Dynamic Memory Management with Garbage Collection for Embedded Applications

Roberto Brega and Gabrio Rivera
brega@ifr.mavt.ethz.ch and rivera@inf.ethz.ch

Swiss Federal Institute of Technology Zurich (ETHZ)
CH–8092 Zurich

Abstract

A software system can be called a *safe-system* with respect to memory, when it supports only strong-typing and it does not allow for the manual disposal of dynamic memory [2]. The first aspect guarantees that untyped, potentially dangerous operations are caught by the compiler or by run-time checks. The second issue is solved by the utilisation of an automatic memory reclamation scheme, i.e. a garbage collector.

In this paper we argue that the careful choice of the programming language, along with an automatic memory reclamation scheme can optimise memory usage, while ensuring that many of the logical errors related to memory can be avoided.

1 Introduction

Implementors of modern embedded machines must face two opposing tendencies. On the one hand, the system should behave like a modern desktop system, with its latest standards, the support for inter-networking, and ease-of-use. On the other hand, unlike a modern desktop system, it is not allowed to fail and must be able to run on tight memory conditions. Therefore implementors will often need to trade-off features for reliability.

This same issue was faced by the Institute of Robotics of the ETH Zurich, Switzerland and by Mechatronic Research Systems, an ETH spin-off company specialised in robotic control systems, as they were asked to realise state-of-the art control solutions for high-end mechatronic products by two Swiss companies.

Meyco Equipment AG needed a robot controller for a redundant manipulator spraying liquid concrete for tunnelling work. Besides the inherent difficulty of controlling the redundant, hydraulically actuated manipulator, the system needed to provide remote diagnostics through standard web services and remote controlling from an embedded console through TCP connections. Similar requirements were to be addressed for a fully autonomous fork-lift truck used in a warehouse of a Swiss manufacturing plant. The two machines needed to have months of up-time, while providing complex software services that could lead to small but inevitable memory leaks, thus having their long-term reliability undermined.

Unwilling to compromise on run-time safety, the Institute of Robotics chose to attack these problems at their roots, by deploying its in-house developed real-time operating system XO/2 [1].

2 The Role of Strong-Typed Programming Languages in Safe Systems

System components are the tools of a software engineer. The safer the tools, the more reliable is the system. It is well-known that languages, or more precisely proper language paradigms and type systems, can do a lot to help programmers. Strangely enough, despite the existence of better alternatives, a lot of safety-critical software gets implemented by means of programming languages that do a poor job of ensuring static or dynamic safety. In those cases, tools are used for uncovering errors that should not have been made possible in the first place, such as array indexes out of range, dangling pointers, casting errors, and memory leaks.

A type-system, as can be found in strong-typed languages, can be seen as the primary method for annotating all entities of a given program. The programmer can no longer tinker with memory addresses and he is forced to play by the rules of the language, in order that his program correctly compiles and is not trapped in a run-time assertion. The assertions bound to program entities enforce a more well-thought structural design, thus they can help in diminishing the chances of deep conceptual errors in the application as well.

3 Automatic Disposal of Dynamic Memory

The central knowledge of all references that exist for a particular object becomes hard to maintain as the dynamic loading of extensions increases. Even worse, it becomes impossible for a programmer to keep track of references in a safe way when the programming language does not impose restrictions on the passing and copying of references. This brings us to the conclusion that in a dynamically extensible system, explicit deallocation of objects is not feasible. Failing to realise this

introduces a new class of run-time problems, such as dangling references and memory leaks.

The only safe possibility for object disposal is by means of a system-wide mechanism performing automatic storage reclamation: a *garbage collector*. A garbage collector defines the liveness of heap objects by their reachability, starting from a working set of global and local references: an object is live only when there is some path that reaches it; on the other hand, an object is not live when no path can be found that reaches the object. When an object is not reachable, it can be disposed of safely.

4 The XO/2 Heap-Manager

XO/2 follows the guidelines presented in sections 2 and 3 in its dynamic memory manager: Each allocated object is bound to a type, and the responsibility of its reclamation is left to the system-wide garbage collector. The heap manager in XO/2 implements a *mark-and-sweep* garbage collector. This scheme has been chosen because of the possibility of an incremental implementation, and its non-destructive behavior during conservative-marking of the procedure's activation frames [3].

The algorithm consists of two distinct phases. The first one, the *marking phase*, traverses the objects graph, by starting from a well-defined *root-set*. The root-set is made up of the global pointers declared in a module variables-block, and all of the local pointers present in the processes' stacks and registers. Every traversed object is marked and their descendants traversed until every reachable object in the heap space has been marked. During the second phase, the *sweep-phase*, the collector disposes of the objects in the heap that have not been traversed.

The pre-emptive, real-time nature of the XO/2 task scheduling explicitly requires the collector to be incremental, interruptible and with a low-overhead with respect to the concurrent running programs, without blocking or delaying accesses to heap-objects.

The traversal phase (marking) of the collector is very sensitive to outside changes of the objects graph brought by pointer operations. Pointer assignment operations such as *p:=q* (where p and q are pointers of the same type or where q is a subtype of p) would invalidate the marking of the graph performed by the collector during the heap traversal.

In order to handle these operations, the compiler generates code for notifying the collector that a change in the graph has occurred. The newly injected code produces only a 1% overhead.

5 Space and Time Considerations

It is of outmost importance that the collector can be started when a low-memory condition occurs. Although memory-efficient traversal algorithms exist—such as the pointer-reversal descent—they are not suitable for pre-empted operation. Another possibility is the classic stack-based traversal. Unfortunately the stack-space needed for marking the heap is proportional to the depth of the graph: When the heap grows too much, the amount of memory available to the traversal decreases accordingly, thus potentially inhibiting the completion of the marking phase.

We devised a scheme that allows the collector to completely mark the heap without allocating additional memory. Consequently, the collector can always complete marking and sweeping regardless of the amount of free memory.

6 Real-World Experiences

The autonomous fork-lift truck and the RoboJet manipulator have been benchmarked against memory usage under different memory load conditions.

We inspected two different heap configurations: the first one with a 512 KB heap-size (basic product functionality), the second one with a 4 MB heap-size (continuous networked monitoring enabled), both of them with an average objects' size of 128 bytes. For completing one mark-and-sweep pass, while the full tasks constellation was running, the average collection time for the 512 KB heap-size amounts to 8 ms, and that for the 4 MB heap-size to 24ms. The tests show that the time needed for a complete collection pass remains low for heap-sizes that are common for our mechatronic applications.

7 Conclusion

As the list of requirements for embedded machines grows, application programmers can find themselves overwhelmed. A careful choice of the programming language, along with a system-wide mechanism for automatic storage reclamation provide invaluable help in keeping the system complexity low. We believe an incremental garbage collection mechanism, supporting transparent execution with respect to real-time tasks should find its way into today's mechatronic products.

References

[1] R. Brega. A real-time operating system designed for predictability and run-time safety. In *Proceedings of The Fourth International Conference on Motion and Vibration Control (MOVIC)*, pages 379–384, Zurich, August 1998.

[2] C. Szyperski and J. Gough. The role of programming languages in the life-cycle of safe systems. *Second International Conference on Safety Through Quality (STQ'95)*, Kennedy Space Center, Cape Canaveral, Florida, USA, October 1995.

[3] Paul R. Wilson, Uniprocessor Garbage Collection Techniques, *Submitted to ACM Computing Surveys*.