Five perspectives on modern memory management: Systems, hardware and theory

Dynamic memory management is a vital feature of all modern programming languages. Yet, heap allocation is difficult for the application programmer to manage correctly and for the systems developer to implement efficiently. Unfortunately, and for very good reasons, memory management errors are not confined to novices: at least one study has shown that up to 40% of programmer time is wasted on hunting down memory management errors.

Automatic dynamic memory management (garbage collection) relieves the programmer of the burden of making a global decision – can an object no longer be used and thus is it safe to free the memory that it occupies? – in a local context. The usual alternative of adding memory management book-keeping detail to module interfaces is undesirable because it weakens abstractions and reduces extensibility. Garbage collection, on the other hand, uncouples the problem of memory management from interfaces instead of dispersing it throughout the code.

The field of memory management continues to present new challenges. The widespread use of languages such as Java, Perl and Python in substantial applications of commercial importance has brought garbage collection into the mainstream: it is more important than ever before. At one extreme, server applications are starting to demand very large numbers of threads, multi-gigabyte heaps and high throughput. At the other end, the advent of Microsoft’s Common Language Infrastructure and C# in particular on the one hand, and the prevalence of Java applications in small devices such as phones on the other, means that garbage-collected applications will become prevalent on the desktop and in the pocket. In this special issue of the Science of Computer Programming, we present five very different perspectives on modern memory management.

Erlang is a strict, functional programming language that supports concurrency, communication, distribution and fault tolerance. It is a key component of products and services for companies like Ericsson, Nortel and T-Mobile (for example, in Ericsson’s AXD301 which provides the ATM switch infrastructure for BT’s network in the UK). Whereas many programming languages expect to support between a few and a few hundred threads, Erlang applications may use hundreds of thousands of concurrent processes. It is therefore vital that Erlang processes be lightweight and highly responsive. In the first article in this special issue, Efficient Memory Management for Concurrent Programs that Use Message Passing, Sagonas and Williamson show how the particular characteristics of Erlang can be exploited to provide efficient garbage collection with short pause times. Their key contribution is a hybrid architecture combining process-local heaps that can be collected independently with a shared area for messages passed between processes. A static analysis is used to speculatively allocate data that might be used as a message in the shared area. An incremental, generational collector imposes little overhead on the user program and, because Erlang has no destructive update, requires no costly barrier mechanisms.

The goal of the Cyclone project is a safe, low-level language. Cyclone is a dialect of C that uses programmer-supplied annotations, a type system, a flow analysis and run-time checks to ensure that programs are safe. One of the first challenges that must be surmounted in such a project is making memory management safe. In Safe Manual Memory Management in Cyclone, Swamy, Hicks, Morrisett, Grossman and Jim describe how statically scoped regions and tracked pointers can be used to construct a variety of safe memory management abstractions. Cyclone pointers may be aliased, unique (alias-free) or reference-counted; a compile-time flow analysis checks correct usage of unique pointers. Unique pointers can also be used to build new memory management abstractions such as dynamic allocation arenas, thereby avoiding the limitations imposed by stack-like, last-in–first-out disciplines. Finally, they describe their
experience using these mechanisms with real programs, including applications and Linux device drivers ported to Cyclone as well as programs written directly in Cyclone.

Object-oriented computation has become the dominant paradigm of the late twentieth and early twenty-first century. Despite this, it has had little influence on computer architecture. In the third article, An Object-Aware Memory Architecture, Wright, Seidl and Wolczko investigate how hardware support for objects, co-designed with the virtual machine, can not only lead to better memory system performance but also enable new memory management algorithms that cannot be implemented efficiently in software. Their architecture is based on an address space for objects using their IDs, mapped by a translator to physical addresses. Indirect access to objects through an object table has been rejected since the late 1980s, despite some advantages for garbage collection. The hardware approach described here avoids the overhead of indirection but retains its advantages. Their cache tags lines with object ID and offset pairs, allowing object loads to go directly to the cache index/tag match hardware. This architecture also allows in-cache garbage collection with little global memory traffic. If most accesses are indeed to recently allocated objects, then the cache is likely to be a good approximation of a young generation. Thus, fast generational-like collection is possible without the need to write back objects to memory. Finally, an architecture based on object IDs allows objects to be relocated concurrently without long mutator (user program) pauses since, unlike in conventional systems, it is no longer necessary to update all references to a relocated object “at once”.

The last two papers in this special issue seek to analyse garbage collection. In his article, On Measuring Garbage Collection Responsiveness, Printezis considers methods for evaluating and illustrating the responsiveness of low-latency collectors. His focus is not on hard real-time systems, in which deadlines must always be met, but on soft real-time systems in which occasional and limited deviations from a responsiveness target can be tolerated (for example, a customer might tolerate a single phone call that takes longer than usual to initiate, but not if this happens regularly). Printezis compares current methods of representing garbage collector pause time, such as minimum and bounded mutator utilisation curves, and GC overhead graphs; these techniques reveal worst-case performance and are best suited to hard real-time systems. Printezis introduces three new \textit{Vmetrics}, designed to evaluate how well a collector meets a given soft real-time goal. A common way to represent responsiveness is to graph either mutator utilisation or GC overhead in a given time slice. The disadvantage of these representations is that they reveal performance with respect to only a single, fixed time slice. Hence, Printezis introduces \textit{cathedral graphs}, a completely new way of representing GC overheads over a range of time slice durations. He concludes with a discussion of \textit{percentile mutator utilisation} graphs that generalise minimum mutator utilisation graphs and are better matched to soft real-time goals.

Generational garbage collection is the most common implementation of GC. Their performance rests on the weak generational hypothesis that “most objects die young” [6]. Since the introduction of generational collectors, researchers have experimented with a variety of region-based heap organisations including older-first [5] and Beltway [3]. However, different programs behave better with different forms of regional collection. Why? In Linear Combinations of Radioactive Decay Models for Generational Garbage Collection, Clinger and Rojas address the theoretical dividing line between different styles of collector. They use linear combinations of Baker’s ‘radioactive decay’ model of object lifetimes [2] to calculate the efficiency of several idealised garbage collectors. Their models explain a number of otherwise puzzling experimental results.

Acknowledgements

The call for contributions to this special issue elicited 19 submissions, of which 5 were selected for publication. I was helped in the preparation of this special issue by a panel of reviewers, chosen for their expertise in the field, and after the submissions had been received. Each paper was read by at least three reviewers as well as by the guest editor. Care was taken to avoid conflicts of interest when assigning reviewers; members of the panel were not involved in the discussion of any submission for which they had a conflict.

I would like to thank the review panel for their assistance: David Bacon (IBM T.J. Watson Research Center), Emery Berger (University of Massachusetts), Hans Boehm (HP Labs), Trishul Chilimbi (Microsoft Research), David Detlefs (Sun Microsystems), Amer Diwan (University of Colorado at Boulder), Kevin Hammond (University of St. Andrews), Tony Hosking (Purdue University), Rick Hudson (Intel), Hillel Kolodner (IBM Haifa Research Lab), Eliot Moss (University of Massachusetts) and Marc Shapiro (INRIA Rocquencourt). I would also like to thank the secondary reviewers: Luc Moreau, Yoav Ossia, Harel Paz, Erez Petrank, Leah Shalev, Daniel Spoonhower, Martin Vechev and Eran Yahev. Finally, Bas van Vlijmen’s help has been invaluable in the production of this special issue.
References


Guest Editor
Richard Jones
Computing Laboratory,
University of Kent,
Canterbury,
Kent, CT2 7NF,
UK
E-mail address: R.E.Jones@kent.ac.uk.
URL: http://www.cs.kent.ac.uk/~rej.
Available online 27 June 2006