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

Konstantinos Sagonas
Dept. of Information Technology
Uppsala University, Sweden
kostis@it.uu.se

Jesper Wilhelmsson
Dept. of Information Technology
Uppsala University, Sweden
jesperw@it.uu.se

ABSTRACT

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

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.4 [**Programming Languages**]: Processors—*memory management (garbage collection), run-time environments*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages, concurrent and distributed languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures, dynamic storage management*

General Terms

Languages, Performance, Measurement, Experimentation

Keywords

Incremental and real-time garbage collection, thread-local heaps, concurrent languages, Erlang

ISMM'04, October 24–25, 2004, Vancouver, British Columbia, Canada.

1. INTRODUCTION

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

Taking up the latter challenge becomes tricky when automatic memory management is performed using garbage collection (GC). The naïve "stop-the-world" approach, where threads repeatedly interrupt execution of 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: it introduces a point of global synchronization between otherwise independent threads—and possibly also tasks—and provides no guarantees for bounds on the length of the individual pauses or for sufficient progress by the application; see [8] for a discussion of the issues that are involved.

Despite the significant progress in developing automatic memory reclamation techniques with real-time characteristics (see e.g., [3, 5, 8, 18, 20]), each technique relies on a number of (often implicit) assumptions about the architecture of the runtime system that might not be the most appropriate ones to follow in a different context. Furthermore, languages have their own characteristics which influence the trade-offs associated with each technique. For example, many collectors for object-oriented languages such as Java assume that allocating an extra header word for each object does not penalize execution times by much and does not impose a significant space overhead. Similarly, the semantics of a language may favor the use of a read rather than a write barrier, or may allow for more liberal forms of incremental collection (e.g., based on replication of objects). Finally, it is clear that the type of GC which is employed interacts with and is influenced by the allocation which is used. It is very difficult to come up with techniques that are well-suited for all runtime environments.

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

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

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

2. THE CONTEXT OF OUR WORK

The work reported in this paper is part of an ongoing research project at Uppsala University in collaboration with the development team of the Erlang/OTP system at Ericsson. Prior work has resulted in the development of the HiPE (High Performance Erlang) native code compiler [16], which nowadays is fully integrated in the Erlang/OTP system, and in investigation of the pros and cons of alternative runtime system architectures for concurrent languages using message passing (work reported in [17] and briefly reviewed in Sect. 2.2). Chief among the current goals of the project are to implement static analyses which determine the intended use of data in highly concurrent languages in order to guide the memory allocator, and to improve the responsiveness of the resulting system by incorporating garbage collectors with real-time characteristics and a high rate of mutator utilization.

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

2.1 Erlang and Erlang/OTP

ERLANG [2] is a strict, dynamically typed functional programming language with support for concurrency, communication, distribution and fault-tolerance. It has automatic memory management and supports multiple platforms. ERLANG was designed to ease the programming of soft real-time control systems commonly developed by the data- and tele-communications industry. Its implementation, the Erlang/OTP system, has so far been used quite successfully both by Ericsson and by other companies around the world (e.g., T-Mobile, Nortel Networks) to develop large (several hundred thousand lines of code) commercial applications.

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

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

ERLANG is often used in high-availability large-scale embedded systems (e.g., telephone centers), where down-time is required to be less than five minutes per year. Such systems cannot be taken down, upgraded, and restarted when software patches and upgrades arrive, since that would not respect their availability requirement. Consequently, an ERLANG system comes with support for upgrading code while the system is running, a mechanism known as *dynamic code replacement*. Moreover, these systems typically also require a high-level of responsiveness, and the soft real-time concerns of the language call for fast garbage collection techniques.

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

2.2 The three runtime systems of Erlang/OTP

Until quite recently, the Erlang/OTP runtime system was based on a *process-centric* architecture; i.e., an architecture where each process allocates and manages its private memory area. The main reason why this memory allocation scheme was chosen was that it was believed it results in lower garbage collection latency. Wanting to investigate the validity of this belief, in [17] we examined two alternative runtime system architectures for implementing concurrency through message passing: one which is *communal* and all processes get to share the same heap, and a *hybrid* scheme where each process has a private heap for process-local data but where a shared heap is used for data sent as messages and thus shared between processes. Nowadays, all three architectures are included in the Erlang/OTP release. We briefly review their characteristics.

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

Communal The biggest advantage is very fast ($O(1)$) interprocess communication, simply consisting of passing a pointer to the receiving process, reduced memory requirements due to message sharing, and low 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 multithreaded or multiprocessor implementation, since locking would be required in order to allocate in and collect the shared memory area in a parallel setting; see [8] for an excellent recent treatment of the subject of parallel real-time GC.

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

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

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

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

3. ORGANIZATION OF THE HYBRID ARCHITECTURE

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

3.1 The pointer directionality invariants

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

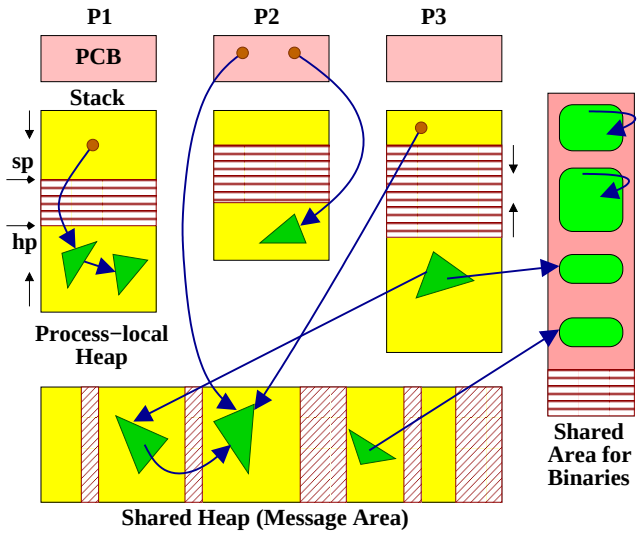


Figure 1: References allowed in the hybrid architecture.

- The area for binaries contains very few references and these are only from the header of a binary object to the start of the actual binary; these are shown in the figure. Note that these pointers will not be seen by the garbage collector.
- The message area only contains references to the shared area for binaries or to objects within the message area itself.
- Neither the shared area for binaries nor the message area contains any cyclic data.

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

3.2 Allocation in the hybrid architecture

To take full advantage of the organization of the hybrid architecture, the system needs to be able to distinguish between data which is process-local and data which is to be shared, i.e., used as messages. This can be achieved by user annotations on the source code, by dynamically monitoring the creation of data as proposed in [13], or by the static *message analysis* that we have described in [6] and integrated in the hybrid runtime system configuration of Erlang/OTP.

For the purposes of this paper, the details of the message analysis are unimportant and the interested reader is referred to [6]. Instead, it suffices to understand how the analysis guides allocation of data in the compiler. The allocation can be described as *allocation by default on the local heap and shared allocation of possible messages*. More specifically, data that is *likely* to be part of a message is allocated speculatively on the shared heap, and all other data on the process-local heaps. To maintain the pointer directionality invariants, this in turn requires that the message operands of all send operations are wrapped with a copy-on-demand operation, which verifies that the message resides in the shared area (as noted above, this can be an $O(1)$ operation), and otherwise copies the locally allocated parts to the shared heap. However, if the message anal-

ysis can determine that a message operand *must* already be on the shared heap, the test can be statically eliminated.

Note that the copying semantics of message passing in ERLANG and the absence of destructive updates allows the message analysis to safely both underapproximate and overapproximate use of data as messages. With underapproximation, 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 overapproximates too much, most of the data will be allocated on the shared heap, and we will not benefit from the process-local heaps; i.e., data will need to be reclaimed by global garbage collection.

3.3 Allocation characteristics of Erlang programs

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

4. GARBAGE COLLECTION IN THE HYBRID ARCHITECTURE

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

4.1 Garbage collection of process-local heaps

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

When a process does run out of its allocated memory, the runtime system garbage collects its heap using a generational Cheney-style semi-space stop-and-copy collector [7]. (Data has to survive two garbage collections to be promoted to the old generation.) Also when running native code instead of bytecode, 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 by reusing information from previous GC scans. Although this collector cannot give any real-time guarantees, pause times when collecting process-local heaps are typically not a problem in practice. This is because most collections are minor and therefore quite fast, and as explained above many Erlang applications have been programmed to use processes for specific, fine-grained tasks that require a relatively small amount of memory. Moreover, because process-local heaps can be collected independently, in a multithreaded 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.

[1] In this respect, process-local heaps are very much like *arenas* used by the Apache Web server [24] to deallocate all the memory allocated by a Web script once the script has terminated.

4.2 Garbage collection of binaries

The shared area for (large) binaries is collected using *reference counting* [11]. The count is stored in the header of binaries and increased whenever a new reference to a binary is created (when a binary is e.g. copied to the message area in the send operation). Each process maintains a *remembered list* of such pointers to binaries stored in the binary area. When a process dies, the reference counts of binaries in this remembered list are decreased. A similar action happens for references which are removed from the remembered list as part of garbage collection. Since cycles in binaries are not possible, cycle collection is not needed and garbage collection of binaries is effectively real-time.

4.3 Garbage collection of the message area

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

This situation can be ameliorated as follows:

- By splitting the message area into generations and performing generational collection on this area. In fact, one can employ a *non-moving* collector (such as *mark-and-sweep*) for the old generation to avoid the cost of repeatedly having to copy long-lived objects. (We still prefer to manage the young generation by a copying collector, because allocation is faster in compacted spaces.)

- By performing an optimization, called *generational process scanning*, which is the natural extension of generational root scanning from the sequential to the concurrent setting. More specifically, similarly to how generational stack scanning tries to reduce the root set which has to be considered during a process-local GC to only the "new" part of the stack, generational process scanning tries to reduce the number of processes whose memory areas are considered part of the root set. In implementation terms, the runtime system maintains information about which processes have been active (or received a message) since the last garbage collection and considers only those processes as part of the root set during the frequent minor collections.

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

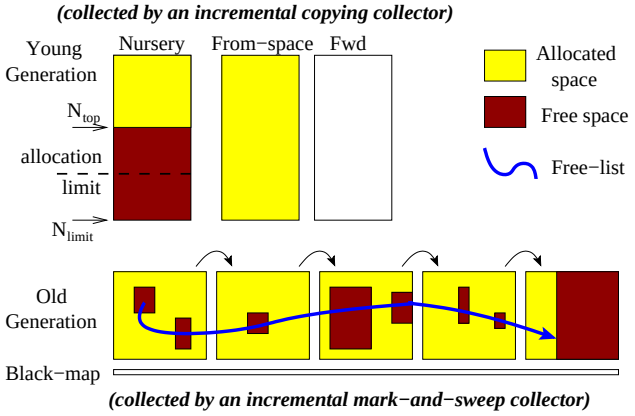


Figure 2: Organization of the message area.

Note that reference counting is *not* the most appropriate such method. The main reason is that one cannot wait until a process dies to decrease reference counts of messages that a process has sent to or received from other processes; consider for example the case of a Web server servicing requests. Furthermore reference counting typically imposes a non-negligible overhead. A different real-time or incremental GC is called for. We describe the one we designed and chose to implement in the next section.

5. INCREMENTAL COLLECTION IN THE SHARED MEMORY AREA

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

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

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

- We also use an area (currently an array of size Σ) of *forwarding pointers* (denoted as Fwd in Figure 2). The reason is that the mutator does not expect to find forwarding pointers in the place of objects. Since the mutator can access objects in the from-space during a collection cycle, forwarding pointers cannot be stored in this area. This would require either making the mutator perform a test on each heap pointer dereferencing and paying the corresponding cost, or the systematic use of *indirection* (as in [5]) and employing a *read barrier* mechanism to maintain the to-space invariant (as e.g. in [3]), which also has a non-trivial associated cost.

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

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

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

5.1 The incremental collection algorithm

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

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

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

If during collection of the young generation, the old generation overflows, its non-moving incremental garbage collector is triggered. This collector uses its own tricolor scheme [18] implemented as follows. We use a stack of references to keep track of gray objects. We also use a bit array (the *black-map*) to mark objects as black (i.e., fully processed). The black-map is needed since there is no room for any mark-bits in the actual objects.

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

5.1.1 Interplay between the mutator and the collector

In incremental tracing garbage collectors, the amount of work to be done in one collection cycle depends on the amount of live data when a *snapshot* of the root set is taken. Since we can not

know this quantity, we have to devise a mechanism that allows us to control how much allocation the mutator is allowed to do between two collection phases. (Relying on user-annotations to specify such a quantity is neither safe nor a "user-friendly" option in the typical multi-thousand line application domain of ERLANG.)

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

5.1.2 The work-based collector

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

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

5.1.3 The time-based collector

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

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

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

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

$$GC_{phases} = \frac{\Sigma - reclaimed\ after\ GC}{\Delta GC}$$

We now get:

$$w_M = \frac{f}{GC_{phases}}$$

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

5.2 Some optimizations

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

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

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

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

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

5.3 Characteristics of the collector

First of all note that the collector does not require any header word in the objects in order to perform incremental copying collection in the young generation. Therefore, it imposes no overhead to allocation. The collector instead uses an extra space, namely the forwarding area, whose size is bounded by Σ. Recall that Σ does not increase during GC and is not affected by the allocation characteristics of the program which is being executed. In the old generation, the only extra overhead is one bit per word for the black-map and a dynamically resizeable stack for the gray objects. Note that for the frequent collections of the young generation, the size of this gray stack is bounded by $\Sigma/2$. The space overhead of the incremental collector is quite low.

Without incrementality, the collector behaves as a *snapshot-at-the-beginning* algorithm [25, Section 3.3.1]. As explained above, in the incremental collector we postpone taking the snapshot of processes in the active queue as long as possible. By incrementally taking partial snapshots of the root set, 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 queue. This means that we will most likely collect more garbage than a pure snapshot-at-the-beginning collector.

An unfortunate side-effect of the root set minimization effort described above is that since we do not actually scan the old generation of process-local heaps during root-scanning, but only the set of references to the message area recorded during process-local

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

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

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

Table 1: Concurrency characteristics of benchmarks.

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

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

garbage collection, some of the rescued objects might actually be already dead at the start of the collection. An object may therefore be kept in the message area for a number of collection cycles until a major process-local garbage collection updates the remembered set of objects (or the process dies) and triggers the next collection cycle of the message area to finally remove the object. This however is an inherent drawback of all generational schemes.

6. MEASUREMENTS

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

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

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

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

yaws A high-performance multithreaded HTTP Web server where each client is handled by a separate Erlang process. Yaws[5] contains about 4,000 lines of code (excluding calls to functions in Erlang/OTP libraries such as HTTP, SSL, etc). We used `httperf` [19] to generate requests for Yaws.

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

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

The performance evaluation was conducted on a dual processor Intel Xeon 2.4 GHz machine with 1 GB of RAM and 512 KB of cache per processor, running Linux. The kernel has been enhanced with the `perfctr` driver [21], which provides access to high-resolution performance monitoring counters on Linux and allows us to measure GC pause times in μs.

6.1 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 in μs. Table 4 confirms that the time spent in the mutator and in performing garbage collection of process-local heaps is not affected by using the incremental collector for the message area. Depending on the configuration, the overhead for the incremental collector compared to the non-incremental collector ranges from a few percent to 2.5–3 times for most programs. The overhead is higher (5.6 times) for **worker** which is a program that was constructed to spend a significant part of its time allocating in (and garbage collecting) the message area.

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

6.2 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 Geo.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 message live data is hard to come by in this benchmark. The mean GC pause time values, in particular the geometric means, show a more consistent behavior. In fact, one can see a correlation between the value of w and the order of pause times in μs.

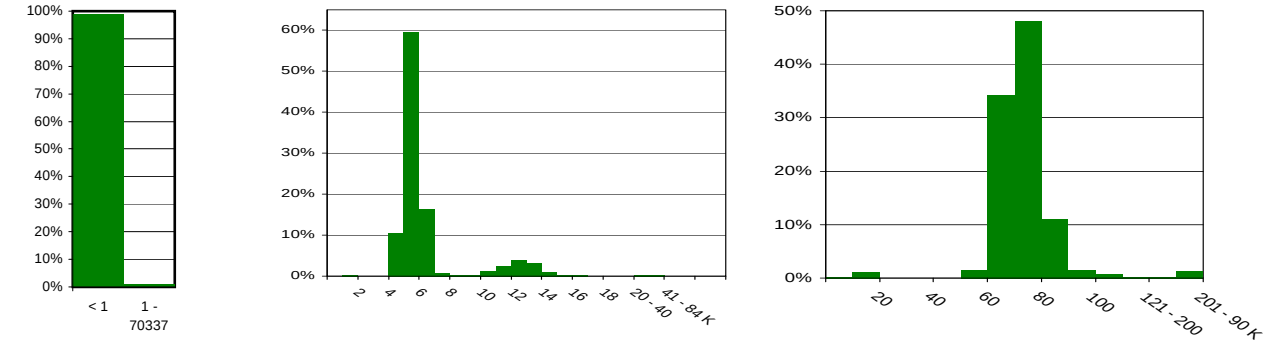
The distribution of pause times (in μs) for the benchmarks using the work-based incremental collector is shown in Figure 3.[6] The majority of collection phases are very fast, and only a very small percentage of the collections might be a problem for a real-time application. On the other hand, a work-based collector whose notion of work is defined in terms of "words reclaimed" naturally cannot guarantee an upper limit on pause times, as data to scavenge might be quite hard to come by.

A time-based incremental collector can in principle avoid this problem; see [3]. Care of course must be taken to detect the case when the mutator is allocating faster than the collector can reclaim, and take an appropriate action. Figure 4 (cf. also Table 2) shows

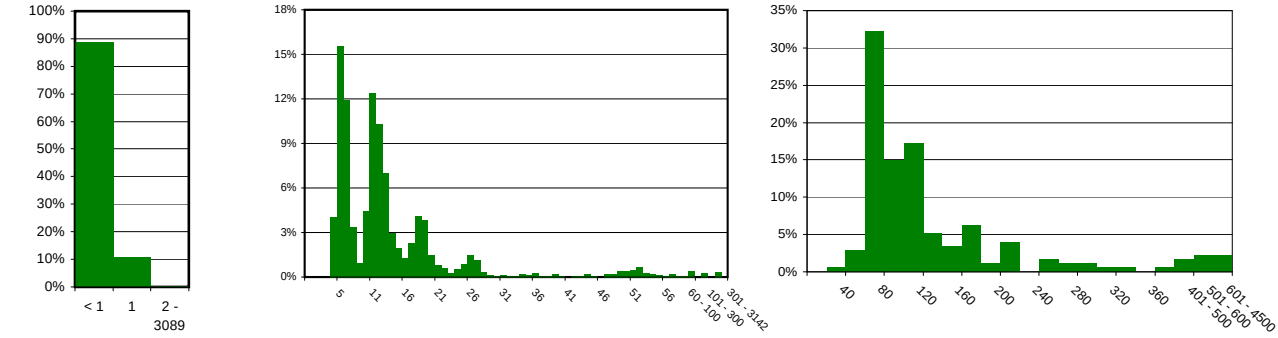
[4] ADHOC: Adaptation of Hyper Objects for Classification.

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

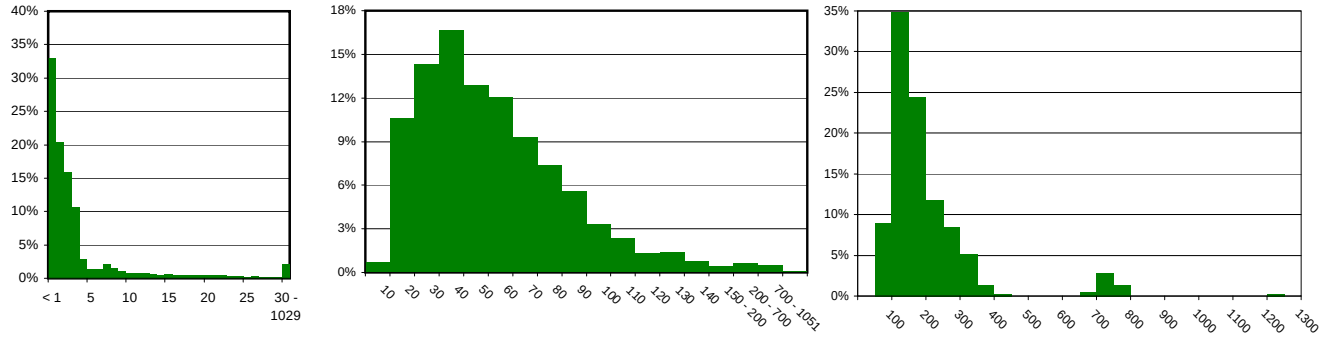
[6] **mnesia** is not included in Fig. 3 as its pause times do not show anything interesting.



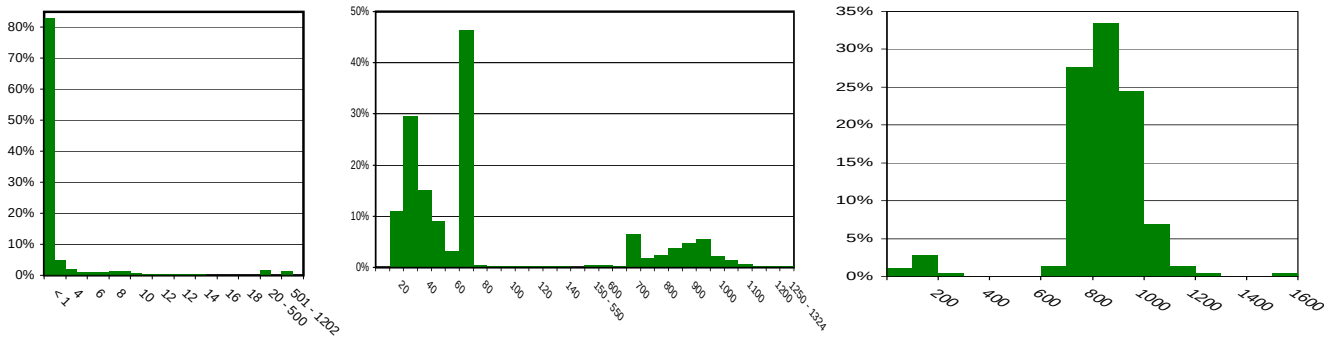
(a) **worker** ($w = 2$) (b) **worker** ($w = 100$) (c) **worker** ($w = 1000$)



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



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



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

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

Benchmark	Total Time (ms)			Local GC Pause Times (μs)			Message area GC Pause Times (μs)		
	Mutator	Local GC	MA GC	Max	Mean	Geo.Mean	Max	Mean	Geo.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

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

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

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

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

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

counts of GC pauses when running three of the benchmarks programs using the time-based incremental garbage collector with a t value of $1ms$ ($1000\mu s$). As mentioned in Footnote 2, when needed, the collector is allowed some small deadline extension, in order to possibly clean up after itself. This explains why there is a small number of values above $1000\mu s$. Note that in Figures 4(c) and 4(b) the number of GCs (the Y axis) is in logarithmic scale.

6.3 Mutator utilization

In any time window, the notion of *mutator utilization* is defined as the fraction of time that the mutator executes; see [8].

Figure 5 shows mutator utilization for the programs we used as benchmarks when using the work-based incremental collector for different values of w. The two synthetic benchmarks exhibit interesting patterns of utilization. As expected, the **worker** benchmark suffers from poor mutator utilization since it is designed to be allocation-demanding and be a serious challenge for the incremental collector. (The first interval of high utilization is the time before the first collection is triggered and the remaining two are periods after a collection cycle has finished and there is free space left in the nursery that the mutator can use for its allocation needs.) Similarly, the mutator utilization of **msort_q** drops significantly when live data in the message area is hard to come by. On the other hand, the mutator utilization of the three "real" programs is good—even for $w = 2$, alhough for **yaws** and **mnesia** this is apparent only with the time axis streched out; Figure 6 shows the same data as Figure 5(k) 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 Figure 7. For both **yaws** (mainly) and **mnesia** the utilization using the time-based collector is slightly worse than that when

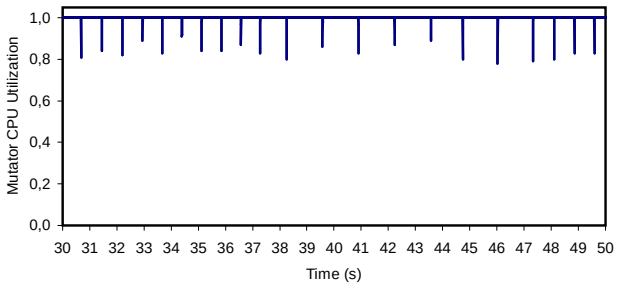
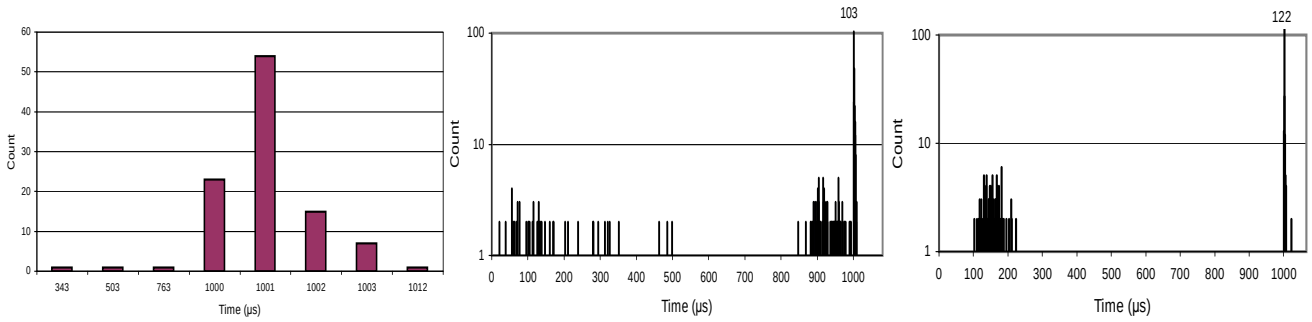


Figure 6: Mutator utilization of yaws ($w = 100$) for the work-based incremental collector shown in detail.

using the work-based one. The choice of an otherwise small, but compared with the total execution time relatively high in this case, t value ($1ms$) jeopardizes the mutator utilization of **msort_q**.

7. RELATED WORK

Runtime system organization. By now, several works have suggested detecting thread-local objects via static *escape analysis*, mainly of Java programs; notable among them are [4, 10, 22]. The goal has been to identify, conservatively and at compile time, the objects that are only going to be accessed by their creating thread and allocate them on the thread-local stack, thereby avoiding synchronization for these objects. In fact, the analysis of [22] is exploited in [23] by suggesting the use of thread-local heap chunks for non-escaping objects and a shared (portion of the) heap for all other data. Thread-local heaps for Java have also been advocated in [13], this time guided by information gathered by a profiler rather than by static analysis.



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

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

Note that, mainly because of the differences in the semantics of Java and ERLANG, all the above works attack the problem of memory allocation in the opposite direction than we do. Rather than allocating in thread-local heaps by default and using analysis to determine which objects are possibly shared, they try to determine objects that will *only* be accessed by their creating thread and allocate them in a thread-local memory area. In contrast, the message analysis that guides our allocator, identifies data that will probably be used in a message, enabling a speculative optimization that allocates data in the shared message area, thereby eliminating the need for copying at send time and making it possible to remove run-time checks altogether. The closest relative of our work is the memory architecture described in [12] which uses thread-local allocation for immutable objects in Caml programs.

Memory management of Erlang programs. The soft real-time concerns of the ERLANG language call for bounded-time GC techniques. One such technique, based on a mark-and-sweep algorithm taking advantage of the fact that the heap in an ERLANG system is *unidirectional* (i.e., is arranged so that the pointers point in only one direction), has been described in [1], but imposes a significant overhead and was never fully implemented. Similarly, [14] describes the design of a near-real-time compacting collector in the context of the Gambit-C Scheme compiler. This garbage collector was intended to be used in the Etos (Erlang to Scheme) system but never made it to an Etos distribution.

Incremental and real-time GC techniques. In the context of other (strict, concurrent) functional language implementations, the challenge has been to achieve low GC latency without paying the full price in performance that a guaranteed real-time garbage collector usually requires. Notable among them is the quasi real-time collector of Concurrent Caml Light [12] which combines a fast, asynchronous copying collector for the thread-specific young generations with a non-disruptive concurrent mark-and-sweep collector for the old generation (which is shared among all threads).

Many concurrent (real-time) garbage collectors for functional languages have also been proposed, either based on incremental copying [5, 15], or on *replication* [20] (see also [8] for a multiprocessor version of one such collector). The main difference between them is that incremental collectors based on copying require a read barrier, while collectors based on replication do not. Instead, they capitalize on the copying semantics of (pure) functional programs, and incrementally replicate all accessible objects using a mutation log to bring the replicas up-to-date with changes made by the mutator.

An excellent discussion and analysis of the trade-offs between work-based and time-based incremental collectors appears in [3]. Our work, although done independently and in a very different context than that of [3], is quite heavily influenced by it, presentation-wise. 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 (the collector of [3]) and our incremental collector. As a rather philosophical difference, we do not ask the user to guide the incremental collector by specifying the maximum amount of simultaneously live data or the peak allocation rate over the time interval of a garbage collection. More importantly, it appears that our system is able to achieve significantly lower pause times and better mutator utilization than the Metronome. We believe this can mostly be attributed to the memory allocation strategy of the hybrid runtime system architecture which is local-by-default. On the other hand, the utilization of our collector is not as consistent as that of [3] for adversarial, synthetic programs,[7] but then again we are interleaving the collector and the mutator in a much finer-grained manner (e.g., collecting just 2 words) or we are forcing our collector to run in a considerably smaller collector quantum ($1ms$ vs. $22.2ms$ which [3] uses).

8. ACKNOWLEDGMENTS

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

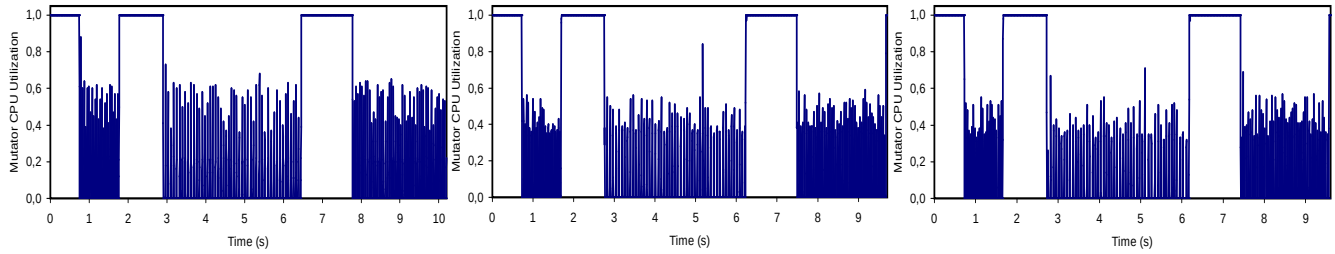
9. REFERENCES

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

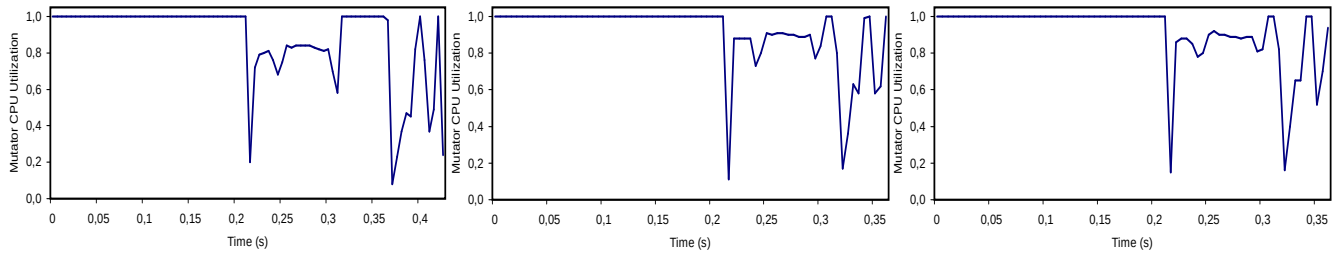
[2] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.

[3] 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*

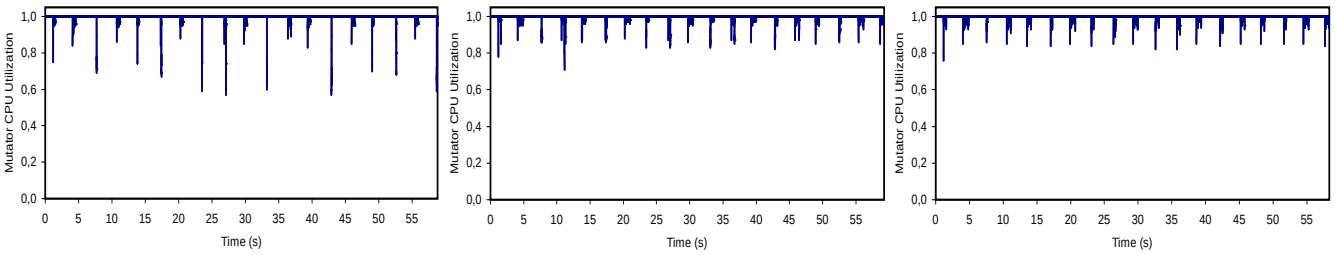
[7] Of course, this very much depends on the choice of these programs!



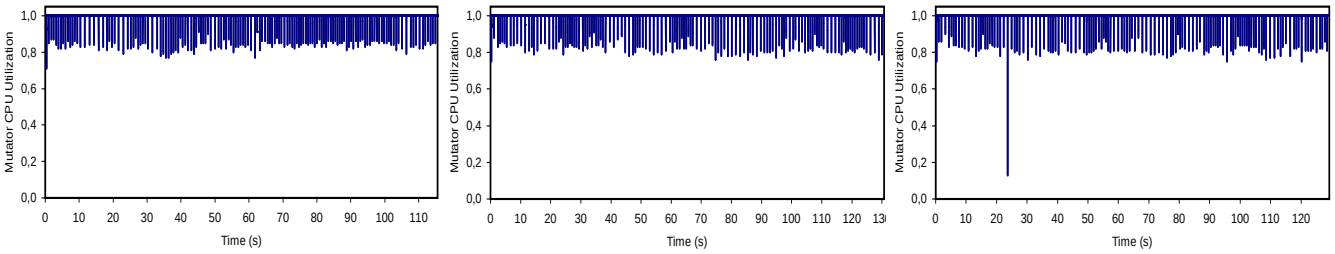
(a) **worker** ($w = 2$) (b) **worker** ($w = 100$) (c) **worker** ($w = 1000$)



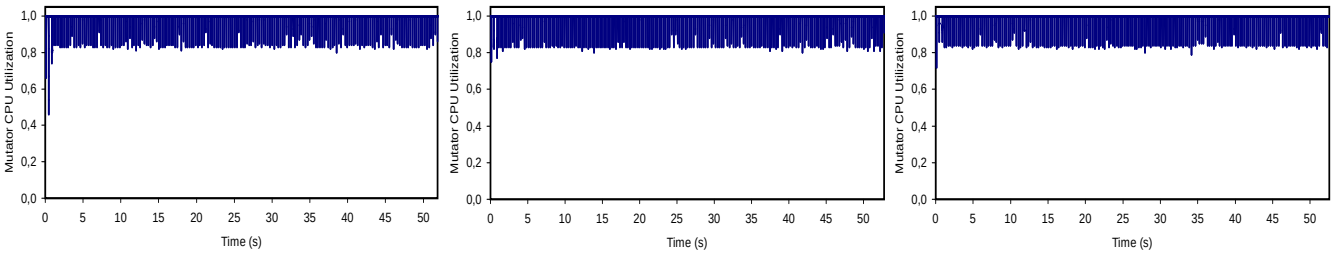
(d) **msort_q** ($w = 2$) (e) **msort_q** ($w = 100$) (f) **msort_q** ($w = 1000$)



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

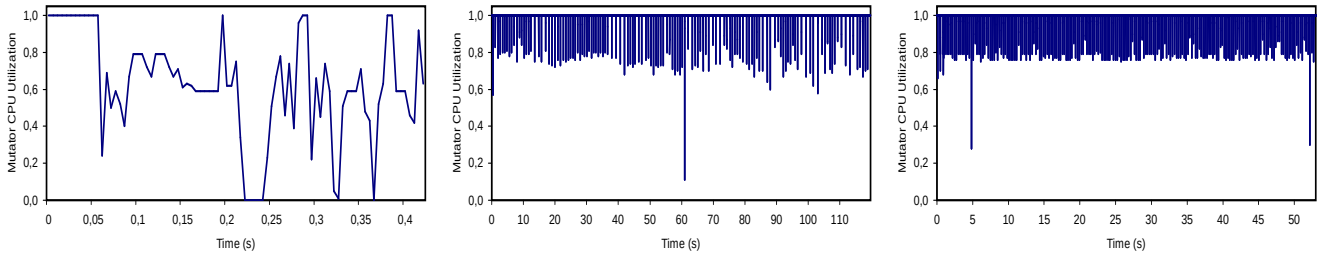


(j) **yaws** ($w = 2$) (k) **yaws** ($w = 100$) (l) **yaws** ($w = 1000$)



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

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



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

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

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 285–298, New York, N.Y., Jan. 2003. ACM Press.

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

[5] 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.

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43, New York, NY, Sept. 2000. ACM Press.

[17] 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.

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

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

[20] 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.

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

[22] 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.

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

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

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