As some optimizations rely on the class hierarchy, important runtime hooks exist in the class loader that can trigger recompilation if previously held assumptions are now incorrect. We described this in more detail in the earlier section "On-stack replacement."

Garbage Collection

The Memory Management Toolkit (MMTk) is a framework for building high-performance Memory Management implementations (Blackburn et al., May 2004). Over the last five years, MMTk has become a heavily used piece of core infrastructure for the garbage collection research community. Its highly portable and configurable architecture has been the key to its success. In addition to serving as the memory management system for Jikes RVM, it has also been ported to other language runtimes, such as Rotor. In this section, we will just touch on some of the key design concepts of MMTk. For general information on garbage collection algorithms and concepts, the best reference is the book *Garbage Collection: Algorithms for Automatic Dynamic Memory Management* (Jones and Lins 1996).

An underlying meta-principle of MMTk is shared with Jikes RVM as a whole: by relying on an aggressive optimizing compiler, high-performance code can be written in a high-level, extensible, and object-oriented style. This principle is most developed in MMTk, where even the most performance-critical operations, such as fast path object allocation and write barriers, are cleanly expressed as well-structured, object-oriented Java code. On the surface, this style appears to be completely incompatible with high-performance (to allocate a single object, there are dozens of source-level virtual calls and quite a bit of complex numeric computation). However, when the allocation sequence is recursively inlined and optimized by the optimizing compiler, it yields tight inline code sequences (on the order of 10 machine instructions to allocate an object and initialize its header) that are identical to those that can be achieved by handcoding the fast path sequence in assembly code or hard-wiring it into the compilers.

MMTk organizes memory into *Spaces*, which are (possibly discontiguous) regions of virtual memory. There are a number of different implementations of Space provided by MMTk, each embodying a specific primitive mechanism for allocating and reclaiming chunks of the virtual memory being managed by the Space. For example, the CopySpace utilizes bump pointer allocation and is collected by copying out all live objects to another Space; the MarkSweepSpace organizes memory by using free lists to chain together free chunks of memory and supports collection by "sweeping" unmarked (dead) objects and relinking them to the appropriate free list.

A *Plan* is MMTk's terminology for what one typically thinks of as a garbage-collection algorithm. Plans are defined by composing one or more Space instances together in different ways. For example, MMTk's GenMS Plan implements the fairly standard generational mark-sweep algorithm, composing a CopySpace to implement the nursery and a MarkSweep space to implement the mature space. In addition to the Spaces used to manage the user-level Java heap, all MMTk Plans also include several spaces used for virtual-machine-level memory. In