A Case for the Unified Heap Approach to Erlang Memory Management

Marc Feeley
Département d'informatique et recherche opérationnelle
Université de Montréal
http://www.iro.umontreal.ca/~feeley

Abstract
The Erlang/OTP system has a process-centric model of memory management. Each process is created with a local heap where it allocates objects. An alternate approach used in the Erlang to Scheme (ETOS) system is to have a single system-wide heap in which all processes allocate. We have performed an empirical performance evaluation of both systems and found that in most cases the unified heap approach is better because it improves the speed of inter-process communication, especially when large objects are transferred, it reduces the overhead of garbage collection for processes with high allocation rates, and it avoids fragmentation.

1 Introduction
The Erlang programming language encourages the use of a functional programming style which puts a heavy stress on the dynamic memory management subsystem. It is thus important that the management of objects (i.e., allocation, garbage collection, etc) be efficient.

1.1 Process Local Heap Approach
The approach used in the Erlang/OTP system [Eri01] is to create each process with a local heap where it allocates objects and maintains the call stack. These process local heaps are managed separately by a compacting collector optimized for Erlang [B99]. Each process triggers collection independently from other processes. The process local heaps are initially quite small (on the order of a kilobyte unless a special option is used) and the heap will grow and shrink as the needs of the process evolve over time.

1.2 Unified Heap Approach
The ETOS system [Fee00, FL98] takes a different approach based on a single unified heap shared by all the processes on a node. This heap contains the continuation of each process (i.e., the call stack information) and the objects allocated by all processes. It is only when the heap is exhausted by one of the processes that the whole unified heap is collected. The algorithm used by the Gambit-C Schemesystem [G98], on top of which ETOS is built, is a hybrid between a Cheney-style [Ch70] two-space copying algorithm (for managing small objects) and a mark-and-sweep non-compacting algorithm (for managing large objects and objects created by the foreign-function interface).

1.3 Performance Folklore
These two approaches are considerably different. The aim of this work is to analyze these approaches and gain empirical evidence to better understand the tradeoffs. In private communications we have heard several unsubstantiated arguments in favor of each approach. Here is a summary:

- Pro “process local heap”
  1. Good real-time behavior - Shorter collection pauses will result if objects are distributed in several heaps instead of a large unified heap.
  2. Low collection overhead - Since there can be no references to objects in a heap from a different process, the heap and all the objects it contains can be deallocated simultaneously at no cost when the process terminates. Very short running processes may not even trigger the collector.
  3. High locality - The caches may perform better because the data accessed by a process is contained in a small area of memory.

- Pro “unified heap”
  4. Fast intra-node communication - An object can be communicated to another process on the same node simply by passing to the process a reference to the object. The object will remain in memory as long as some process refers to it (directly or not). With process local heaps it is necessary to deep copy the arguments of a process and the messages sent to its mailbox. There is a saving in time and also in space.
  5. Low fragmentation - The whole memory in the unified heap is available to any process that needs it; if a process requests a block of memory then the request will be satisfied without triggering the collector if at least that much memory is available. With process local heaps, a process cannot allocate in the heap of another process even if that heap contains large amounts of unused space. A low fragmentation will result in a better usage of memory which translates to less frequent invocations of the collector.
  6. Fast process spawning - Process descriptors can be handled like any other object and allocated cheaply in the unified heap. With process local heaps the allocation of processes and their
heaps is under the control of a different memory manager.

Our goal is to check the validity of these arguments by comparing the Erlang/OTP system and the ETOS system on benchmark programs specially designed to stress particular aspects of the system. We first describe each system in more detail and then explain the benchmarks and results.

2 Erlang/OTP

The general algorithm used by the Erlang/OTP system collector is described in [BS98]. Given the lack of other documentation we have examined the source code of version R7B-3 to better understand the particulars of the system.

Figure 1 shows how processes and their local heaps are organized. Explicit memory management (malloc/free) is used for allocating process descriptors and their local heaps. A 32768 entry process descriptor table (not shown) contains pointers to these descriptors for quickly mapping a process identifier to a process descriptor. Each process descriptor contains a pointer to its local heap. Erlang objects are allocated at one end and stack frames at the other end. A heap is resized by calling realloc, moving the stack and possibly the Erlang objects and updating all references (realloc may have to deallocate the old heap and allocate a new heap). Note that the explicit memory management of process descriptors and heaps is prone to fragmentation of the C heap and slow allocation/deallocation.

The collector performs compaction of the heap using a two phase algorithm. The first phase sweeps the whole heap and compacts it and the second phase slides the live objects back to their final destination. The main advantage of this algorithm is that it requires no extra space (a Cheney-style copying collector needs a "to-space" to copy the objects, which effectively doubles the memory required during the collection). On the other hand it means the collection time is proportional to the size of the heap (a Cheney-style copying collector takes time proportional to the live objects only).

The collector can optionally become generational by supplying a special option when the process is spawned. The default is to collect the whole local heap, i.e., a full sweep is performed, when the process exhausts its local heap.

The algorithm for resizing the heap is rather complex. Basically, a heap is resized when the live objects at the end of a collection occupy less than 25% or more than 75% of the heap. The final size is the smallest integer in a Fibonacci series that is greater than twice the live objects.

3 ETOS and Gambit-C

The Gambit-C system was designed for portability (it generates standard C code) and for general use (it makes few assumptions about the platform, operating system and compiler used). One of the consequences of supporting separate compilation and Scheme's tail-call semantics in standard C is that function calls and returns, especially between different modules, is much slower than if machine code was generated. Programs compiled with Gambit-C run on average about half as fast as when machine code is generated [FMRW97]. Fortunately most of the benchmarks used here rely on local tail-calls which are handled efficiently. There should not be a big difference between the performance observed with Gambit-C and a good compiler generating machine code.

<table>
<thead>
<tr>
<th>Process Descriptors</th>
<th>Process Local Heaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Organization of the Erlang/OTP memory

3.1 Memory Management

To accommodate multitasking environments, the unified heap can grow and shrink as the needs of the program evolve. To keep the OS process size proportional to the amount of live data, by default, at the end of a collection cycle, the heap is resized so that the live objects occupy 50% of the heap (i.e., there will be as much free space as live objects). This keeps the collection cost roughly proportional to the allocation cost. As shown in Figure 2 this resizing is achieved by allocating the heap in 512KB sections which are treated like a sequence within which allocation proceeds linearly (stack frames are allocated from the top down and objects smaller than 1KB are allocated from the bottom up). Each section contains a from-space and a to-space (not shown) which is needed for the Cheney-style copying. The sections, which are obtained from the C heap with malloc/free, are added or removed to approximate the 50% live ratio. At the boundary between sections there is some fragmentation (up to 32KB wasted space) needed for efficient handling of Scheme's "rest parameters" and so that the compiler can combine several allocations in the same basic block and perform a single heap overflow test.

In dedicated embedded environments with limited memory it would be more efficient to use all of physical memory for the unified heap, thus eliminating the section boundary fragmentation and resizing overhead. Since this setting is probably closer to the target applications of Erlang, the benchmarks have been run with a command line option which forces the heap to a given size of 10MB. The section boundary fragmentation is still present but not the resizing overhead.

Although the standard distribution of Gambit-C comes only with a blocking collector (i.e., the collector is only triggered when the heap is full), the collector can be turned into an incremental one with good real-time behavior as we have shown in previous work [LF90]. For example, a complex program (the Gambit-C compiler) running on a (now old) 500MHz DEC Alpha 21164A processor spent 8% of its time in the incremental collector and paused were roughly one millisecond on average and no more than 3 milliseconds. The incremental collector was not used in our experiments because it has not been kept in sync with the recent versions of Gambit-C.
3.2 Process Management

A priority based scheduler controls the execution of the processes. Each process has a priority represented as a floating point number. This gives a very fine control on the scheduling order and supports deadline scheduling (by setting the priority to the inverse of the time by which a process must be done). All synchronization primitives respect the process priorities. Red-black trees [CLR90] are used to implement the various process priority queues. This translates to slightly slower process queue manipulations, and in particular process spawning, than those of Erlang/OTP which only supports four priority levels and uses a small table of linked lists.

Process descriptors are allocated in the unified heap like other objects. Each process descriptor contains a reference to a continuation object which represents the set of frames in the process' call stack. Stack frames are allocated linearly at the top end of the heap. When there is a context switch, the new process allocates its own stack frames immediately after those of the previously active process. When a function returns the current stack frame is removed from the stack and the caller's stack frame, if it is not at the top of the stack, is copied there before control returns to the caller. This is known as the incremental stack heap strategy for implementing continuations [CH099, CHO88].

4 Comparing Erlang/OTP and ETOS

The versions of the systems we have evaluated are Erlang/OTP R73-3 and ETOS 2.4 with Gambi-C 4.0 alpha 7. The platform used is a PC with 1.4GHz AMD Athlon, 46KB L1 cache, 64KB L1 D-cache, 256KB L2 cache, 512MB RAM and running Linux kernel 2.4.2-2 and gcc 2.95.3. Programs were normally run at least three times on an unloaded machine and the median run time is reported. The heap size for ETOS was set to 10MB. No special command-line options were used for Erlang/OTP and Gambi-C.

4.1 Real-Time Behavior

Erlang is targeted to soft real-time applications for which it is desirable, but not required, to respond to messages within a small time frame (on the order of few milliseconds to tens of milliseconds depending on the application). In this context an incremental collector is useful if it can limit the collection pauses to a few milliseconds and leave enough time for the mutator (main program) to continue doing some useful work.

At first glance the process local heap approach seems attractive because, if the application data can be evenly distributed over the processes, each collection will complete in a fraction of the time it would take with a unified heap. When a process exhausts its (relatively small) heap the node will pause (i.e., no messages can be handled) while the collector works. Collection will be more frequent but shorter.

However the distribution of the collections in time must also be considered. If we take a global view of the system we see that the real-time response can be compromised. Consider the case where at a given point in time many processes have almost exhausted their local heap and will be collecting shortly. Even if individual collections are short there can be a long period of time when the system is not responsive because it is context-switching from one collecting process to the next. This may seem like a border case, but in fact this is to be expected in practice when many similar processes are spawned at about the same time; for example a web-server receiving simultaneous requests for a site's home page. The lack of coordination of the collections means it is impossible to put an upper bound on the pause time.

By using a unified heap the collector has a global view of memory management and can 'plan' how the collection work is parcelled out. The performance evaluation in [LP99] showed that maximal collection pauses are close to the average pause. We did not instrument the Erlang/OTP system to investigate its worst-case real-time behavior.

4.2 Memory Management Speed and Spawning Speed

To measure the speed of memory management we used the code in Figure 3. The function 'go' spawns 3!Process processes (one at a time) each of which allocates 4!-tuples to build a chain. Note that no garbage is generated in the tail-recursive function 'build' and that little data is communicated between the parent and child processes. The 'min_heap_size' option, which is ignored by ETOS, sets the minimum heap size to 232 words, which is the default in Erlang/OTP.

In addition to this program (garbo) we tried a variant where 3 out of 4 tuples immediately become garbage (garbob) obtained by replacing the "build" and "process" functions with

\[
\text{build}(0,X) \rightarrow X;
\]
\[
\text{build}(0,X) \rightarrow \text{build}(3\text{!tuples},1,2,3,X).
\]

\[
\text{process}(N,Parent) \rightarrow X = \text{build}(X \text{!div} 4,\text{nil}),
\]
\[
\text{Parent} ! 0.
\]
Finally we also tried variants of these programs which use the generational collector by adding the spawn option “{fullsweep_after, 2}” (garb0\_gen and garb75\_gen).

These programs were run with small amounts of allocation (0 \(\leq N \leq 200\) and \(\text{nProcs} = 1000000\)) and with large amounts of allocation (0 \(\leq N \leq 50000\) and \(\text{nProcs} = 1000\)). The run times are shown in Figures 4 and 5 respectively.

Several interesting things can be observed in the results. When \(N = 0\) there is no allocation (except for the processes) and the main cost is spawning a process and synchronizing with it. We see that Erlang/OTP can do this about twice as fast as ETOS (1.6 seconds compared to 3.0 for ETOS). This can be explained by the added complexity of manipulating red-black tree priority queues (this is consistent with a previous version of Gambit-C which used doubly-linked lists and no support for priorities, which could do these operations twice as fast as the current version).

For garb75 and values of \(N\) below 45 we see that the additional cost of allocating tuples is about the same in both systems (i.e., the curves have a similar slope). This comes as a surprise because the program generated by ETOS is an efficient machine code executable with inline code for the tuple allocations whereas Erlang/OTP is using emulated BEAM code. The overhead of emulating the BEAM code in this case does not adversely affect overall performance, probably because the emulator and emulated code are sitting in the level 1 cache and the memory accesses needed for allocating and initializing the tuples are relatively slow on this fast processor (for both systems). On the other hand the slope of garb0 in this range is slightly higher. This is probably due to the additional loop overhead (there are four times more calls to “build”) and not because of more expensive allocation.

For a value of \(N\) around 50 the Erlang/OTP process overflows their local heap and the collector is called, creating a step in the curve. This step is more pronounced for garb0 because the collector has to move more live objects. Moreover, in the case of garb0 but not in the case of garb75, there is an additional cost for resizing the local heap. For larger values of \(N\) we see similar steps each time the process overflows its local heap and garb75 is roughly twice as fast as garb0. The steps get wider in accordance to the Fibonacci progression of heap sizes. We also observe that the use of a generational collector does not give consistently better or worse performance.

Note that we cannot see any significant inflection of the
build(0, X) \rightarrow X;
build(N, X) \rightarrow build(N-1, (1, 2, 3, X)).

process(Struct, Parent) \rightarrow
Parent ! Struct.
go(0, Struct) \rightarrow done;
go(NProcs, Struct) \rightarrow
Child = spawn(comm.process,[Struct, self()]),
receive
X \rightarrow X
end,
go(NProcs-1, Struct).
start(NProcs, 0) \rightarrow
go(NProcs, build(0, nil)).

Figure 6: Code to measure cost of communication

curves that would indicate better cache efficiency when small local heaps are used. The level 1 data cache can hold a heap with about 3000 4-tuples (4 bytes for each element plus 4 bytes for the header) and the level 2 cache about 12000 4-
tuples, yet at these points in Figure 7 the garb0 curve is not anomalous.

Comparatively ETOS's performance is more regular and predictable. There is no step in the curve because the collection

Figure 7: Communication cost for sending and receiving a structure containing N 4-tuples.

cost of memory management would surely increase. However, we expect this to slow down typical programs by a factor of 1.5 and 2.5 which would
still be an improvement over Erlang/OTP. In a prototype implementation of Brooks's real-time collection algorithm [Bro84],
we obtained a slowdown of 1.19 for a program similar to the

garbage-free "build" function written in Scheme.

4.3 Intra-Node Communication

Figure 8: A program that causes fragmentation

the "build" function (compare this curve with the curve for
garb0 in Figure 7). On the other hand, the communication
cost for ETOS is a small constant.

4.4 Fragmentation

To show how severe fragmentation can become in the pro-

cess local heap approach we wrote the program shown in

Figure 8. It spawns a total of 10000 processes. Each process creates a chain of 500 4-tuples, thus growing its local
heap to about 14KB, and then creates a similar child process

and waits until the child is done. The last process immedi-
ately signals it is done. It is important to note that after

the chain of 500 4-tuples is created it is no longer used and

could be reclaimed. However the process does not allocate

Figure 7.

enough objects after this to trigger a collection so the local

heap does not shrink. Almost all the space in the local heap is

wasted. When the program is run the size of the UNIX

process grows to 145MB, which exceeds the memory capacity

of small embedded systems. By comparison the UNIX

process grows to 24MB with ETOS, even for much higher
values of N (each process accounts for about 2KB in the heap if we include the factor of 4 overhead caused by the to-space and 50% live ratio).

5 Conclusion

The empirical evidence presented in this paper shows that the unified heap approach to memory management has several benefits over the process local heap approach used by Erlang/OTP R7B-3. Memory management is somewhat faster (except for processes that allocate very little), intranode communication is much faster, and fragmentation is avoided. Although we did not empirically evaluate the real-time behavior, we think the unified heap approach is also superior because it can make tighter maximal pause guarantees than the process local heap approach when an incremental collector is used.

The contrived benchmark programs we used were designed to highlight problem areas of the process local heap approach. A more thorough investigation would be required to evaluate the approaches with real applications. It is clear that the performance will depend on the mix of operations of the application (process spawning rate, allocation rate, life expectancy of objects, communication rate, size of objects communicated, etc).

Just before publishing this paper, we have been informed by the High-Performance Erlang (HiPE) group that they have completed the implementation of the unified heap approach in a pre-release of Erlang/OTP. It will be interesting to see how this affects performance.

Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References


