\![\texttt{hd}\ v]\!] &= \mathit{head}(\mathcal{V}_v[\![v]\!]) \\
\mathcal{V}_b[\![\texttt{tl}\ v]\!] &= \mathit{tail}(\mathcal{V}_v[\![v]\!]) \\
\mathcal{V}_b[\![\texttt{element}_k\ v]\!] &= \mathit{elem}\ k\ \mathcal{V}_v[\![v]\!] \\
\mathcal{V}_b[\![v_1\,!\ v_2]\!] &= \mathcal{V}_v[\![v_2]\!] \\
\mathcal{V}_b[\![\texttt{receive}]\!] &= \bot \\
\mathcal{V}_b[\![\texttt{spawn}\ (v_1\ v_2)^\beta]\!] &= \bot \\
\mathcal{V}_b[\![\texttt{primop}\ o(v_1, ..., v_n)]\!] &= \bot
\end{aligned}
$$

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

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

Fig. 5. The main analysis functions.

—A `spawn` has an effect similar to a send operation. The function application is evaluated by the spawned process, and the arguments will reside in the shared heap at the time of the application.

The following constraints complete the general framework:

(1) $\forall\lambda \in \mathit{Label}_\Lambda : \mathit{Out}(\lambda) = \mathcal{V}_e[\![e]\!]$, when λ is the label of $(\texttt{fn}\ x.e)^\lambda$.
(2) $\forall\beta \in \mathit{Label}_B : \forall\lambda \in \mathit{calls}(\beta) : \mathit{Out}(\lambda) \sqsubseteq \mathit{In}(\beta) \wedge \mathcal{V}_v[\![v_2]\!] \sqsubseteq \mathit{Val}(x)$, when β is the label of $(v_1\ v_2)^\beta$ and λ is the label of $(\texttt{fn}\ x.e)^\lambda$.
(3) For each expression $\texttt{let}\ x = b\ \texttt{in}\ e$ in the program: $\mathcal{V}_b[\![b]\!] \sqsubseteq \mathit{Val}(x)$, and for each $\texttt{letrec}\ x_1 = f_1, \ldots, x_n = f_n\ \texttt{in}\ e$: $\forall i \in [1,n] : \mathcal{V}_b[\![f_i]\!] \sqsubseteq \mathit{Val}(x_i)$ (note that this is the only place where $\mathcal{V}_b[\![\cdot]\!]$ is used).

It is easy to verify from the definitions that the constraint system has a least-fixpoint solution due to monotonicity.

To see what happens with lists, suppose $z = \mathcal{V}_b[\![v_1 :^\delta v_2]\!] = \mathit{cons}\ \delta\ x\ y = (\{\delta\}, \langle x \rangle) \sqcup y$. If y is $(s, \langle v, \ldots \rangle)$, then the set of top-level constructor labels of z is $s \cup \{\delta\}$. Furthermore, $\mathit{head}\ z$ will yield $x \sqcup v$, while $\mathit{tail}\ z$ yields z itself (note that $x \sqsubseteq \mathit{head}\ z$ and $y \sqsubseteq \mathit{tail}\ z$). 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; however, we can still separate levels of nesting, as in lists of lists, etc. In the vast majority of Erlang programs, cons cells are used exclusively for constructing proper lists, so we do not consider the loss of precision for nonproper lists to be an issue.

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, for example, Scheme or ML, so we expect that the improvements to the determined call graphs would be insignificant in practice.

4.3 Termination, Complexity, and Correctness of the Framework

Finding the least solution for *Val* and *In* to the aforementioned 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:

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

The constraints given in Section 4.2 are modified, as follows, for some fixed k:

(2) $\forall\beta \in \mathit{Label}_B : \forall\lambda \in \mathit{calls}(\beta) : \theta_k(\mathit{Out}(\lambda)) \sqsubseteq \mathit{In}(\beta) \wedge \theta_k(\mathcal{V}_v[\![v_2]\!]) \sqsubseteq \mathit{Val}(x)$, when β is the label of $(v_1\ v_2)^\beta$ and λ is the label of $(\texttt{fn}\ x.e)^\lambda$.

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,[5] or, if we view the abstract values as *annotated types*, a simple soft typing system. Better precision could be achieved by a graph representation of data, as, for example, in the storage use analysis of Serrano and Feeley [1996]. Our current approximation was chosen to fit easily into our existing analysis framework and to allow us to explore the usefulness of message analysis as a tool for guiding the allocation of data.

It is well known that control flow analysis has cubic worst-case time complexity (see, for example, Heintze and McAllester [1997]). Since our analyses are based on the standard 0CFA, and the terms of our domain have a fixed maximum depth imposed by the previously described 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 6.2.

Finally, we propose that any solution to the constraint system is a safe approximation of all possible executions of the program:

THEOREM 1. *If (Val, In) is a solution to the aforementioned constraint system for program e, where e has no free variables, then for every variable x in e, if* $\rho(x) = t$ *in any judgement in any derivation of* $[\,] \vdash e, \sigma \rightarrow t', \sigma'$, *then* $t \in \gamma(Val(x)) \vee t \in Term\downarrow_{\{message\}}$.

PROOF. By structural induction over the derivations of $[\,] \vdash e, \sigma \rightarrow t, \sigma'$ (the semantics as specified in Figure 3 is nondeterministic, e.g., with respect to received terms, so there may be several derivations). □

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 already mentioned, in a scheme where data is allocated on the shared heap by default, the analysis needs to determine what heap-allocated data cannot escape the creating process, or conversely, what data can possibly escape. Following Shivers [1988], this can be done in the preceeding framework by letting *Escaped* represent the set of all escaping values, and adding the following straightforward rules to the system (mainly shown here for easy comparison with our message analysis):

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

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

After the fixpoint iteration converges, if the label of a data constructor operation (including lambda expressions) in the program is not in $labels(Escaped)$, 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 want the message 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 might contain data which has been constructed on the local heap, but at a point outside the code being analyzed (recall that we typically compile one module at a time).

For this purpose, we let the label *unsafe* 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) $(\{unsafe\}, \top) \sqsubseteq In(\beta)$ for all call sites $(v_1\ v_2)^\beta$ such that $xlambda \in calls(\beta)$;
(2) $\forall \lambda \in calls(xcall) : (\{unsafe\}, \top) \sqsubseteq Val(x)$, when λ is the label of $(\texttt{fn}\ x.e)^\lambda$;
(3) $\mathcal{V}_v[\![v_2]\!] \sqsubseteq Message$ for every $v_1 ! v_2$ in the program; and
(4) $\mathcal{V}_v[\![v_1]\!] \sqsubseteq Message$ and $\mathcal{V}_v[\![v_2]\!] \sqsubseteq Message$ for every `spawn` $(v_1\ v_2)^\beta$ in the program.

Note that since the result of a `receive` is necessarily a message, we know that it already resides in the shared area, and is therefore not *unsafe*.

The main difference from the escape analysis of the previous section, 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 a message. However, the rest of the analysis always overapproximates the set of possible message constructors. Due to the way the message analysis information will be used, this is not a problem.

If we did not underapproximate in the case of external calls, a lot of data would be allocated on the shared heap although in fact never sent as message, much like when the strategy is to allocate only guaranteed nonmessages locally; the majority of functions in typical code do not cause their arguments to be sent as messages. Although we have not done so yet, it would be straightforward to extend the message analysis to a multipass analysis that stores information for every analyzed module about which function arguments may become part of a message. Such information would be particularly useful for certain often-used library functions that do perform message passing. However, note that since both over- and underapproximation is allowed, it is not critical that modules are analyzed in any particular order, or that the information is up-to-date at all times, for example, when loading new code for a module.

4.6 Precision and Soundness of the Message Analysis

If (*Val*, *In*, *Message*) is a solution to the message analysis constraint system for program *e*, then the following properties hold:

THEOREM 2. *If δ is the label of a data constructor in e, and $\delta \notin labels(Message)$, then a term created at δ can only become part of a message if it escapes from e.*

PROOF SKETCH. For any final state σ' such that $\delta \in \sigma'$, assume that no term labeled δ escapes from e and prove a contradiction, that is, that $\delta \notin \sigma'$, by induction over the derivations of $[\,] \vdash e, \sigma \rightarrow t', \sigma'$. □

THEOREM 3. *For each variable x in e, if $unsafe \notin labels(Val(x))$, then if δ is the label of some subterm of a value bound to x in any execution of e, and $\delta \notin labels(Val(x))$, then $\delta = message$.*

PROOF SKETCH. Assume that for some $\delta \notin labels(Val(x))$, δ is the label of a subterm of some t such that $\rho(x) = t$ in some derivation of $[\,] \vdash e, \sigma \rightarrow t', \sigma'$, and $\delta \neq message$. Prove a contradiction, that is, that either $\delta \in labels(Val(x))$ or δ cannot be the label of a subterm of any such t. Note that if some $\delta \neq message$ is not in e, then a value labeled δ can only enter e by being passed from an external call site or by being returned from a call to an external lambda. In both cases, if the value is bound to some x, then *unsafe* will be in $labels(Val(x))$. On the other hand, if $unsafe \notin labels(Val(x))$, then it follows from Theorem 1 that $\delta \in labels(Val(x))$. □

Theorem 2 tells us that it is reasonable to consider only those constructors whose labels are in *labels*(*Message*) to be likely messages, while Theorem 3 allows us to eliminate copying operations as we rewrite allocation points. As we will see in the following section, only the latter is required for safety.

5. USING THE ANALYSIS INFORMATION

Achieving shared allocation of possible messages when allocating on the process-local heaps by default 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 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 completely reside on the shared heap already (i.e., if it could contain data constructed at points other than those rewritten in the previous step—such as the *unsafe* external constructor), the argument is wrapped in a call to `copy` in order to maintain the pointer-directionality invariant.[6]

[6]In terms of the semantics of Figure 3, the copy operation replaces all the labels of any term with *message*. Note that if the top-level label of a term is already *message*, then by the pointer-directionality invariant, all its subterms are labeled *message*.

In effect, with this scheme we attempt to push the runtime 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 runtime. As a consequence, if an argument to such a misplaced constructor has been 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 invariant, 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 resent to another process). If such redundant copying becomes a problem in practice (see Section 6.3 for an example), probabilistic methods or profiling data could likely be used to improve the precision of the analysis.

For comparison, when allocating on the shared heap by default, each data constructor in the program (such that a value constructed at that point is guaranteed by escape analysis *not* to be part of any message) can simply be rewritten so that the allocation will be performed on the local heap. With this scheme, unless the analysis is able to report some constructors as nonescaping, the process-local heaps will not be used at all.

5.1 Differences Between Escape Analysis and Message Analysis: An Example

Figure 6 shows an Erlang program using two processes (the line numbers are not part of the program). The `main` function takes three equal-length lists, reverses each element of the third list, combines the lists into a single list of nested tuples, filters this list using a Boolean function `test` defined in some other module `mod`, and finally, sends the second component of each element in the resulting list to a newly 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`. 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, escape analysis can determine that the lists constructed in the functions `map`, `zipwith3`, and `filter` (lines 14, 22, and 27, respectively) are guaranteed to not escape the executing process, and may be locally allocated. However, since the elements created by the lambda in line 9 are being passed to an unknown function via `filter`, they must be conservatively viewed as escaping, and so must the lists constructed by `reverse`.

On the other hand, the message analysis recognizes that only the innermost of the tuple constructors in line 9, plus the lists constructed by `reverse`, and the closure passed to `spawn` (line 5), can possibly be messages. Thus, creating these

```
1   -module(example).
2   -export([main/3]).
3
4   main(Xs, Ys, Zs) ->
5      P = spawn(fun receiver/0),
6      Zs1 = map(fun (Z) -> reverse(Z, []) end, Zs),
7      foreach(fun (X) -> P ! element(2, X) end,
8              filter(fun (X) -> mod:test(X) end,
9                     zipwith3(fun (X, Y, Z) -> {X, {Y, Z}} end,
10                             Xs, Ys, Zs1))),
11     P ! stop.
12
13  map(F, [X | Xs]) ->
14     [F(X) | map(F, Xs)];
15  map(F, []) -> [].
16
17  reverse([X | Xs], As) ->
18     reverse(Xs, [X | As]);
19  reverse([], As) -> As.
20
21  zipwith3(F, [X | Xs], [Y | Ys], [Z | Zs]) ->
22     [F(X, Y, Z) | zipwith3(F, Xs, Ys, Zs)];
23  zipwith3(F, [], [], []) -> [].
24
25  filter(F, [X | Xs]) ->
26     case F(X) of
27        true -> [X | filter(F, Xs)];
28        false -> filter(F, Xs)
29     end;
30  filter(F, []) -> [].
31
32  foreach(F, [X | Xs]) ->
33     F(X), foreach(F, Xs);
34  foreach(F, []) -> ok.
35
36  receiver() ->
37     receive
38        stop -> ok;
39        {X, Y} -> io:fwrite("~w: ~w.\n", [X, Y]), receiver()
40     end.
```

Fig. 6. Erlang program example.

directly on the shared message area could reduce copying if data is allocated locally by default. However, the argument `Y` is created externally (it is taken from the elements of `Ys`), and might need to be copied to maintain the pointer-directionality invariant. In contrast, the argument `Z` is the result from a call to `reverse` and can be guaranteed to not require further copying if we rewrite the list constructor in line 18 as

$$[^{S}\mathit{copy}(\texttt{X}) \mid \texttt{As}],$$

that is, to push the copying operation past the constructors of the reversed list, which are created on the shared message area. The lambda body in line 9 can

then be rewritten to

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

where the outer tuple is locally allocated (note that the *copy* wrappers will not copy data that already resides on the shared message area; see Section 3.2).

6. PERFORMANCE EVALUATION

The performance evaluation was conducted using Erlang/OTP R9 (Release 9).[7] Its default runtime system architecture was process-centric. The communal ("shared heap") architecture is also available and can be selected by specifying the `-shared` flag when the system is started. Based on R9, we also implemented the modifications needed for the hybrid architecture using the local-by-default allocation strategy,[8] and included the aforementioned message analysis and transformation as a final 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. A compiler option invokes the message analysis.

All benchmarks were run 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 of integers, splits it into two sublists, and spawns two new processes for sorting the new lists. Since new lists are continously created, data received in one message is rarely passed on in another.

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 final data structure is sent to the parent process.

nag Creates a ring of 1000 processes. Each process sends out a large message (about 200 words) which will be passed on 200 steps in the ring. The **nag** is designed to test the behavior of memory architectures under different program characteristics. It comes in two flavours: **same** and **keep**. In the **same** variant, a single message is first created and distributed to all processes, and is then continuously received and resent. In the **keep** variant, each process keeps received messages live

[7]Available as open source from `www.erlang.org` and commercially from `www.erlang.com`.
[8]The hybrid architecture will be available in Erlang/OTP R10 and can be selected by specifying the `-hybrid` command line option at system startup time.

Table I. 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%
prettyprint	1	1,068	10,796	0.5	30%	9.9	1.8%
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%

by storing them in a list, and explicitly creates a new copy of each message before passing it on.

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 Compilation Overhead due to the Analysis

Table I shows sizes and compilation times for the benchmarks for compilation both to byte code and 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 size, byte code size, and compilation time for all their modules. We have also included the nonconcurrent programs **prettypr**, **pseudoknot**, and **inline** to show the overhead of the analysis on the compilation of single-module applications which contain functions of quite large size.

Using a depth of $k = 4$ (our current default) 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%), which is a hard program for this kind of depth-k-based analysis since it mostly consists of functions returning large nested tuples (all heap-allocated). Changing the depth to $k = 3$, the percentage drops to 13%, and with $k = 2$, to 9.6%.

More importantly, we can see from Table I that the analysis appears to scale well when performed on the whole program (**eddie_m**) rather than on a single module at a time (**eddie**).

Table II. Effectiveness of Message Analysis in the Hybrid System

Benchmark	Messages Sent	Messages Copied		k Words	k Words	Words Copied		Overapprox.
		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%	2.1%
msort_q	49,149	66.7%	0.0%	451	60,210	0.8%	0.07%	2.6%
worker	802	100.0%	0.0%	19,995	19.984	100.0%	0.01%	0.1%
nag (same)	203,000	100.0%	0.5%	606	40,827	2.0%	0.05%	−23.9%
nag (keep)	203,000	100.0%	0.5%	40,606	40,627	100.0%	0.05%	0.0%
eddie	40,028	100.0%	0.1%	200	421	81.0%	33.47%	0.0%
eddie_m	40,028	100.0%	0.1%	260	421	81.0%	19.22%	0.0%
mnesia	1,061,290	100.0%	25.2%	5,461	11,203	77.8%	33.68%	10.5%

6.3 Effectiveness of the Message Analysis

Table II shows the 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.[9] We also show the amount of words allocated directly on the shared heap when the analysis is enabled (referred to as "words preallocated"). "Words sent" shows the total size of sent messages, while "words copied" are the percentages of words that actually needed copying. The final column shows the amount of overapproximation, which is computed as the number of words preallocated compared to the difference in copying with and without analysis.

In the hybrid system, the number of words copied also includes forced copying when allocating message data (see Section 5), 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 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 classifies all data as shared will have the same effect for the hybrid system); however, storing data in shared memory can have a higher cost in a multithreaded implementation.

It is clear from Table II that, especially when a large amount of data is 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 (**eddie_m**), only 19% of the sent data is copied. We can, furthermore, see that message analysis typically causes a significant portion of the message data to be preallocated on the shared heap (58–100%) with only a small amount of overapproximation (the amount of "overapproximation" for the **nag (same)** benchmark is actually negative due to sharing).

In the **mnesia** benchmark, however, we encountered for the first (and so far, only) time a problematic case of overapproximation. The numbers shown in Tables II and III were measured when only 28 out of 29 modules were compiled using message analysis. When additionally compiling the final module with

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

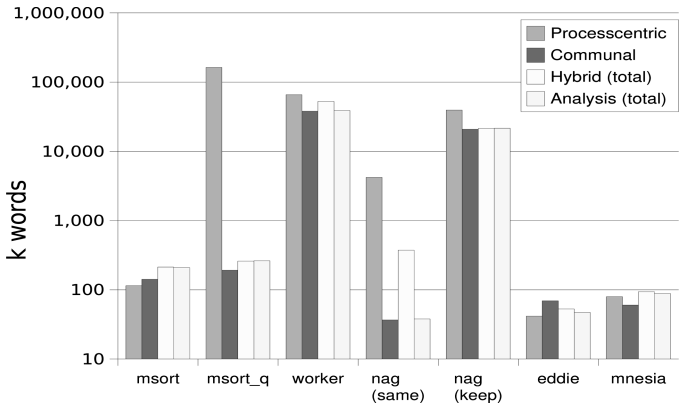


Fig. 7. Total heap usage (k words).

the analysis enabled, the number of words copied almost doubled. Studying the code in question as well as the effects of the analysis, we discovered that while a small fraction of the overapproximation might be avoided by a more precise analysis (e.g., distingushing call contexts), the main problem could be attributed to the style of programming: The rogue module was written so that whenever an error occurred or an "info query" was received, large parts of otherwise local data was passed in a message to the outside world. This meant that many data structures that were normally used only for internal bookkeeping (and were 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 to 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.

6.4 Memory Utilization

Recall that our main goal is not only 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 7 shows total heap usage in the different systems in terms of the maximum heap sizes at garbage collection time[10] (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 III. 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 the reuse of message data can reduce the sizes of the local heaps as 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

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

Table III. Heap Utilization (*k words)*

			Plain Hybrid		With Analysis		
Benchmark	Process-centric	Communal	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

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 message analysis can improve the memory behaviour of the hybrid system, sometimes even reducing the size of the shared area (figures showing the heap usage of individual benchmarks can be found in Appendix A).

A seemingly anomalous detail in Table III is that in the **nag (same)** benchmark, the use of the shared area is much higher without the analysis than with it. This is somewhat counterintuitive, as without the analysis, nothing is preallocated on the shared area. The reason for this behavior is that when a message is copied-on-demand, the original reference is not updated so as to point to the shared heap. We did not originally think this detail would be worthwhile to handle. It turns out, however, that in a program like this one, where effectively, a multicast is performed which distributes a single 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 this effect for this particular program, but does not make the hybrid system immune to it). If the implementation of message copying had been modified so that it also updated pointers within the original message to point to the shared copy instead, such repeated deep-copying could have been avoided (only the topmost constructor would be copied each time). However, if the message is a large object without internal pointers, such as an Erlang "binary" (a chunk of bytes) or an array of integers in another language, there are no internal pointers to update. In this case, one way of avoiding repeated copying 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 the 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), 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 that the memory can be used by another process, keeping the total memory usage low; however, in a

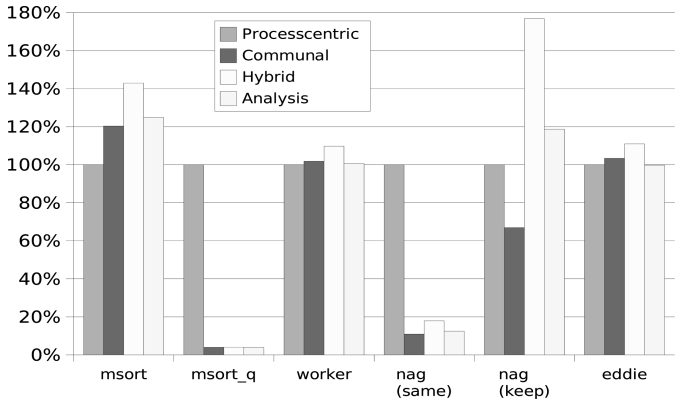


Fig. 8. Normalized runtimes.

multithreaded implementation we do not want processes to share their scratch memories).

6.5 Runtime Performance

Figure 8 shows the execution times for the benchmarks[11] 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.

Note that in a single-processor setting, the hybrid system can never hope to be faster than both the process-centric system and the communal system for the same benchmark. When a relatively small part of the total runtime is spent in message passing (as in **msort**), the process-centric system tends to win because of its simplicity, which results in better usage of machine registers and fewer runtime tests. When a lot of data is being passed in messages, the communal architecture usually wins because of its fast message passing and the improved cache behaviour due to sharing. The hybrid system tries to combine these properties into a memory architecture which is scalable in a multithreaded or distributed shared memory environment; as such, these numbers are quite promising.

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

7. RELATIONSHIP TO 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 light on message analysis by comparing it with escape analysis and region inference.

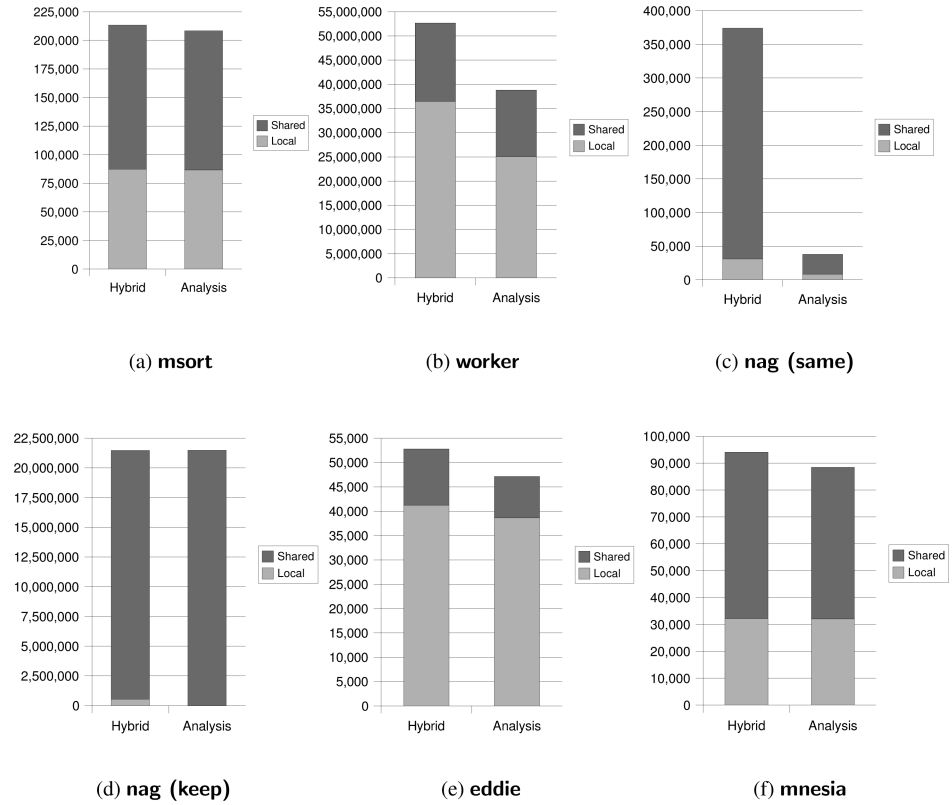
[11]Times for **mnesia** are not shown, as runtimes do not make sense for this benchmark.

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 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 previously mentioned, our analysis framework can be best understood as an extension of Shivers' [1988] control flow analysis and 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 by 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. [2003] used a reachability-graph-based escape analysis for the same purposes. Ruf [2000] focused on synchronization removal by regarding only properties over the whole lifetimes of objects, tracking the flow of values through the 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, we note that it addresses the problem of using analysis to guide the memory allocator 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 this data in the shared area of the hybrid system and eliminating the need for copying at send time, thus making it possible to remove some runtime 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, for example, 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, a concept reinvented many times but first introduced in Ross [1967], typically allocates objects in separate areas according to their lifetimes. The compiler, guided by a



(a) **msort** (b) **worker** (c) **nag (same)**

(d) **nag (keep)** (e) **eddie** (f) **mnesia**

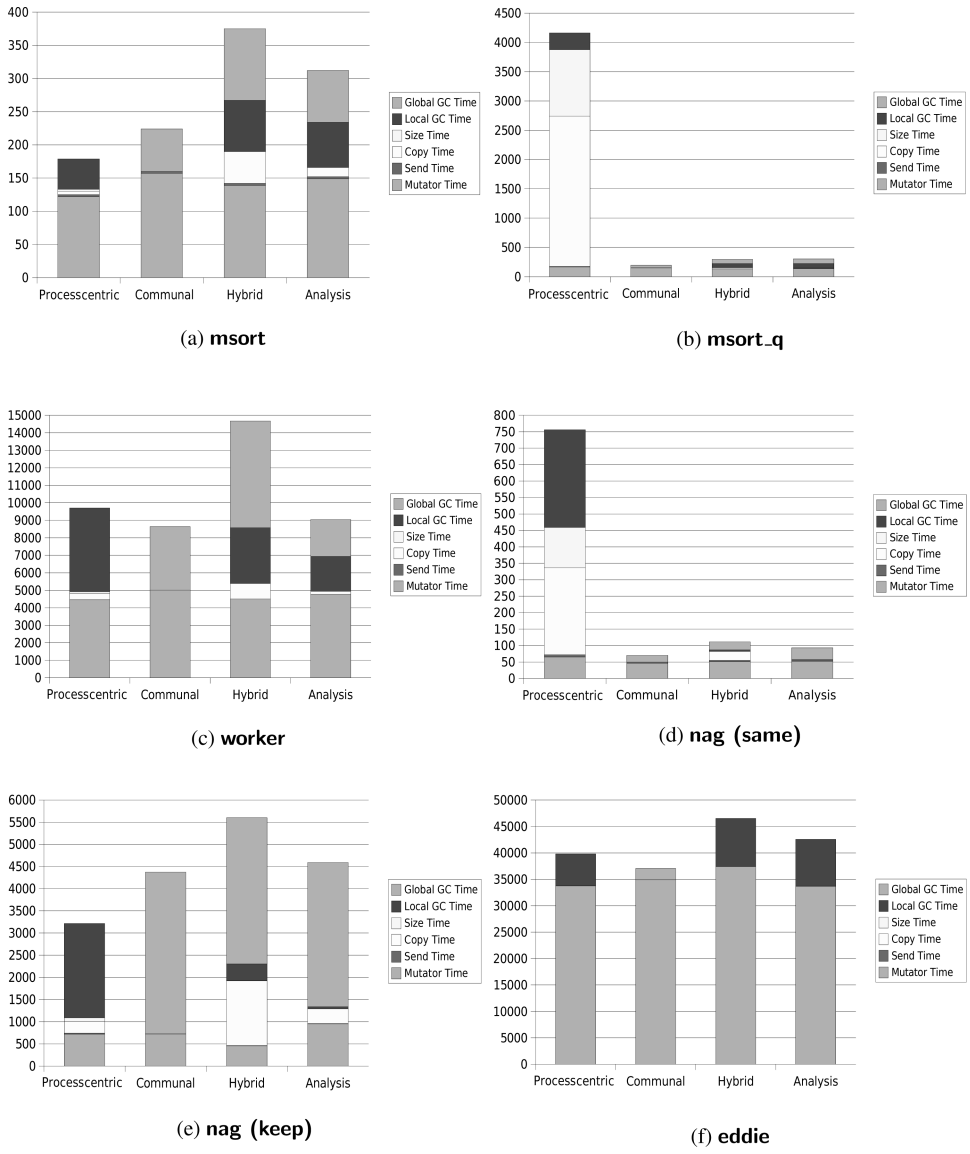
Fig. 9. Heap size improvement due to message analysis.

static analysis called *region inference* that was introduced by Tofte and Talpin [1994], 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, we 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 by the experimental evaluation, the analysis is, in practice, both fast and precise enough to discover most of the data that will become part of some message.

For a single-processor setting, what our work has shown is that the hybrid architecture, even without analysis, is a reasonable architecture in itself. The use



(a) **msort** (b) **msort_q**

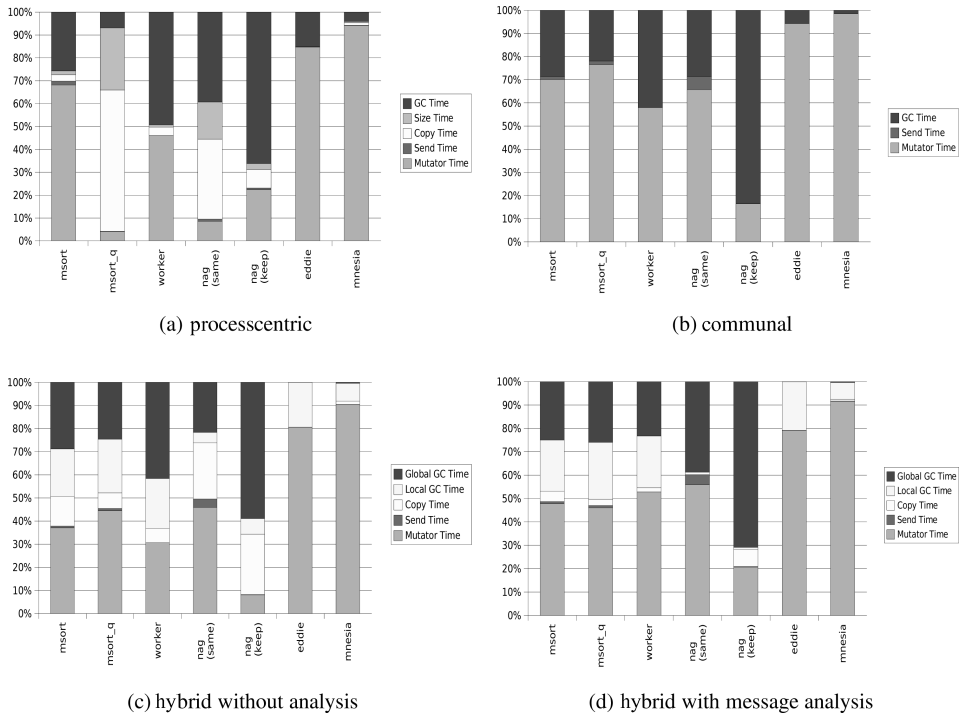
(c) **worker** (d) **nag (same)**

(e) **nag (keep)** (f) **eddie**

Fig. 10. Performance of individual benchmarks.

of message analysis further improves its time and space performance. Moreover, the resulting system is the enabling technology for a high-performance implementation on top of a multithreaded or distributed shared memory implementation. We are currently working towards such an implementation.

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



(a) processcentric (b) communal

(c) hybrid without analysis (d) hybrid with message analysis

Fig. 11. Execution time percentages for the different systems.

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,[12] 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 method of programming.

APPENDIX

A. HEAP MEMORY USAGE

Figure 9 shows the effect of message analysis-guided heap allocation on the sizes of heaps in the hybrid system. While the changes in the allocation pattern are in some cases not big enough to overly affect the heap enlargement policy, as in **msort**, and are rarely as extreme as in **nag (same)**, message analysis typically makes the hybrid system almost as memory-efficient as the communal system (see Figure 7). We have left out **msort_q** since it behaves very much like **msort** in this respect.

B. EXECUTION TIMES

Figure 10 shows execution time details for the benchmarks (see also Figure 8). It must be noted that the current (copying, two-generational) garbage collector

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

is tailored to 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 underway, but is not expected to be ready any time soon.

Figure 11 makes it easier to see, for each benchmark, where the time is spent in the different systems. It is, for example, clear that message analysis removes much of the copying overhead in the hybrid system.

ACKNOWLEDGMENTS

We thank all three anonymous reviewers for their constructive comments and helpful suggestions, which have considerably improved the presentation of our work.

REFERENCES

ARMSTRONG, J., VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. 1996. *Concurrent Programming in Erlang*, 2nd ed. Prentice Hall, Herfordshire, UK.

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, 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, 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, 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*, G. L. Steele, ed. ACM Press, New York, 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, 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, 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, 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. 76–87.

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, 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, Los Alamitos, Calif. 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, 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, 88–99.

JONES, R. E. AND LINS, R. 1996. *Garbage Collection: Algorithms for Automatic Memory Management*. John Wiley & Sons, Chichester, UK.

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. LNCS, vol. 1551. Springer, Berlin, 152–163.

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

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, 116–127.

PETTERSSON, M., SAGONAS, K., AND JOHANSSON, E. 2002. The HiPE/x86 Erlang compiler: System description and performance evaluation. In *Proceedings of the 6th International Symposium on Functional and Logic Programming*, Z. Hu and M. Rodríguez-Artalejo, eds. LNCS, vol. 2441. Springer, Berlin, 228–244.

ROSS, D. T. 1967. The AED free storage package. *Comm. ACM 10,* 8, 481–492.

RUF, E. 2000. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, 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, 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, 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, 18–24.

TOFTE, M. AND TALPIN, J.-P. 1994. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, 188–201.

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

Received April 2004; revised October 2004; accepted December 2004