Efficient Memory Management for Concurrent Programs that Use Message Passing ⋆,⋆⋆

Konstantinos Sagonas, Jesper Wilhelmsson *

Department of Information Technology, Uppsala University, Sweden

Abstract

We present an efficient memory management scheme for concurrent programming languages where communication occurs using message passing with copying semantics. The runtime system is built around process-local heaps, which frees the memory manager from redundant synchronization in a multi-threaded implementation and allows the memory reclamation of process-local heaps to be a private business and to often take place without ever triggering garbage collection. The allocator is guided by a static analysis which speculatively allocates data possibly used as messages in a shared memory area. To respect the (soft) real-time requirements of the language, we develop and present in detail a generational, incremental garbage collection scheme tailored to the characteristics of this runtime system. The incremental collector imposes no overhead on the mutator, requires no costly barrier mechanisms, has a relatively small space overhead and can be scheduled either based on a time or on a work quantum. We have implemented these schemes in the context of an industrial-strength implementation of a concurrent functional language used to develop large-scale, highly concurrent, telecommunication applications. Our measurements across a range of applications indicate that the incremental collector imposes only very small overhead on the total runtime, can achieve very short pause times (1 millisecond or less) while being able to sustain a high degree of mutator utilization.

⋆ Research supported in part by grant #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 a conference paper by the same authors titled "Message Analysis-Guided Allocation and Low-Pause Incremental Garbage Collection in a Concurrent Language" which appeared in the *Proceedings of ISMM'04: Fourth ACM SIGPLAN International Symposium on Memory Management*, pages 1–12, Oct. 2004. `http://doi.acm.org/10.1145/1029873.1029875`.

* Corresponding author.
Email addresses: `kostis@it.uu.se` (Konstantinos Sagonas), `jesperw@it.uu.se` (Jesper Wilhelmsson).

1 Introduction

Concurrent real-time programming languages with automatic memory management present new challenges to programming language implementors. One is how to tailor the runtime system to the intended use of data and achieve performance which does not degrade for highly concurrent applications and scales well in a multi-processor setting. By highly concurrent, we mean applications consisting of several thousand or several hundreds of thousands of threads (or cooperating processes). Another challenge is to achieve the high level of responsiveness that is required by applications from domains such as embedded control and telecommunication systems.

Taking up the latter challenge becomes tricky when automatic memory management is performed using garbage collection (GC). The naïve "stop-the-world" approach, where threads repeatedly interrupt execution of a user's program in order to perform garbage collection, is clearly inappropriate for applications with real-time requirements. It is also problematic on principle. This is because it introduces a point of global synchronization between otherwise independent threads and possibly also tasks (i.e. between collections of threads conceptually forming independent work units). Also, it provides no guarantees for bounds on the length of the individual pauses or for sufficient progress by the application.

Despite the significant progress in developing automatic memory reclamation techniques with real-time characteristics [3,4,8,16], each technique is designed around a number of (often implicit) assumptions about the architecture of the runtime system that might not be the most appropriate ones to follow in a different context and thus may not be easily adaptable. Also, different languages have different characteristics which influence the trade-offs associated with each technique. For example, many collectors for object-oriented languages such as Java assume that allocating an extra header word for each object does not penalize execution times by much and does not impose a significant space overhead. In functional languages where the majority of objects occupy just two words, adding an extra word to all objects just for the collector's convenience is typically not a viable alternative. (After all, the collector, although important, is only a part of the implementation.) Similarly, the semantics of a language may favor the use of a read rather than a write barrier and the absence of destructive updates may allow for more liberal forms of incremental collection (for example, based on replication of objects [18]). Finally, it is clear that the type of GC which is employed interacts with and is influenced by the object allocation technique that is used (in a thread-local or shared memory region, such that allows the regions to be collected independently, and so on) and, if applicable, the static analysis which guides the allocator. In short, it is very difficult to come up with techniques that can be readily tailored to all languages and runtime environments and most implementation choices which influence the performance of garbage collection have non-trivial interactions. An important point, which this

article implicitly tries to make a case for, is that one needs a carefully selected *combination* of memory allocation and garbage collection techniques in order to achieve high performance. Memory allocation and memory reclamation concerns typically involve tradeoffs which influence each other.

The work we describe in this article is performed in the context of a concurrent functional language without destructive updates and where interprocess communication occurs using message passing. We first review the language (Section 2) and then present the details of a runtime system whose memory manager splits the allocated memory into areas based on the intended use of data (Section 3). Its memory allocator is guided by a static analysis, which speculatively allocates data possibly used as messages in a shared memory area. Based on the characteristics of each memory area, we then discuss the various types of garbage collection methods which are employed (Section 4). In the main body of this article (Sections 5 and 6), which contain its primary contribution, we develop and analyze a generational incremental garbage collection scheme tailored to the runtime system of Section 3. Notable characteristics are that the collector imposes no noticeable overhead on the mutator, requires no costly barrier mechanisms, has a relatively small space overhead, and can be scheduled on a very fine-grained manner using either a work or a time quantum. [1] We report on the performance of these memory management schemes across a range of applications (Section 7) and show that when using the incremental collector, through various optimizations which we discuss in the article, we are able to sustain the overall performance of the system, obtain extremely small pause times, and achieve a high degree of mutator utilization. The article ends with a review of related work (Section 8) and some concluding remarks (Section 9).

2 Preliminaries

To describe the context of work, we briefly review the Erlang language (Section 2.1) and the runtime system architectures of the Erlang/OTP system (Section 2.2).

2.1 Erlang and Erlang/OTP

Erlang [2] is a strict (i.e., uses call-by-value), dynamically typed functional programming language with support for concurrency, communication, distribution and fault-tolerance. It offers automatic memory management and supports multiple platforms. Erlang was designed to ease the programming of soft real-time con-

[1] The time-based variant of the incremental garbage collection algorithm we develop is actually a real-time GC algorithm, but we refrain from referring to it as such since in our current implementation it is applied to only part of the memory reclaimed using GC.

trol systems commonly developed by the data- and tele-communications industry. Its implementation, the Erlang/OTP system, has so far been used quite successfully both by Ericsson and by other companies around the world (e.g., T-Mobile, Nortel Networks) to develop large commercial applications.

Erlang's basic data types are atoms, numbers (floats and arbitrary precision integers), and process identifiers; compound data types are lists and tuples. A notation for objects (records in the Erlang lingo) is supported, but the underlying implementation of records is the same as tuples. To allow efficient implementation of telecommunication protocols, Erlang also includes a *binary* data type (a vector of byte-sized data) and a notation to perform pattern matching on binaries. There is no destructive assignment of variables or objects stored in the heap (i.e., all heap data is immutable) and consequently cyclic references cannot be created. Because recursion is the only means to express iteration, tail call optimization is a required feature of Erlang implementations.

Processes in Erlang are extremely light-weight (significantly lighter than for example Java threads) and their number in typical applications is quite large (it is not uncommon to spawn several hundred thousand processes on a single node). Erlang's concurrency primitives — `spawn`, "`!`" (send), and `receive` — allow a process to spawn new processes and communicate with other processes through asynchronous message passing with *copying semantics*. At the language level, this means that it is indistinguishable whether processes share the same or have physically different copies of a message. [2] It also means that, if needed, messages can be replicated at will by the runtime system. For example, the mutator is allowed to access messages in both the from-space and to-space of a copying collector. A message is simply an Erlang term, which means that any data value can be sent as a message, and the message's recipient may be located on any machine on the network. Each process has a *mailbox*, essentially a message queue, which contains pointers to each message sent to the process. The message itself, being a term, is stored on the heap. Message reception from the mailbox is *selective*: the `receive` statement scans the mailbox and selects from it the first message that matches the patterns that it expects. In send operations, the receiver is specified by its process identifier, regardless of where it is located, making distribution all but invisible. To support robust systems, a process can register to receive a message if another one terminates. Erlang also provides a mechanism that allows a process to time-out while waiting for messages and a try/catch-style exception mechanism for error handling.

[2] In Erlang there are are no destructive updates, but even if they were, the copying semantics of message passing dictate that updates to a message by for example the sender would be invisible to the receiver after the message has been sent. On the other hand, the copying semantics of message passing do *not* dictate that messages have to be eagerly copied when sent, which allows for efficient implementations of message-passing concurrency such as those described in this article.

Erlang is often used in high-availability large-scale embedded systems, such as telephone centers, with very strict requirements on continuous and smooth operation (typically, down-time is required to be less than five minutes per year). Moreover, these systems typically also require high level of responsiveness, and the soft real-time concerns of the language call for fast garbage collection techniques such as the ones presented in this article.

2.2 The three runtime systems of Erlang/OTP

Until quite recently, the Erlang/OTP runtime system was based on a *process-centric* architecture; that is, an architecture where each process allocates and manages its private memory area. The main reason why this memory allocation scheme was chosen was that it was believed that it results in lower garbage collection latency. Wanting to investigate the validity of this belief, in a previous paper [14] we examined two alternative runtime system architectures for implementing concurrency through message passing: one which is *communal* where all processes get to share the same heap, and a *hybrid* scheme where each process has a private heap for process-local data but where a shared heap is used for data sent as messages and thus shared between processes. All three architectures are included in the Erlang/OTP release. Selecting the desired one is a matter of invoking the system with the appropriate command-line option. We briefly review their characteristics.

Process-centric In this architecture, interprocess communication requires copying of messages and thus is an $O(n)$ operation where n is the message size. Since garbage collection of process-local heaps typically happens only for processes that overflow their area and the collector does not have global control of allocated memory, other processes often hold on to more memory than they actually need. Thus memory fragmentation tends to be higher than in a shared heap architecture. 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 in constant time (without garbage collection).

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

Hybrid An architecture that tries to combine the advantages of the above two architectures: interprocess communication can be fast and GC latency for the fre-

quent collections of the process-local heaps is expected to be small. No locking is required for the garbage collection of the process-local heaps, and the pressure on the shared heap is reduced so that it does not need to be garbage collected as often. Also, as in the process-centric architecture, when a process terminates, its local memory can be reclaimed by simply attaching it to a free-list.

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

In this article we concentrate on the hybrid architecture. The reasons are both pragmatic and principled. Pragmatic because this architecture behaves best in practice, and principled because it combines the best performance characteristics of the other two runtime system architectures and is the enabling technology for a scalable multi-threaded implementation. Also, the garbage collection techniques developed in its context are applicable to the other runtime system architectures with only minor adjustments.

3 Memory Organization of the Hybrid Architecture

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

3.1 The pointer directionality invariants

A key point in the design of the hybrid architecture is to be able to garbage collect the process-local heaps individually. In particular, we want to avoid having to consider roots outside the memory area of the process which is being collected. In a multi-threaded system, this allows process-local heaps to be collected indepen-

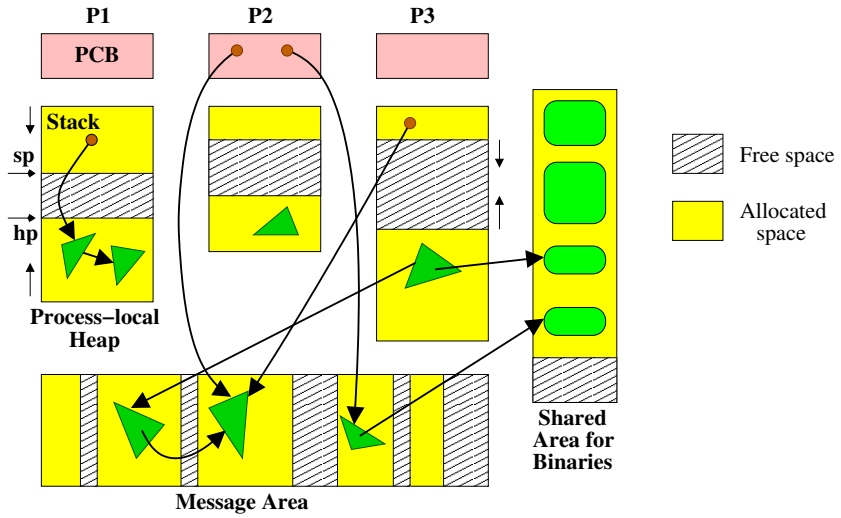


Fig. 1. Layout of the hybrid architecture showing allowed references.

dently and without synchronization. To achieve this, we maintain as an invariant of the runtime system that there are no pointers from the shared areas to the local heaps, nor from one process-local area to another. These invariants are maintained by the allocator. Once allocation has taken place, the invariants are automatically preserved since there are no pointer destructions in our setting. If, on the other hand, pointers from the shared areas to the local heaps were allowed or could be created during execution, then these would have to be traced so that what they point to would be considered live during process-local collections. Keeping track of them could be achieved by a write barrier, but we want to avoid completely the time and space overhead that a barrier mechanism incurs, no matter how small this overhead might be made.

Figure 1 shows all types of pointers that can exist in the system. In particular:

- The area for binaries does not contain pointers to the process-local heaps or the message area. Binaries are objects (e.g., a network packet) whose contents are raw sequences of zeros and ones and consequently this area has no references visible to the garbage collector.
- The message area only contains references to the shared area for binaries or to objects within the message area itself.
- None of the areas contains any cyclic data.

The pointer directionality invariants for the message area are also crucial for our choice of memory allocation strategy, since they make it easy to test at runtime whether or not a piece of data resides in the message area by making a simple $O(1)$ pointer comparison.

3.2 Allocation in the hybrid architecture

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

For the purposes of this article, the details of the message analysis are unimportant and the interested reader is referred to the corresponding paper [5]. Instead, it suffices to understand how the analysis guides allocation of data in the compiler. The allocation can be described as *allocation by default on the local heap and shared allocation of possible messages*. More specifically, data that is *likely* to be part of a message is allocated speculatively on the shared heap and all other data on the process-local heaps. To maintain the pointer directionality invariants, this in turn requires that the message operands of all send operations are wrapped with a copy-on-demand operation, which verifies that the message resides in the shared area, and otherwise copies the locally allocated parts to the shared heap. However, if the message analysis can determine that a message operand *must* already be on the shared heap, the test can be statically eliminated. In practice this happens often as the analysis is quite effective; see the performance results reported in reference [5].

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

3.3 Allocation characteristics of Erlang programs

From the memory manager's perspective, the Erlang heap only contains two kinds of objects: cons cells (that is, list objects with a head and a tail element whose size is just two words) and boxed objects. Boxed objects are tuples, arbitrary precision integers, floats, binaries, and function closures. Boxed objects contain a header word which either directly or indirectly includes information about the object's size. In contrast, there is no header word for cons cells. Fig. 2 shows a detailed breakdown of the allocation characteristics of three of the benchmark programs

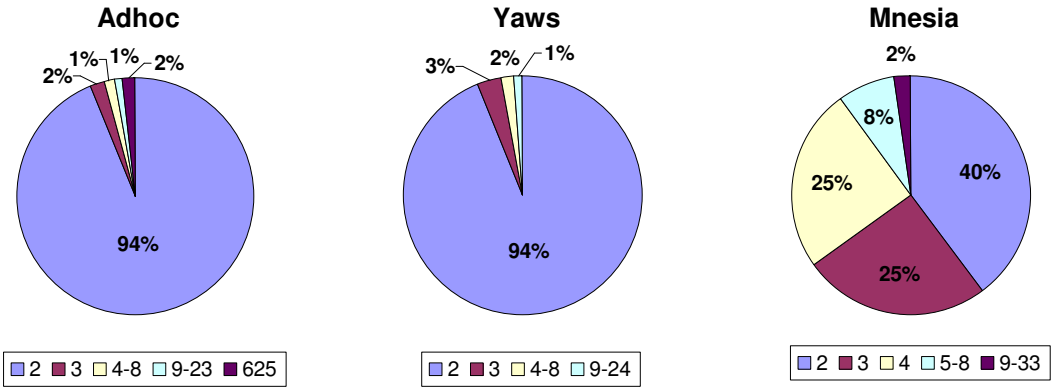


Fig. 2. Sizes of heap objects for three big Erlang applications.

used in this article. We have also run a wide range of other Erlang programs and commercial applications we have access to, and have discovered that nearly three quarters (73%) of all heap-allocated objects are cons cells. Out of the remaining ones, fewer than 1% are larger than eight words. Although these numbers have to be taken with a grain of salt, since each application has its own memory allocation characteristics, it is quite safe to conclude that, in contrast to for example a Java implementation, there is a significant number of heap-allocated objects which are small in size and do not contain a header word. Adding an extra word to every object would significantly penalize execution and space consumption and is therefore not an option we consider as viable in our setting. How this constraint influences the design of the incremental garbage collector is discussed in Section 6.

4 Garbage Collection in the Hybrid Architecture

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

4.1 Generational garbage collection of process-local heaps

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

memory management model in mind. [3]

When a process runs out of its allocated memory, the runtime system garbage collects its heap using a generational, Cheney-style, stop-and-copy collector [7]. An interesting characteristic of this generational collector is that it does not require the use of a remembered set mechanism. This is because the Erlang/OTP process-local heaps are *unidirectional* (that is, pointers always point from new to old objects) and as a result the runtime system never creates any pointers from the old generation to the young one. Again, ensuring this property is possible since there are no destructive assignments in our setting. In the generational collector, data has to survive two garbage collections to be promoted to the old generation. Also when running native code instead of byte code, the collector is guided by *stack descriptors* (also known as *stack maps*) and the root set is further reduced by employing *generational stack scanning* [9], an optimization which reduces the cost of scanning the root set in the process-local stacks by reusing information from previous GC scans.

Although the generational collector cannot give any real-time guarantees, pause times when collecting process-local heaps are typically not a problem in practice. This is because most collections are minor and therefore quite fast, and as explained above many Erlang applications have been programmed to use processes for specific, fine-grained tasks that require a relatively small amount of memory. More importantly, because this runtime architecture has been designed with the ability to collect process-local heaps independently and without requiring any global synchronization in a multi-threaded implementation, pauses due to collecting process-local heaps do not jeopardize the responsiveness of the entire system as the mutator can service other processes which are in the ready queue.

4.2 Garbage collection of binaries via reference counting

The shared area for binaries is collected using *reference counting*. The count is stored in the header of binaries and increased whenever a new reference to a binary is created (for example, when a binary is sent as a message to another process). Each process maintains a *remembered list* of such pointers from the process-local heap to binaries stored in the binary area. When a process dies, the reference counts of binaries in this remembered list are decremented. A similar action happens for references which are removed from the remembered list as part of garbage collection. References from the message area are stored in a similar list handled by the garbage collector of the message area. Since cycles in binaries are not possible, cycle collection is not needed. Also, as binaries do not contain any pointers, deletion of a binary does not have cascading effects. In short, garbage collection of binaries is effectively real-time.

[3] In this respect, process-local heaps are very much like *arenas* used by the Apache Web server [22] to deallocate all the memory allocated by a Web script after its termination.

4.3 Generational process scanning collection of the message area

Since the message area is shared between processes, the root set is typically large and consists of both the stacks and the process-local heaps of all processes in the system. As a result, pause times for collecting the message area using a naïve, "stop-the-world" type of collector are quite high. This situation can be partly ameliorated as follows:

- By splitting the message area into generations and performing generational collection on this area. In fact, one can employ a *non-moving* collector (such as *mark-sweep*) for the old generation to avoid the cost of repeatedly having to copy long-lived objects. However, we still prefer to manage the young generation by a copying collector. This is partly in order to capitalize on the fact that most objects tend to die young. More importantly, because a copying collector results in faster allocation; the live data is compacted and allocation can take place by pointer bumping.
- By performing an optimization introduced in this article, which we will call *generational process scanning* because it is the natural extension of generational stack scanning [9] from the single- to the multiple-process setting. More specifically, similarly to how generational stack scanning tries to reduce the root set that has to be considered during a process-local GC to only the new part of the stack (that is, the part created in the period between two consecutive minor garbage collections), generational process scanning tries to reduce the number of processes whose memory areas are considered part of the root set. In implementation terms, the runtime system maintains information about processes which have been active communication-wise (that is, allocated in the message area or received a message) since the last garbage collection. These are the only processes that could hold new references to objects in the message area. These processes are put in a data structure, called the *dirty process set*, and are the only ones considered as part of the root set during the frequent minor collections. Otherwise, the collector of the message area would have to scan possibly many thousands of processes for references to the messaga area.

These techniques for collecting the message area are quite effective. However, they cannot of course provide any real-time guarantees — not even soft real-time ones — and cannot prevent collection of the message area becoming a bottleneck in highly concurrent applications. For the message area, we need a garbage collection method that is guaranteed to result in small pause times such as an incremental or a real-time one.

Note that reference counting is *not* the most appropriate real-time GC technique to employ in the message area, even though there are no cyclic data in messages. The main reason is that unless the compiler maintains precise information about points where variables containing pointers to messages become dead (which in

general requires an escape analysis that the variable is not returned as a result from a function to some other function), the only point when reference counts of messages can be decreased is when a process dies. Decreasing reference counts only when processes die is not an effective memory reclamation technique; consider for example server-like applications where some processes are long lived. Furthermore reference counting typically imposes a non-negligible overhead, especially in a multi-threaded setting where locking is required. A different method for real-time or incremental GC is called for. In the following sections, we describe the one we designed and implemented.

5 An Incremental Collector for the Message Area

Terminology and assumptions We will use the term *collection stage* to refer to a contiguous period of time during which incremental garbage collection takes place. The term *minor collection cycle* will be used to refer to a complete collection of the young generation and *major collection cycle* to refer to a complete collection of the entire message area (both its young and old generation). The collector of the message area that we present is incremental and is designed with the ability to run concurrently with a single mutator thread or even with a number of mutator threads which schedule Erlang processes. However, the collector itself is *not* concurrent. Even on a multi-threaded system, at most one collector thread is ever present and is a responsibility of the runtime system to ensure this invariant.

We will use the terms of the *tricolor abstraction* [16, Section 6.1], where objects are assigned one of three colors, white, gray or black. All objects are white (meaning unprocessed) at the beginning of the collection cycle. The collector will then change the colors of all objects it visits to gray and later to black (meaning partly and fully processed, respectively). All gray objects will be visited by the garbage collector at some point during the collection cycle, to mark them black. At the end of the collection cycle, no gray objects remain and all objects still white will be reclaimed.

Organization of the message area Figure 3 shows the organization of the message area during a collection cycle.

- The young generation is managed by a copying collector and consists of two equal-sized parts, the *nursery* and the *from-space*. The size of each part, Σ, is constant. The nursery is used by the mutator as the allocation area during a collection cycle. The from-space is used in the copying collection; the *to-space* is the old generation.
- The old generation, which is collected by a mark-sweep collector, consists of n pages in a linked list. Allocation uses a free-list and can take place using either

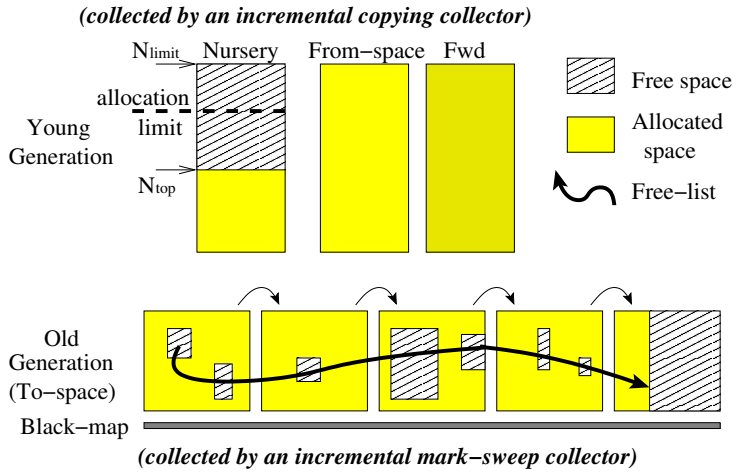


Fig. 3. Organization of the message area during a collection cycle.

an algorithm as simple as for example first-fit or by segregating the free-list into sublists for objects of different sizes.

- We also use an area of *forwarding pointers* (denoted as Fwd in Fig. 3). The reason is that we allow the mutator to access objects in the from-space between collection stages, that is, during a collection cycle. The mutator does not expect to find forwarding pointers in the place of objects, and therefore forwarding pointers cannot be stored in the from-space. The size of this area is limited by the size of the from-space (Σ).
- To mark an object in the old generation as live we use a bit array, called the *black-map*. This bit array is needed in our implementation since there is no room for any mark-bits in the actual objects. In other implementations, mark-bits could be placed in the object headers if space permits.
- Finally, we also use a pointer into the nursery (the *allocation limit*), whose purpose and usage we describe in Section 5.2.

Note that the collector does not require any header word in the objects in order to perform incremental copying collection in the young generation. Therefore, it imposes no overhead to allocation. The collector instead uses an extra space, namely the forwarding area, whose size is bounded by Σ. Recall that Σ is constant; it does not increase during garbage collection and is not affected by the allocation characteristics of the program which is being executed. In the old generation, the only extra overhead is one bit per word for the black-map. To keep track of gray objects we use a dynamically resizeable stack. Note that for the frequent collections of the young generation, the size of this gray stack is bounded by $\Sigma/2$ (since the size of the smallest heap-allocated object is two words). Overall, the space overhead of the incremental collector is quite low.

As with all incremental collectors, a crucial issue is to decide how and when the switch between the mutator and the collector will occur. We have investigated two different approaches to this interaction, one which is *work-based* and one which is

```
proc message_area_gc() ≡
  if gc_major = true →
      major_collection()
  else if gc_cycle = true →
      minor_collection()
  else
      gc_cycle := true
      swap(nursery, fromspace)  // Atomic operation; see Section 6.2
      clear_forwarding_pointers()
      if need_major = true →
          major_collection()
      else
          minor_collection()
  update_allocation_limit()
```

Fig. 4. The garbage collection algorithm for the message area.

time-based. In both approaches the collector is given a fixed (work-based or time-based) *quantum* to work in. Once this quantum expires, we calculate how much the mutator can be allowed to work before it is time for the next collection stage. This is discussed in detail in Section 5.2.

5.1 The algorithm

Figures 4–8 show the structure of the incremental collector. First, two Boolean flags are checked: gc_cycle indicates whether a collection cycle is in progress, while gc_major indicates whether the current collection cycle is a minor or a major one. The gc_major flag is set only if gc_cycle is also set, so it is enough to check gc_major to know if there is a major collection cycle in progress. Both flags are set to false in the end of the collection cycle (cf. Fig. 9), which means that if both are false in the beginning of *message_area_gc* a new collection cycle should be initiated.

A new collection cycle begins with setting gc_cycle. The from-space and the nursery switch roles and all forwarding pointers are reset. Then the $need\_major$ flag, which if needed has been set during the previous collection cycle (cf. Fig. 8), is checked to see if this is to be a minor or a major collection cycle and the appropriate action is taken.

5.1.1 Actions during minor garbage collection

A minor collection (cf. Fig. 5) starts by picking up the first process from the dirty process set and conceptually takes a *snapshot* of its root set. Because we do not want to take a physical copy of the root set snapshot, we simply record the val-

```
proc minor_collection() ≡
  foreach p ∈ Dirty_Process_Set do
    Roots := collect_roots(p)    // Atomic operation in the time-based collector
    foreach r ∈ Roots do
      if quantum_expired() = true → abort
      if points_to(r, fromspace) = true → forward(r)

    foreach g ∈ Gray do
      foreach reference field ref of g do
        if quantum_expired() = true → abort
        if points_to(ref, fromspace) = true → forward(ref)
      mark_black(g)                          // Remove g from the Gray stack
    remove(p, Dirty_Process_Set)

  foreach reference ref in the nursery do
    if quantum_expired() = true → abort
    if points_to(ref, fromspace) = true → forward(ref)

  foreach g ∈ Gray do
    foreach reference field ref in g do
      if quantum_expired() = true → abort
      if points_to(ref, fromspace) = true → forward(ref)
    mark_black(g)                            // Remove g from the Gray stack

  gc_cycle := false
```

Fig. 5. The incremental minor collection algorithm. All roots are traversed and live objects in the *fromspace* are copied to the old generation and marked as gray. Thereafter, all gray objects are traversed in a similar way to copy their children.

ues of the stack and process-local heap pointer and the pointers in the message queue. So taking a snapshot simply consists of recording a set of pointers and is typically a very fast operation. After recording this information, the mutator can continue allocating in the nursery and reading from from-space since there are no destructive updates. The snapshot is then scanned and when a live object is found in the from-space and this object has not yet been forwarded, the object is copied to the old generation and added to a stack of gray objects (cf. Fig. 7). Each time we copy an object, we update the original root references to point to the new location (*update_source_ref*) and store a forwarding pointer in the forward area to ensure objects are copied at most once (*set_forward_ptr*). If the object has been previously forwarded (i.e. if a forwarding pointer for it exists), we just update its reference in the root set.

When all the roots from the process are scanned, we pop the gray objects one by one and each object is scanned for references. If the popped object refers to an object in the from-space that has not already been forwarded, the newly found object is copied and pushed onto the gray stack. In the generational setting, an object is gray if it has been copied to the old generation but not yet scanned for references to other

```
proc copymark(ref) ≡
  if points_to(ref, fromspace) = true →
      forward(ref)
      mark_blackmap(ref)
  elsif points_to(ref, old_generation) = true →
      if gray_or_black(ref) = false →
          push_gray(ref)
          mark_blackmap(ref)
```

Fig. 6. The copymark procedure.

```
proc forward(object) ≡
  if is_forwarded(object) = true →
      update_source_ref(object)
  else
      destination := copy_object(object)
      push_gray(destination)
      update_source_ref(object)
      set_forward_ptr(object, destination)
```

Fig. 7. Procedure that forwards objects to the old generation. If the object is not forwarded already, it is copied to the old generation and pushed onto the gray stack.

```
funct copy_object(object) ≡
  if available_in_old() < sizeof(object) →
      allocate_new_page()
      if gc_major = false → need_major := true
  copy := copy_to_old(object)
  return copy
```

Fig. 8. Function to copy objects to the old generation. If allocation in the old generation fails the $need\_major$ flag is set. This ensures the next collection cycle will be a major one.

objects. An object is fully processed and becomes black when all its children are either black or gray.

Eventually the gray-stack is empty and the process is removed from the dirty process set and we pick the next one to process its root set. During a collection cycle, processes may become dirty again *only* by receiving a message allocated in the from-space. This effectively acts as a *write barrier*, albeit one with an extremely low cost; namely, it requires one extra test for each entire send operation rather than a test for each memory write.

At the end of a minor collection cycle we also have to look through the objects in the nursery to update references still pointing to the from-space (or possibly copy the referred objects) since the mutator can create references from objects in the nursery to objects in the from-space between collection stages. Because the nursery might contain references to objects not copied to the old generation yet, a final check that the gray stack is empty is needed at the end.

```
proc major_collection() ≡
  if need_major = true →
      Dirty_Process_Set := All_Processes
      clear_blackmap()
      gc_major := true
      need_major := false
      fetch_new_page_for_old()

  foreach p ∈ Dirty_Process_Set do
    Roots := collect_roots(p)          // Atomic operation in the time-based collector
    foreach r ∈ Roots do
      if quantum_expired() = true → abort
      copymark(r)

    foreach g ∈ Gray do
      foreach reference field ref of g do
        if quantum_expired() = true → abort
        copymark(ref)
      mark_black(g)                              // Remove g from the Gray stack
    remove(p, Dirty_Process_Set)

  foreach reference ref in the nursery do
    if quantum_expired() = true → abort
    copymark(ref)

  foreach g ∈ Gray do
    foreach reference field ref in g do
      if quantum_expired() = true → abort
      copymark(ref)
    mark_black(g)                                // Remove g from the Gray stack

  sweep()        // Traverse the black-map and build a free-list of the unmarked areas

  gc_major := false
  gc_cycle := false
```

Fig. 9. The incremental major collection algorithm. First, all processes are considered dirty. Thereafter the collection for the old generation follows.

5.1.2 Actions during major garbage collection

If during collection of the young generation the old generation overflows, a flag ($need\_major$) is set so that the next garbage collection cycle will be a major one (cf. Fig. 8). A new page is linked to the old generation and added to the free-list to allow the copying garbage collector of the young generation to finish its current minor collection cycle.

The major garbage collector (cf. Fig. 9) is a combination of a copying collector

and a mark-sweep collector. The copying part of the collector is the same as in the minor collection and copies objects from the young to the old generation, linking in new pages to the old generation if needed. The old generation is collected by a mark-sweep collector.

In the beginning of a major collection cycle, we set the gc_major flag to let the $message\_area\_gc$ procedure know this is a major collection and clear the black-map. (The mark-bits in the black-map could also be cleared in the sweep-phase.)

The major collection cycle then proceeds with the collector fetching the roots from the processes. As in the minor collection, this is done by taking a snapshot of the root set one process at the time. While scanning the root set, reachable objects in the young generation that are not already marked are copied to the old generation, pushed onto the gray-stack and immediately marked as live. Reachable objects in the old generation get the same treatment except that they are not copied.

When all the roots from one process have been scanned we proceed to pop the gray objects as in the minor collection. If the popped object refers to an object that has not already been marked as live, the referred object is copied if it resides in the young generation, pushed onto the gray stack and marked.

Because the mutator (which allocates in the nursery) might in the meantime have created pointers to the message area we also scan the nursery. When all dirty processes have been processed and we are out of gray objects, we proceed to the sweep phase. In the sweep phase we simply scan the black-map and build a new free-list from the unmarked areas. If an entire page turns out to be free we release that page.

5.2 Interplay between the mutator and the collector

During a collection cycle, the collector might exceed its allowable (time or work based) quantum and be forced to abort collection and yield to the mutator. As shown in Fig. 5 and 9 this happens in the beginning of for-loops. The **abort** statement, which can be thought as a **return**, will return control to the main collection procedure (*message_area_gc*) at the point where we call the *update_allocation_limit* procedure. This procedure calculates how much the mutator is allowed to work.

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

To control how much the mutator should be allowed to work, we use an *allocation limit* (cf. Fig. 3). When the mutator reaches this limit the collector is invoked. This is a cheap way to control the interleaving and furthermore imposes no additional overhead on the mutator. This is because, even in a non-incremental environment, the mutator checks against a limit anyway (the end of the nursery, N_{limit}). The allocation limit is updated using a calculated estimate, depending on the type of the collector, as described in the following two subsections.

5.2.1 Update of the allocation limit in the work-based incremental collector

The underlying idea of how to update the allocation limit in the work-based collector is simple. Consider the case of the collecting the young generation, which is managed by a copying collector. If, during a collection stage, the mutator allocates w_M words of heap and the collector rescues from the from-space w words of live data (where by rescue in this case we mean copy to the old generation), then for correctness it must of course be the case that $w_M \leq w$. After each collection stage the allocation limit is therefore updated to $N_{top} + w$, where N_{top} denotes the top of the nursery (that is, its first free word; cf. Fig. 3). Note that this calculation is exact, rather than an estimate as in the case of the time-based collector of the following section.

In a minor collection the area we collect, the from-space, has the same size as the nursery and we can therefore guarantee that the collection cycle will end before the nursery is exhausted by mutator allocations. In fact, since this is a young generation and most of newly allocated data tends to die young, the collection cycle is often able to collect the from-space before a significant amount of allocation takes place in the nursery.

In a major collection we collect the entire message area. However, the size of the nursery, where the mutator allocates during a collection cycle, remains constant (Σ). This means that in order for the mutator to be allowed to allocate w words, we have to process not only w words of live data as in the minor collection, but a fraction of the entire message area. We denote this fraction by w_{GC}.

We formulate an equation that says that the fraction of work done in relation to the entire message area should be the same or greater than the fraction w/Σ. In this equation we have taken double the size of the old generation. This is to guarantee that the sweep-phase is covered as well.

$$\frac{w_{GC}}{2 \mid old \mid +\Sigma} \geq \frac{w}{\Sigma}$$

By extracting w_{GC}, we now get a formula to use when calculating how much work needs to be done to allow the mutator to allocate w words.

$$w_{GC} \geq w \left(\frac{2 \mid old \mid}{\Sigma} + 1 \right)$$

After a major collection stage N_{top} is updated with w as in the minor collection. During the major collection, work is calculated when marking objects in the old generation (recall that objects are also marked when copied) and in the sweep-phase.

5.2.2 Update of the allocation limit in the time-based incremental collector

In the time-based collector, the *collector time quantum*, denoted t, determines the time interval of each collection stage. After this quantum expires, the collector is interrupted and the mutator is resumed.

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

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

For simplicity, let's consider a minor collection. Assuming the worst case scenario, namely that the entire from-space of size Σ is live, at the end of a collection stage we conservatively estimate what fraction of the total collection we managed to do. Then, again conservatively, we estimate how many more collection stages it will take to complete the collection cycle, provided we are able to continue rescuing live data at the same rate.

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

We now get:

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

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

In a major collection, we use the same formula but instead of Σ we use the size of the entire message area.

5.3 Restarting a collection and termination

All collections start by calling *message_area_gc*. If we are in the middle of a collection cycle, the gc_cycle flag indicates this, and the collection proceeds by picking up the next dirty process. If the mutator has in the meantime added new processes to the dirty process set, these processes will be considered. Note that this does not affect termination since the mutator always allocates in the nursery and these processes can not contain any pointers to objects in the from-space (or the old generation) that were not live at the beginning of the garbage collection cycle.

In the work-based collector, we are guaranteed that the collection cycle will finish before the nursery overflows. This is because our update of the allocation limit is conservative. In the time-based collector, we have no such guarantee even though we try to make such a conservative estimate, and a mechanism to handle overflow (e.g., allocate a new nursery or extend the collection cycle) is required.

Note that the work-based collector can be interrupted while collecting the roots of a process (its stack and PCB). The next collection stage will start over with the abandoned process and since we measure work in number of words rescued and we never rescue the same object twice, we can guarantee that we will finish with the process in due time. In the time-based collector there is no such guarantee. With a sufficiently small t and a process with a large enough stack or mailbox, the time-based collector could start over and over again on the same process without making progress.

6 Our Implementation

We have implemented the incremental garbage collection algorithm of the previous section in the Erlang/OTP system. In this section we describe a few design choices together with a number of effective optimizations.

6.1 Design choices

Sizes of different areas We chose to implement the forwarding area as an array of constant size, currently the same size as the from-space. Note that the incremental collection algorithm makes no assumptions about the size of pages and different pages can have different sizes. Pages can be allocated from the operating system upon need, but in order to reduce the number of system calls we preallocate an array of pages (each page being $32Kwords$ in size) in our implementation. The pages not used by the old generation are kept in a linked list on the side.

User interaction The value of w in the work-based collector is user-specified. However, regardless of the user setting, we ensure that in all collection stages w is at least as big as the mutator need w_M ($w_M \leq w$). The choice of w naturally affects the pause times of the collector as described in Section 7.2. In the time-based collector, the value of t is also specified (in $\mu secs$) by the user based on the application's demands.

Locking the PCB In a single-threaded implementation, when we take a snapshot of a process, no work is being done concurrently. This means that the process which is collected need not be interrupted or locked to have its snapshot taken. In a multi-threaded implementation, locks or write-barriers on the PCB are required, but still a process does not need to be suspended to have its snapshot taken. This is because all heap data in Erlang are immutable. Therefore, a process can allocate on its process-local heap at the same time as the collector rescues objects from it. The points where locking is required is when recoding the snapshot and when a process is removed from the dirty process set. At the latter point, the PCB of the process must be locked to ensure that the process does not become dirty (by a message send) at the same time when it is taken off the dirty process set.

Forwarding pointers We chose to store forwarding pointers in a separate area to allow the mutator to access objects in the from-space in between collection stages. Most copying collectors choose another approach, namely to store the forwarding pointers in the from-space. In some implementations this is done with an extra header word, others overwrite the old copy with the forwarding pointer. (An extra header word is not an option in our case as explained previously.) Storing forwarding pointers in the place of objects would also be possible. However, it would require a change in the mutator either to perform a test on each heap pointer dereference and pay the corresponding cost, or to systematically use Brooks-style *indirection* [4] and employ a *read barrier* mechanism to make sure that the mutator never sees objects in the from-space (as for example in the Metronome [3]), a mechanism which also has a non-trivial associated cost.

6.2 Atomic operations in the incremental collector

Even though our implementation of the incremental garbage collector respects its work- or time-quanta during the majority of the collection cycle, there are a few phases where the collector can not be interrupted. We call these parts of the collection cycle *atomic operations*. Our atomic operations are:

- Swapping the nursery and the from-space. This is because, in a multi-threaded setting, the mutator may not be allocating concurrently with this operation.

- Collecting the root set of a process in the time-based collector. This is because we want to guarantee that we will never get caught in an endless loop while considering the root set of a process. [4] In practice, collecting the root set from the stack and the PCB is very fast (excluding the mailbox, the PCB has a constant size) and the fact that this operation is atomic does not jeopardize the real-time characteristics of our collector.
- Setting up and cleaning up auxiliary areas for the collection. As seen in Fig. 5 and 9 checking if the quantum has expired occurs in the beginning of each loop. This means that all sequential pieces of code outside the loops are effectively atomic operations. This includes setting, checking, and clearing flags and bitmaps before and after collection, calculating the allocation limit, and some logging enforced by the runtime system. (In some cases this may result in a violation of the time quantum specified for the collector, although this extension is typically very small; cf. Section 7.2.)

In our implementation, two phases of the collector have not been made incremental yet: 1) updating reference counts of binaries and collecting those whose count drops to zero and 2) the sweep-phase. Making these phases incremental is straightforward. Recall that there is no need for an atomic sweep-phase since the mutator never allocates in the old generation.

6.3 Some optimizations

In the beginning of the major collection cycle, all processes in the system are put in the dirty process set, in a more or less random order. [5] However, each time a process receives a message, it is conceptually moved last in the set (as if it were reborn). This way, we keep the busiest processes last and scan them as late as possible and the dirty process set is effectively used as a queue. The rationale for wanting to postpone their processing is three-fold:

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

Another technique we use to postpone the processing of members of the dirty process set is to process the stack of gray objects after we are finished with each process (instead of processing all processes in the dirty process set first and then processing the complete gray stack).

[4] In the work-based collector, there is no such risk; see end of Section 5.3.
[5] The order is actually determined by the age of the processes, oldest first.

By incrementally taking partial snapshots of the root set, i.e., only one process at a time, we allow the remaining processes to create more garbage as we collect the process at the head of the set. This means that we will most likely collect more garbage than if a complete snapshot was taken at the beginning of the collection cycle.

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

When the collector rescues objects from the young generation to the old, it uses the free-list. But since a new page is allocated at the beginning of a major collection, we can cheaply allocate in this page by pointer bumping during the collection.

Finally, a very important optimization is to have process-local garbage collections record pointers into the message area in a remembered set. Note that it is not so common to have the collector rather than the mutator build the remembered set. This way we avoid scanning the old generation of their local heaps. An unfortunate side-effect of this optimization is that since we do not actually scan the old generation of process-local heaps during root scanning, but only the remembered sets, some of the rescued objects might be already dead at the start of the collection. An object may therefore be kept in the message area for a number of collection cycles until a major process-local garbage collection updates the remembered set of objects (or the process dies) and the next collection cycle of the message area is triggered to finally remove the object. This however is a drawback inherent to all generational schemes.

7 Measurements

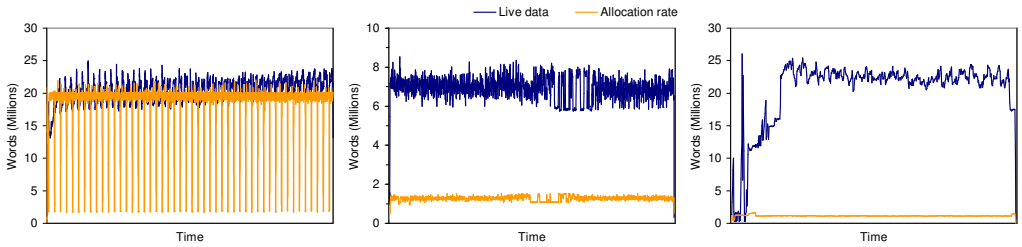
For the performance evaluation we used synthetic benchmarks (the first two below) and three applications with a high degree of concurrency from different domains:

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

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

Table 1
Concurrency characteristics of the benchmarks.

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



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

Fig. 10. Allocation rates and live data for the three applications.

adhoc A framework for algorithms for genetics. The specific benchmark simulates a population of chromosomes using processes and applies crossovers and mutations. The code of AdHOC [6] consists of about 8,000 lines of code.

yaws A high-performance multi-threaded HTTP Web server where each client is handled by a separate Erlang process. Yaws [7] contains about 4,000 lines of code (excluding calls to functions in Erlang/OTP libraries such as HTTP, SSL, etc). We used `httperf` [17] to generate requests for Yaws.

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

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

Figure 10 shows the allocation rates and volume of live data during the runtime of the non-synthetic benchmarks. The allocation rate unit is words allocated per second. Note that the scales on the y-axis differ between figures. It is clear that these applications are allocation intensive and their memory footprint is quite large, especially considering the sizes of heap objects (Fig. 2). **adhoc** allocates on average 20 million words per second and has a footprint of about the same size. This means that 20 million words of garbage are produced per second. The allocation rate and footprint of **yaws** is somewhat more modest, but throughout the program more than

[6] ADHOC: Adaptation of Hyper Objects for Classification.
[7] YAWS : Yet Another Web Server; see `yaws.hyber.org`.

Table 2
Number of GCs when using the work-based incremental collector.

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

Table 3
Mutator times, total GC times and pause times using the non-incremental collector.

Benchmark	Total time (ms)			Local GC (μs)			Message area GC (μs)		
	Mutator	Local GC	MA GC	Max	Mean	G.mean	Max	Mean	G.mean
worker	3,591	2,756	1,146	7,673	395	68	178,916	89,811	77,634
msort_q	174	3	29	577	9	4	16,263	9,807	11,646
adhoc	61,578	7,848	27	88	6	7	1,650	1,242	1,174
yaws	240,985	11,359	153	370	8	7	1,088	649	636
mnesia	53,276	4,487	88	4,722	4	5	1,413	485	458

200 million words are allocated, and the maximum live data at any time is about 8.7 million words, so we can safely say that the garbage collector has a fair amount of work to do here as well.

The performance evaluation was conducted on an Intel Xeon 2.4 GHz machine with 1 GB of RAM and 512 KB of cache, running Linux. The 2.6.10 kernel has been enhanced with the `perfctr` driver [19], which provides access to high-resolution performance monitoring counters on Linux and allows us to measure GC pause times in μs.

7.1 Runtime and collector performance

To provide a base line for our measurements, Table 3 shows time spent in the mutator, garbage collection times, and GC pause times for all benchmarks when using the non-incremental collector for the message area. Observe that the first three columns of the table are in ms while the remaining ones are in μs. Table 4 confirms that the time spent in the mutator and in performing garbage collection of process-local heaps is effectively not affected by using the incremental collector for the message area. Depending on the configuration, the overhead for the in-

Table 4
Mutator times and total GC times (in ms) using the work-based incremental collector.

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

cremental collector of the message area compared to the non-incremental collector ranges from a few percent to 2.5–3 times for most programs. The overhead is higher (5.6 times) for **worker** which is a program that was constructed to spend a significant part of its time allocating in (and garbage collecting) the message area.

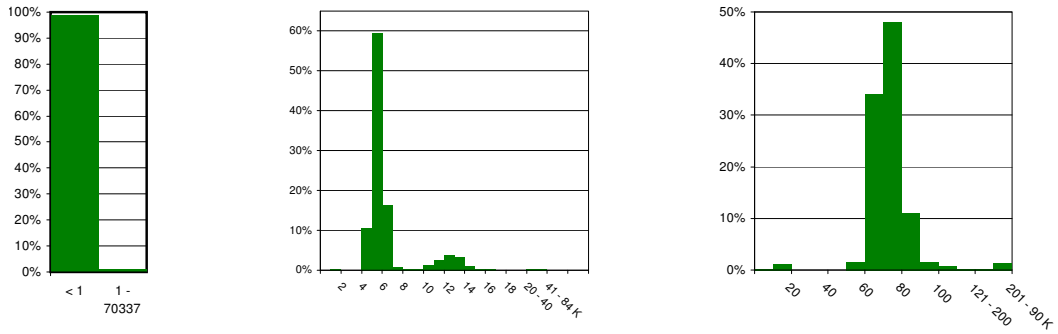
We have chosen $\Sigma = 100K\ words$, a deliberately small value to stress the incremental collector. A larger value of Σ is likely to result in less time spent in garbage collection since more data will become garbage between collections.

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

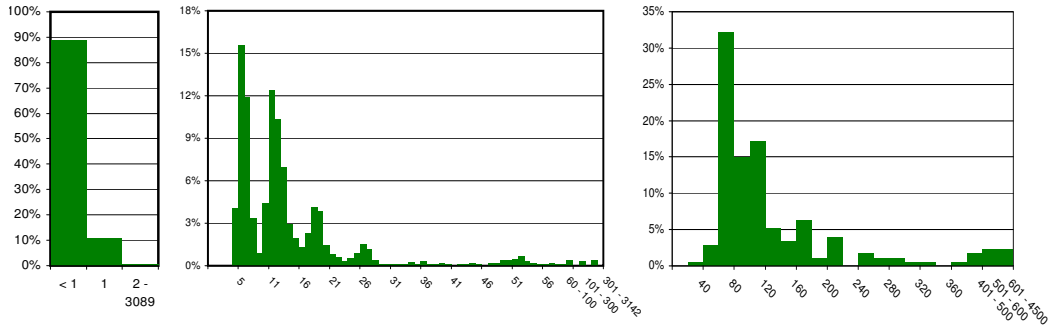
7.2 Garbage collection pause times

Table 5 shows pause times for the incremental work-based collector using three different choices of w, collecting $2, 100$, and 1000 words, respectively. As expected, for most benchmarks, the incremental garbage collector significantly lowers GC pause times, both their maximum and mean values (the columns titled G.mean show the geometric mean of all pause times) compared with the non-incremental collector (cf. the last three columns of Table 3). The maximum pause times of **yaws** (for $w = 100$ and 1000) are the only slight exception to this rule, and the only explanation we can offer for this behavior is that perhaps live message data is hard to come by in this benchmark. The mean GC pause time values, in particular the geometric means, show a more consistent behavior. In fact, one can see a correlation between the value of w and the order of pause times in μs.

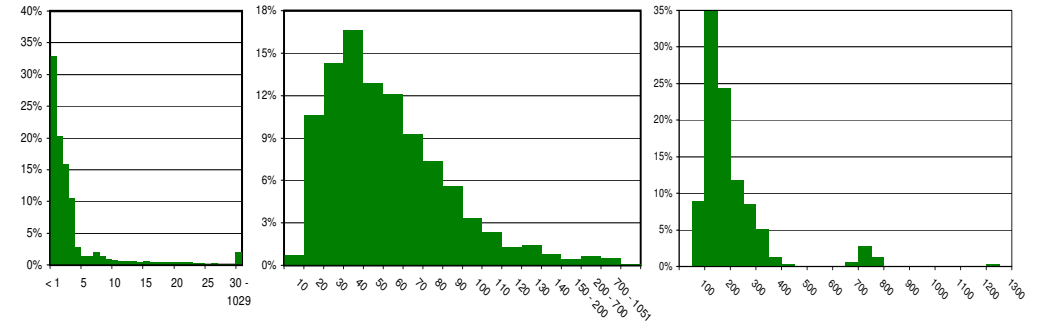
The distribution of pause times (in μs) for the benchmarks using the work-based



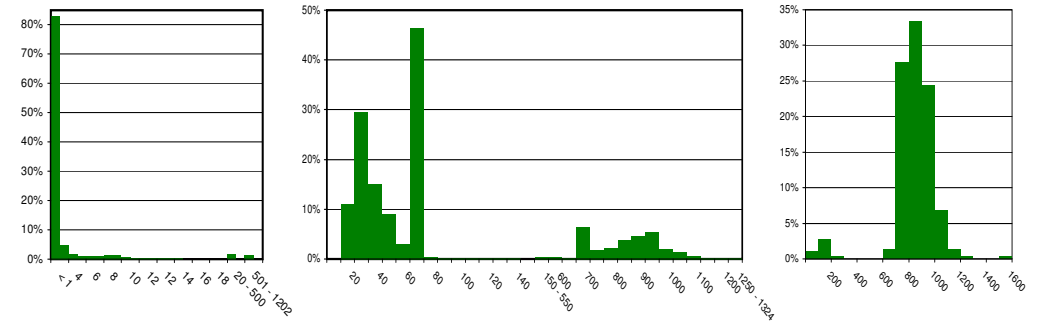
(a) **worker** ($w = 2$, $w = 100$, and $w = 1000$)



(b) **msort_q** ($w = 2$, $w = 100$, and $w = 1000$)



(c) **adhoc** ($w = 2$, $w = 100$, and $w = 1000$)

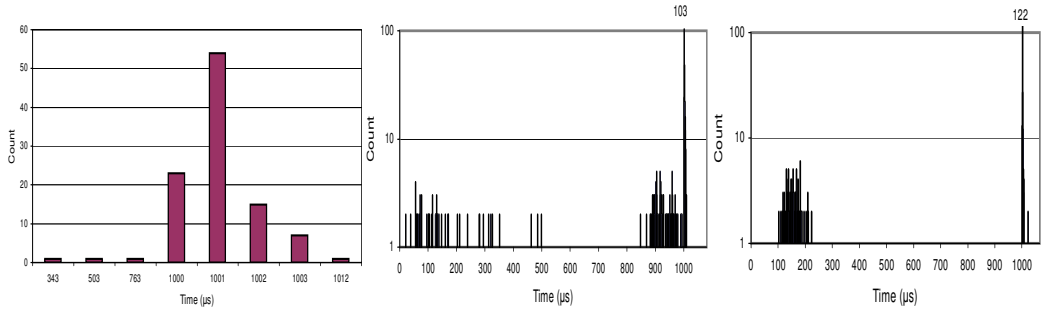


(d) **yaws** ($w = 2$, $w = 100$, and $w = 1000$)

Fig. 11. Distribution of pause times (in μs) for the work-based incremental collector.

Table 5
Pause times (in μs) for the incremental (work-based) collector.

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

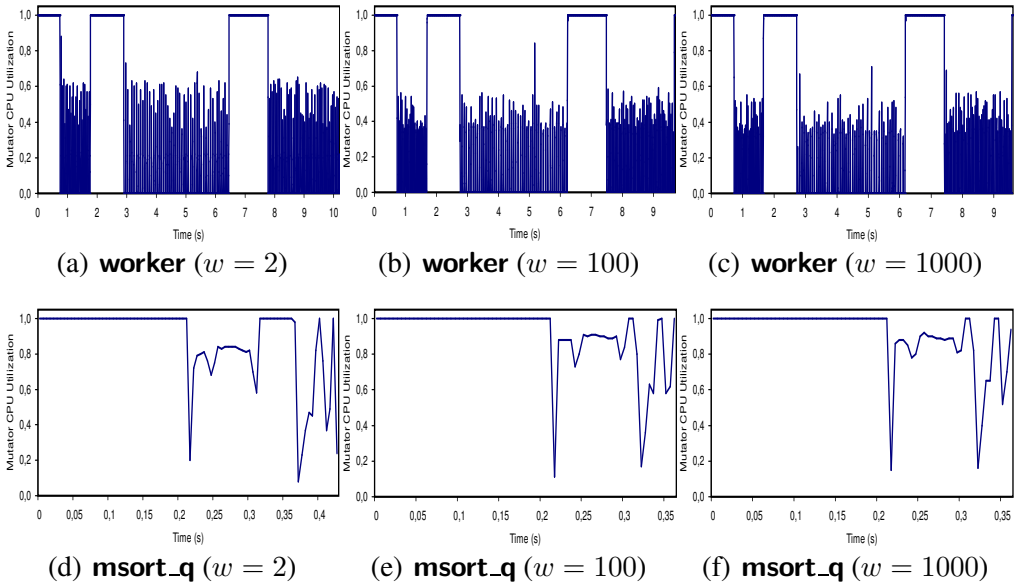


(a) **msort_q** ($t = 1000\mu s$) (b) **yaws** ($t = 1000\mu s$) (c) **mnesia** ($t = 1000\mu s$)

Fig. 12. Counts of pause times (in μs) for the time-based incremental collector.

incremental collector is shown in Fig. 11. [8] The majority of collection stages are very fast, and only a very small fraction of the collections might be a problem for a real-time application. On the other hand, a work-based collector whose notion of work is defined in terms of "words rescued" naturally cannot guarantee an upper limit on pause times, as live data to rescue might be hard to come by.

[8] **mnesia** is not included in Fig. 11 as its pause times do not show anything interesting.



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

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

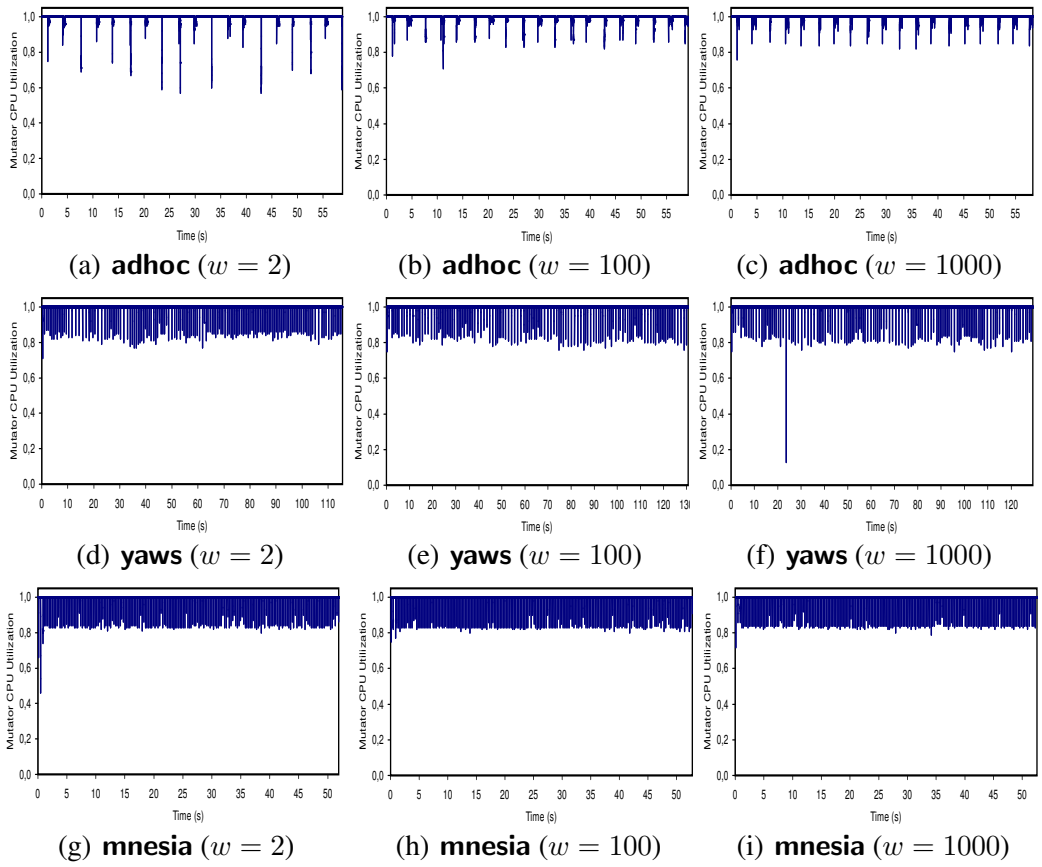
Fig. 13. Mutator utilization of the work-based collector on synthetic benchmarks.

A time-based incremental collector can in principle avoid this problem [3]. Care of course must be taken to detect the case when the mutator is allocating faster than the collector can reclaim, and to take some appropriate action. Figure 12 (cf. also Table 2) shows counts of GC pauses when running three of the benchmarks programs using the time-based incremental garbage collector with a t value of $1ms$ ($1000\mu s$). As mentioned in Section 6.2, when needed, the collector is allowed some small deadline extension, in order to possibly clean up after itself. This explains why there is a small number of values above $1000\mu s$. Note that in Fig. 12(b) and 12(c) the number of garbage collections (the y-axis) is in logarithmic scale.

7.3 Mutator utilization

The notion of *mutator utilization*, introduced by Cheng and Blelloch [8], is defined as the fraction of time that the mutator executes in any time window.

Figures 13 and 14 show mutator utilization for the programs we used as benchmarks when using the work-based incremental collector for different values of w. The two synthetic benchmarks exhibit interesting patterns of utilization. As expected, the **worker** benchmark suffers from poor mutator utilization since it is designed to be allocation-demanding and be a serious challenge for the incremental collector. The first interval of high utilization is the time before the first collection is triggered and the remaining two are periods after a collection cycle has finished and there is free space left in the nursery that the mutator can use for its allocation needs. Similarly, the mutator utilization of **msort_q** drops significantly when live data in the message area is hard to come by since each collection stage will take more time, but the mutator's work is still limited by the same w. On the other hand, the mutator



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

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

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

Fig. 14. Mutator utilization of the work-based collector on real-world programs.

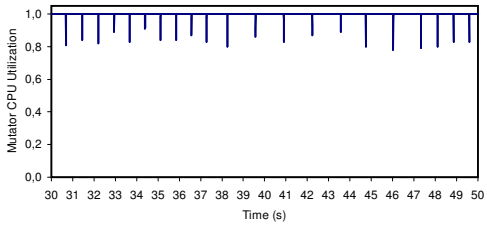
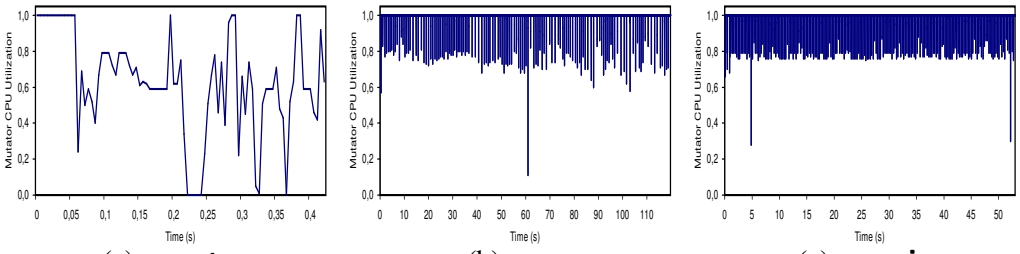


Fig. 15. Mutator utilization of **yaws** ($w = 100$) shown in detail.

utilization of the three real applications is good — even for $w = 2$, although for **yaws** and **mnesia** this is apparent only with the time axis stretched out. Figure 15 shows the same data as Fig. 14(e) but only for a portion of the total time needed to run the benchmark.

Mutator utilization for the time-based incremental collector is shown in Fig. 16. For both **yaws** (mainly) and **mnesia** the utilization using the time-based collector is slightly worse than that when using the work-based one, but quite satisfactory nevertheless. The mutator utilization for the synthetic **msort_q** program is not so good. But in this case, this is partly explained by the fact that although the t value is quite small ($1ms$), it is significant compared with the total execution time of the benchmark which is less than half a second.



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

Fig. 16. Mutator utilization for the time-based ($t = 1000\mu s$) incremental collector.

8 Related Work

Runtime system organization In the context of Java, several authors have suggested detecting thread-local objects via *escape analysis*. The goal of escape analysis has been to identify, conservatively and at compile time, objects that can only be accessed by their creating thread and allocate them on the thread-local stack, thereby avoiding the cost of synchronization for these objects. In an early such work, Steensgaard [21] exploits the escape analysis of Ruf [20] and suggests the use of thread-local heap chunks for non-escaping objects (which form the young generations) and a shared old generation for all other data. However, because all static fields are roots for all memory areas, a global *rendezvous* is required even for thread-local collections. In contrast, due to the pointer direction invariant that we maintain in our system, all process-local collections can happen independently.

Thread-local heaps for Java, guided by information gathered by profiling, have also been advocated by Domani *et al.* [11]. However, in a Java runtime system pointers can be overwritten and hence pointers can be created between any two areas in the system (for example, between different thread-local heaps). Thus, a write barrier mechanism is needed to trap such pointers which in turn can impose an unbounded amount of work during garbage collection. A runtime system organization that avoids this problem was proposed by Jones and King [15] while this article was under reviewing. Their runtime system organization is guided by a compile-time escape analysis that accommodates dynamic class loading and is similar to that of this article not only because it is structured around thread-local heaps and a shared memory area, but also because it maintains pointer directionality invariants which allow for thread-local collections to occur independently and without the use of a write barrier mechanism that does unbounded work.

In the functional world, the closest relative of our work is the memory architecture proposed by Doligez and Leroy [10] which uses thread-local allocation for immutable objects in Caml programs. Because thread-local heaps contain only immutable objects, local garbage collections can be performed independently.

Memory management of Erlang programs The soft real-time concerns of the Erlang language call for bounded time GC techniques. One such technique, based on a mark-sweep algorithm that takes advantage of the fact that the heap in an Erlang system is *unidirectional* (i.e., is arranged so that the pointers point in only one direction), was suggested by Armstrong and Virding [1]. However, their technique imposes a significant overhead and was never fully implemented. Similarly, Larose and Feeley designed a near real-time compacting collector [12] in the context of their Gambit-C Scheme compiler. This garbage collector was intended to be used in the Etos (Erlang to Scheme) system but never made it to an Etos distribution.

Incremental and real-time GC techniques In the context of other (concurrent) functional language implementations, the challenge has been to achieve low GC latency without paying the full price in performance that a guaranteed real-time garbage collector usually requires. Notable among techniques used in concurrent functional languages are the quasi real-time collector of Concurrent Caml Light [10] which combines a fast copying collector for the thread-specific young generations with a non-disruptive concurrent mark-sweep collector for the old generation (which is shared among all threads), and the recent work towards an incremental generational collector for Haskell [6] which uses various optimizations to eliminate the (cost of the) write barrier.

Many concurrent garbage collectors for strict functional languages have also been proposed, either based on incremental copying [4,13], or on *replication* [18] (see also reference [8] for a multi-processor version of one such collector). The main difference between them is that incremental collectors based on copying require a read barrier, while collectors based on replication do not. Instead, they capitalize on the copying semantics of (pure) functional programs, and incrementally replicate all accessible objects using a mutation log to bring the replicas up-to-date with changes made by the mutator.

Because of the copying semantics of message passing, our collector is not only free to replicate objects, but is also allowed to let the mutator access objects both in the from- and in the to-space. Thus, in contrast to typical replication collectors, we are not only free from having to preserve the invariant that the mutator *only* accesses objects in the from-space (or having to keep a mutation log), but we can also avoid having to update the root set at the end of the collection as an atomic step. By capitalizing on properties such as these, our incremental collector is able to achieve significantly shorter pause times than those currently reported in the literature.

Similar to our case, Nettles and O'Toole [18] also recognized the problem of adding an extra word to small objects to store the forwarding pointer and using indirection for all objects (despite the fact that their SML/NJ implementation already had a header word in all heap objects that could be overwritten with a forwarding pointer

— but indirection was only rarely used.) As they report in [18, Section 3.2], not being able to merge the header with the forwarding pointer has a prohibitive cost. Our collector does not require a header word in objects and its additional space requirement is constant and limited to the area of forwarding pointers in the young generation (cf. Fig. 3).

An excellent discussion and theoretical analysis of the trade-offs between work-based and time-based incremental collectors appears in a recent paper by Bacon, Cheng, and Rajan [3] describing the Metronome, a real-time garbage collector for Java. Given the different semantics (copying vs. sharing) of concurrency in Erlang and Java, and the different compiler and runtime system implementation technologies involved in Erlang/OTP and in Jikes RVM, it is very difficult to do a fair comparison between the Metronome and our incremental collector. As a rather philosophical difference, we do not ask the user to guide the incremental collector by specifying the maximum amount of simultaneously live data or the peak allocation rate over the time interval of a garbage collection. [9] More importantly, it appears that our system is able to achieve significantly shorter pause times and better mutator utilization than the Metronome. We believe this can mostly be attributed to its memory allocation strategy which is local-by-default. On the other hand, the utilization of our collector is not as consistent as that of the Metronome for adversarial synthetic programs (e.g., the **worker** program of Section 7), but then again we interleave the collector and the mutator in a much finer-grained manner (for example, collecting just 2 words) or we force our collector to run in a considerably smaller collector time quantum ($1ms$ or less vs. $22.2ms$ which the performance section of the Metronome paper [3] uses).

9 Concluding Remarks

We have presented the details of a complete memory management scheme for the implementation of concurrent programming languages using message passing. Besides being able to exploit the absence of mutable objects in our setting, there are also other reasons why our experimental results are so encouraging. Chief among them is the fact that we have attacked memory management from multiple rather than a single direction. We devised a runtime system architecture that is tailored to the intended use of data, designed a flexible message analysis that can guide its memory allocator, and implemented a carefully tuned incremental and generational garbage collector for its shared data area.

By partitioning allocation into process-local heaps, we are not only able to achieve process isolation, but also a guarantee that garbage collection of process-local areas can run concurrently with the rest of the system, without jeopardizing its global re-

[9] Work towards lifting this limitation is underway (private communication, Oct. 2004).

sponsiveness in highly concurrent applications. By devising an incremental garbage collector for the global data area, which can be scheduled in a very fine-grained manner, we significantly avoid having a point that can become a global bottleneck. In short, we see our work as the enabling technology for a scalable, high-performance implementation on top of a multi-threaded or distributed shared memory implementation. We are currently working towards such an implementation.

On a wider perspective, we believe that concurrency via message passing is fundamentally superior over the presently more common shared data structure approach: among its advantages are isolation, scalability, and transparent distribution. [10] Independently of whether message passing will eventually dominate, our work can be of benefit even for concurrent languages with sharing semantics when the immutability of certain objects can be established by other means (for example, via the type system or by using static analysis).

Acknowledgements

We sincerely thank Richard Jones, Marc Shapiro, and an anonymous reviewer of this special issue for requests for clarification, constructive criticism, and numerous detailed suggestions for changes which have significantly improved the presentation of our work.

References

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

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

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

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

[10] It is probably not by chance that message passing often gets reinvented as "the next best thing" or shows up disguised in buzzwords such as RMI, SOAP, and XML-RPC.

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

[6] A. M. Cheadle, A. J. Field, S. Marlow, S. L. Peyton Jones, and R. L. White. Exploring the barrier to entry - incremental generational garbage collection for Haskell. In A. Diwan, editor, *Proceedings of the Fourth ACM SIGPLAN International Symposium on Memory Management*, pages 163–174, New York, N.Y., Oct. 2004. ACM Press.

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

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

[9] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'98*, pages 162–173, New York, N.Y., 1998. ACM Press.

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

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

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

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

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

[15] R. Jones and A. C. King. A fast analysis for thread-local garbage collection with dynamic class loading. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005)*, pages 129–138, Los Alamitos, CA, Sept./Oct. 2005. IEEE Computer Society Press.

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

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

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

[19] M. Pettersson. Linux performance-monitoring counters kernel extension. Available from: http://user.it.uu.se/∼mikpe/linux/perfctr/.

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

[21] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In T. Hosking, editor, *Proceedings of ISMM'2000: ACM SIGPLAN International Symposium on Memory Management*, pages 18–24, New York, N.Y., Oct. 2000. ACM Press.

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