# Message Analysis for Concurrent Languages*

Richard Carlsson, Konstantinos Sagonas, and Jesper Wilhelmsson

Computing Science Department, Uppsala University, Sweden
{richardc,kostis,jesperw}@csd.uu.se

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

## 1 Introduction

Many programming languages nowadays come with some form of built-in support for concurrent processes (or threads). Depending on the concurrency model of the language, interprocess communication takes place either through synchronized shared structures (as e.g. in Java), using synchronous message passing on typed channels (as e.g. in Concurrent ML), or using asynchronous message passing (as e.g. in ERLANG). Most of these languages typically also require support for automatic memory management, usually implemented using a garbage collector. So far, research has largely focused on the memory reclamation aspects of these concurrent systems. As a result, by now, many different garbage collection techniques have been proposed and their characteristics are well-known; see e.g. [15].

A less treated, albeit key issue in the design of a concurrent language implementation is that of memory allocation. It is clear that, regardless of the concurrency model of the language, there exist several different ways of structuring the memory architecture, each having its pros and cons. Perhaps surprisingly, till recently, there has not been any in-depth investigation of the performance tradeoffs that are involved in the choice between these alternative architectures. In [14], we provided the first detailed characterization of the advantages and disadvantages of different memory architectures in a language where communication occurs through message passing.

* Research supported in part by the ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson Development.

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

*Our contributions* Our first contribution, which motivates our analysis, is in the area of runtime system organization. Based on the pros and cons of different memory architectures described in [14], we describe two different variants of a runtime system architecture that has process-specific areas for allocation of local data, and a common area for data that is shared between communicating processes (i.e., is part of some message). In doing so, it allows interprocess communication to occur without actual copying, uses less overall space due to avoiding data replication, and allows for the frequent process-local heap collections to take place without a need for global synchronization of processes, reducing the level of system irresponsiveness due to garbage collection.

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

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

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

## 2 Preliminaries & Prior Work

### 2.1 Erlang and Core Erlang

ERLANG [1] is a strict, dynamically typed functional programming language with support for concurrency, distribution, communication, fault-tolerance, on-the-fly code replacement, and automatic memory management. ERLANG was designed to ease the programming of large soft real-time control systems like those commonly developed in the telecommunications industry. It has so far been used quite successfully both by

Ericsson and other companies around the world to construct large (several hundred thousand lines of code) commercial applications.

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

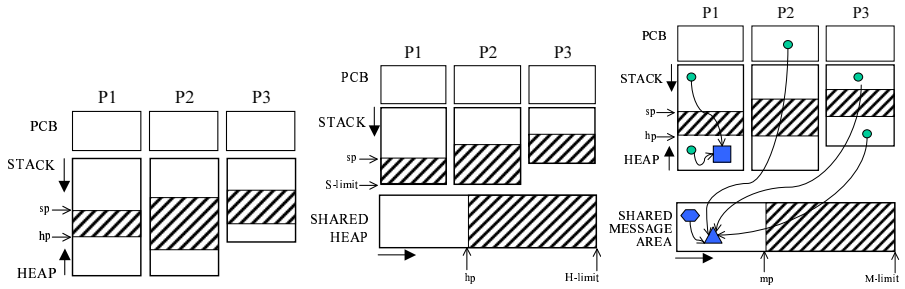
Processes in ERLANG are extremely light-weight (lighter than OS threads), their number in typical applications can be large (in some cases up to 50,000 processes on a single node), and their memory requirements vary dynamically. ERLANG's concurrency primitives – `spawn`, "`!`" (send), and `receive` – allow a process to spawn new processes and communicate with other processes through asynchronous message passing. Any value can be sent as a message and processes may be located on any machine. Each process has a *mailbox*, essentially a message queue, where all messages sent to the process will arrive. Message selection from the mailbox is done by pattern matching. In send operations, the receiver is specified by its process identifier, regardless of where it is located, making distribution all but invisible. To support robust systems, a process can register to receive a message if some other process terminates. ERLANG provides mechanisms for allowing a process to timeout while waiting for messages and a catch/throw-style exception mechanism for error handling.

ERLANG is often used in "five nines" high-availability (i.e., 99.999% of the time available) systems, where down-time is required to be less than five minutes per year. Such systems cannot be taken down, upgraded, and restarted when software patches and upgrades arrive, since that would not respect the availability requirement. Consequently, ERLANG systems support upgrading code while the system is running, a mechanism known as *dynamic code replacement*.

Core Erlang [7, 6] 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 unusual and complicated variable scoping rules, fixed-order evaluation, and only allows top-level function definitions, Core Erlang is similar to the untyped lambda calculus with `let`- and `letrec`-bindings, and imposes no restrictions on the evaluation order of arguments.

### 2.2 Heap Architectures for Concurrent Languages using Message Passing

In [14] we examined three different runtime system architectures for concurrent language implementations: One *process-centric* where each process allocates and manages its private memory area and all messages have to be copied between processes, one which is *communal* and all processes get to share the same heap, and finally we proposed a *hybrid* runtime system architecture where each process has a private heap for local data but where a shared heap is used for data sent as messages. Figure 1 depicts memory areas of these architectures when three processes are currently in the system; shaded areas show currently unused memory; the filled shapes and arrows in Fig. 1(c) represent messages and pointers.




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

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

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

**Process-centric.** This is currently the default configuration of Erlang/OTP. Interprocess communication requires copying of messages, thus is an $O(n)$ operation where $n$ is the message size. Also, memory fragmentation is high. Pros are that the garbage collection times and pauses are expected to be small (as the root set need only consist of the stack of the process requiring collection), and upon termination of a process, its allocated memory area can be reclaimed without GC. This property in turn encourages the use of processes as a form of *programmer-controlled regions*: a computation that requires a lot of auxiliary space can be performed in a separate process that sends its result as a message to its consumer and then dies. This memory architecture has recently also been exploited in the context of Java; see [11].

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

**Hybrid.** Tries to combine the advantages of the above two architectures: interprocess communication is fast and GC latency for the frequent collections of the per-process heaps is expected to be small. Also, this architecture allows for reclamation of data of short-lived, memory-intensive processes to happen without GC, but simply by attaching the process-local heap to a free list. However, to take advantage of this architecture, the system should be able to distinguish between data that is process-local and data which is to be shared and used as messages. This can be achieved by user annotations on the source code, by dynamically monitoring the creation of data as recently proposed in [11], or by a static analysis as we describe in Sect. 4.

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

## 3 The Hybrid Architecture

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

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

**Local allocation of non-messages.** Here, only data that is known to *not* be part of a message may be allocated on the process-local heap, while all other data is allocated on the shared heap. This gives $O(1)$ process communication for processes residing on the same node, since all possible messages are guaranteed to already be in the shared area, but utilization of the local heaps depends on the ability to decide through program analysis which data is definitely not shared. This approach is used by [19]. Because it is not possible in general to determine what will become part of a message, underapproximation is necessary. In the worst case, nothing is allocated in the process-local heaps, and the behaviour of the hybrid architecture with this allocation strategy reduces to that of the shared heap architecture.

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

**Copying of messages.** If the second strategy is used, as is the case in our implementation of the hybrid system, we must be prepared to copy (parts of) messages as necessary to ensure the pointer directionality invariant. Since we do not know how much of a message needs to be copied and how much already resides in the shared area, we can not

ensure that the space available on the shared heap will be sufficient before we begin to copy data.

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

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

## 4 Message Analysis

To use the hybrid architecture without user annotations on what is to be allocated on the local and shared heap respectively, program analysis is necessary. If data is allocated on the shared heap by default, we need to single out the data which is guaranteed to not be included in any message, so it can be allocated on the per-process heap. This amounts to escape analysis of process-local data [4, 5, 8].

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

### 4.1 The analyzed language

Although our analyses have been implemented for the complete Core Erlang language, for the purposes of this paper, the details of Core Erlang are unimportant. To keep the exposition simple, we instead define a sufficiently powerful language of A-normal forms [12], shown in Fig. 2, with the relevant semantics of the core language (strict, higher-order, dynamically typed and without destructive updates), and with operators for asynchronous send, blocking receive, and process spawning. We also make the simplifying assumption that all primitive operations return atomic values and do not cause escapement; however, our actual implementation does not rely on that assumption.

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

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

**Fig. 2.** A mini-Erlang language

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

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

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

### 4.2 General framework

The analyses we have this far implemented are first-order dataflow analyses, and are best understood as extensions of Shivers' closure analysis [18]. Indeed, we assume that closure analysis has been done, so that:

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

The domain $V$ is defined as follows:

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

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

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

$$\sqsubseteq = \bigcup_{i\geq 0} \sqsubseteq_i^*$$

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

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

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

$$\mathcal{V}_v[\![c]\!] = \bot$$
$$\mathcal{V}_v[\![x]\!] = \textit{Val}(x)$$

$$\mathcal{V}_e[\![v]\!] = \mathcal{V}_v[\![v]\!]$$
$$\mathcal{V}_e[\![(v_1\ v_2)^l]\!] = \textit{In}(l)$$
$$\mathcal{V}_e[\![\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2]\!] = \mathcal{V}_e[\![e_1]\!] \sqcup \mathcal{V}_e[\![e_2]\!]$$
$$\mathcal{V}_e[\![\texttt{let } x = b \texttt{ in } e]\!] = \mathcal{V}_e[\![e]\!]$$

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

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

$$\mathcal{V}_b[\![\texttt{spawn}\ (v_1\ v_2)^l]\!] = \bot$$
$$\mathcal{V}_b[\![\texttt{primop}\ o(v_1, ..., v_n)]\!] = \bot$$

where

$$\textit{cons}\ l\ x\ y = (\{l\}, \langle x \rangle) \sqcup y$$
$$\textit{tuple}\ l\ \langle x_1, \ldots, x_n \rangle = (\{l\}, \langle x_1, \ldots, x_n \rangle)$$

and

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

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

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

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

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

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

### 4.3 Termination

Finding the least solution for *Val*, *In*, and *Out* to the above constraint system for some program by 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 $\theta_k$ as:

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

where

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

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

Note that without the special treatment of list constructors, this form of approximation would generally lose too much information; in particular, recursion over a list would confuse the spine constructors with the elements of the same list. In essence, we have a "poor man's escape analysis on lists" [16] for a dynamically typed language.

### 4.4 Escape analysis

As mentioned, in the scheme where data is allocated on the shared heap by default, the analysis needs to determine which heap-allocated data cannot escape the creating process, or reversely, which data can possibly escape. Following [18], we let *Escaped* represent the set of all escaping values, and add the following straightforward rules:

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

After the fixpoint iteration converges, if the label of a data constructor operation (including lambdas) in the program is not in $\mathit{labels}(\mathit{Escaped})$, the result produced by that operation does not escape the process.

It is easy to extend this escape analysis to simultaneously perform a more precise closure analysis than [18], which only uses sets, but doing so here would cloud the issues of this paper. Also, ERLANG programs tend to use fewer higher-order functions, in comparison with typical programs in e.g. Scheme or ML, so we expect that the improvements to the determined call graphs would not be significant in practice. Note that although our analysis is not in itself higher-order, we are able to handle the full higher-order language with generally sufficient precision.

### 4.5 Message analysis

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

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

We have the following rules:

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

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

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

## 5 Using the Analysis Information

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

### 5.1 Local allocation of non-messages

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

### 5.2 Shared allocation of possible messages

This requires two things:

1. Each data constructor in the program such that a value constructed at that point is likely to be a part of a message, is rewritten so that the allocation will be done on the shared heap.
2. For each argument of those message constructors, and for the message argument of each send-operation, if the passed value is not guaranteed to already be allocated on the shared heap, the argument is wrapped in a call to `copy`, in order to maintain the pointer directionality requirement.

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

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

**Fig. 3.** ERLANG program example.

### 5.3 Example

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

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

For this example, our escape analysis determines that only the list constructors in the functions `zipwith3` and `filter` (lines 13 and 18, respectively) are guaranteed to not escape the executing process, and can be locally allocated. Since the actual elements of the list, created by the lambda passed to `zipwith3` (line 8), are being passed to an unknown function via `filter`, they must be conservatively viewed as escaping.

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

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

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

## 6 Performance Evaluation

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

### 6.1 The benchmarks

The performance evaluation was based on the following benchmarks:

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

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

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

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

### 6.2 Effectiveness of the message analysis

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

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

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

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

### 6.3 Compilation overhead due to the analysis

In the byte code compiler, the analysis takes on average 19% of the compilation time, with a minimum of 3%. However, the byte code compiler is fast and relatively simplistic; for example, it does not in itself perform any global data flow analyses. Including the message analysis as a stage in the more advanced HiPE native code compiler [13], its portion of the compilation time is below 10% in all benchmarks. ERLANG modules are separately compiled, and most source code files are small (less than 1000 lines). The numbers for **eddie** show the total code size and compilation times for all its modules. We have included the non-concurrent programs **prettyprint**, **pseudoknot**, and **inline** to show the overhead of the analysis on the compilation of larger single-module applications.

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

**Table 2.** Compilation and analysis times.

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

### 6.4 Runtime performance

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

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

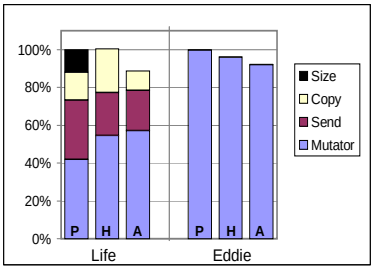



**Fig. 4.** Performance of non-synthetic programs.

In Fig. 4, the **life** benchmark shows the behaviour when a large number of small messages are being passed. The hybrid system with analysis is about $10\%$ faster than the process-centric system, but we can see that although enabling the analysis removes the need for actual copying of message data (cf. Table 1), we still have a small overhead for the runtime safety check performed at each send operation (this could in principle be removed), which is comparable to the total copying time in the process-centric system

when messages are very small. We can also see how the slightly more complicated bookkeeping for sending messages is noticeable in the process-centric system, and how on the other hand the mutator time can be larger in the hybrid system. (One reason is that allocation on the shared heap is more expensive.) In **eddie**, the message passing time is just a small fraction of the total runtime, and we suspect that the slightly better performance of the hybrid system is due to better locality because of message sharing (cf. Table 1).

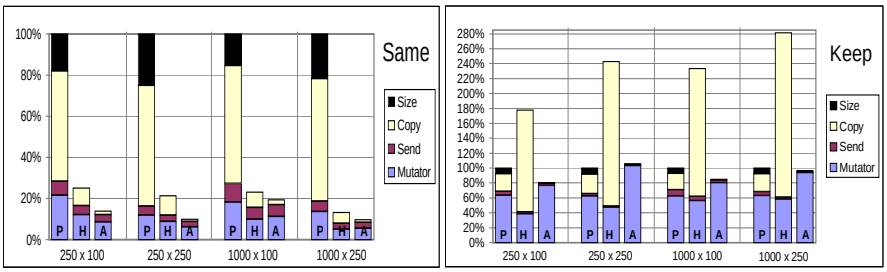



**Fig. 5.** Performance of the **same** and **keep** variants of the **nag** benchmark.

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

## 7 Related Work

Our message analysis is in many respects similar to escape analysis. Escape analysis was introduced by Park and Goldberg [16], and further refined by Deutsch [9] and Blanchet [3]. So far, its main application has been to permit stack allocation of data in functional languages. In [4], Blanchet extended his analysis to handle assignments and applied it to the Java language, allocating objects on the stack and also eliminating synchronization on objects that do not escape their creating thread. Concurrently with Blanchet's work, Bogda and Hölzle [5] 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.* [8] used a reachability graph based escape analysis for the same purposes. Ruf [17] focuses on synchronization removal by regarding only properties over the whole lifetimes of objects, tracking the flow of values through global state but sacrificing precision within methods and especially in the

presence of recursion. It should be noted that with the exception of [8], all these escape analyses rely heavily on static type information, and in general sacrifice precision in the presence of recursive data structures. Recursive data structures are extremely common in ERLANG and type information is not available in our context.

Our hybrid memory model is inspired in part by a runtime system architecture described by Doligez and Leroy in [10] 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 [2], an operating system for executing Java programs. Using escape analysis to guide a memory management system with thread-specific heaps was described by Steensgaard [19].

Notice that it is also possible to view the hybrid model as a runtime system architecture with a shared heap and separate *regions* for each process. Region-based memory management, introduced by Tofte and Talpin [20], typically allocates objects in separate areas according to their lifetimes. The compiler, guided by a static analysis called *region inference*, is responsible to generate code that deallocates these areas. The simplest form of region inference places objects in areas whose lifetimes coincide with that of their creating functions. In this respect, one can view the process-specific heaps of the hybrid model as regions whose lifetime coincides with that of the top-level function invocation of each process, and see our message analysis as a simple region inference algorithm for discovering data which outlives their creating processes.

## 8 Concluding Remarks

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

## References


1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
2. G. Back, W. C. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2000. http://www.cs.utah.edu/flux/papers/.
3. B. Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Conference Record of the 25th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'98)*, pages 25–37. ACM Press, Jan. 1998.

4. B. Blanchet. Escape analysis for object oriented languages. Application to Java$^{TM}$. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 20–34. ACM Press, Nov. 1999.
5. J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99)*, Nov. 1999.
6. R. Carlsson. An introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, Sept. 2001.
7. R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 030, Information Technology Department, Uppsala University, Nov. 2000.
8. J.-D. Choi, M. Gupta, M. Serrano, V. C. Shreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 1–19. ACM Press, Nov. 1999.
9. A. Deutsch. On the complexity of escape analysis. In *Conference Record of the 24th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 358–371, Jan. 1997.
10. D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–123. ACM Press, Jan. 1993.
11. T. Domani, G. Goldshtein, E. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *Proceedings of ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, pages 76–87. ACM Press, June 2002.
12. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, June 1993.
13. E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, Sept. 2000.
14. E. Johansson, K. Sagonas, and J. Wilhelmsson. Heap architectures for concurrent languages using message passing. In *Proceedings of ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, pages 88–99. ACM Press, June 2002.
15. R. E. Jones and R. Lins. *Garbage Collection: Algorithms for automatic memory management*. John Wiley & Sons, 1996.
16. Y. G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–127. ACM Press, July 1992.
17. E. Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 208–218. ACM Press, June 2000.
18. O. Shivers. Control fbw analysis in Scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174. ACM Press, June 1988.
19. B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, pages 18–24. ACM Press, Oct. 2000.
20. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, Feb. 1997.