Practical Factors

- Real-time Garbage Collection
- Memory Overheads
- Interaction with Virtual Memory
- Garbage Collection for Weakly-Typed Languages

Real-time Garbage Collection

- Real-time collectors built from incremental collectors
  - Schedule an incremental collector as a periodic task
  - Runtime allocated determines amount of garbage that can be collected in each period
  - The amount of garbage that can be collected can be measured: how fast can the collector scan memory (and copy objects, if a copying collector)
  - The programmer must bound the amount of garbage generated to within the capacity of the collector
    - Hard real-time systems **must** always stay within the bounds of the collector
    - Soft real-time systems meet statistical bounds

A Real-time Garbage Collector with Low Overhead and Consistent Utilization

David F. Bacon
dfb@watson.ibm.com

Perry Cheng
perryche@us.ibm.com

V.T. Rajan
vtrajan@us.ibm.com

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT

Now that the use of garbage collection in languages like Java is becoming widely accepted due to the safety and software engineering benefits it provides, there is significant interest in applying garbage collection to hard real-time systems. Past approaches have generally suffered from one of two major flaws: either they were not provably real-time, or they imposed large space overheads to meet the real-time bounds. We present a mostly non-moving, dynamically defragmenting collector that overcomes both of these limitations: by avoiding copying in most cases, space requirements are kept low; and by fully incrementalizing the collector we are able to meet real-time bounds. We implemented our algorithm in the Jikes RVM and show that at real-time resolution we are able to obtain mutator utilization rates of 45% with only 1.6–2.5 times the actual space required by the application, a factor of 4 improvement in utilization over the best previously published results. Defragmentation causes no more than 4% of the traced data to be copied.

General Terms

Algorithms, Languages, Measurement, Performance

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Java; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

Keywords

Read barrier, defragmentation, real-time scheduling, utilization

1. INTRODUCTION

Garbage collected languages like Java are making significant inroads into domains with hard real-time concerns, such as automotive command-and-control systems. However, the engineering and product life-cycle advantages consequent from the simplicity of programming with garbage collection remain unavailable for use in the core functionality of such systems, where hard real-time constraints must be met. As a result, real-time programming requires the use of multiple languages, or at least (in the case of the Real-Time Specification for Java [9]) two programming models within the same language. Therefore, there is a pressing practical need for a system that can provide real-time guarantees for Java without imposing major penalties in space or time.

We present a design for a real-time garbage collector for Java, an analysis of its real-time properties, and implementation results that show that we are able to run applications with high mutator utilization and low variance in pause times.

The target is uniprocessor embedded systems. The collector is therefore concurrent, but not parallel. This choice both complicates and simplifies the design: the design is complicated by the fact that the collector must be interleaved with the mutators, instead of being able to run on a separate processor; the design is simplified since the programming model is sequentially consistent.

Previous incremental collectors either attempt to avoid overhead and complexity by using a non-copying approach (and are therefore subject to potentially unbounded fragmentation), or attempt to prevent fragmentation by performing concurrent copying (and therefore require a minimum of a factor of two overhead in space, as well as requiring barriers on reads and/or writes, which are costly and tend to make response time unpredictable).

Our collector is unique in that it occupies an under-explored portion of the design space for real-time incremental collectors: it is a *mostly non-copying* hybrid. As long as space is available, it acts like a non-copying collector, with the consequent advantages. When space becomes scarce, it performs defragmentation with limited copying of objects. We show experimentally that such a design is able to achieve low space and time overhead, and high and consistent mutator CPU utilization.

In order to achieve high performance with a copying collector, we have developed optimization techniques for the Brooks-style read barrier [10] using an "eager invariant" that keeps read barrier overhead to 4%, an order of magnitude faster than previous software read barriers.

Our collector can use either time- or work-based scheduling. Most previous work on real-time garbage collection, starting with Baker's algorithm [5], has used work-based scheduling. We show both analytically and experimentally that time-based scheduling is superior, particularly at the short intervals that are typically of interest in real-time systems. Work-based algorithms may achieve short individual pause times, but are unable to achieve consistent utilization.

The paper is organized as follows: Section 2 describes previ-

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.
Copyright © 2003 ACM 1-58113-628-5/03/0001 $5.00.

Bacon *et al.*, "A real-time garbage collector with low overhead and consistent utilization". ACM Symposium on Principles of Programming Languages, New Orleans, LA, USA, January 2003. DOI: 10.1145/604131.604155

Memory Overheads

- Garbage collection trades ease-of-use for predictability and overhead
- Garbage collected programs will use significantly more memory than **correctly written** programs with manual memory management
  - Many copying collectors maintain two semispaces, so double memory usage
  - But – many programs with manual memory management are **not correct**

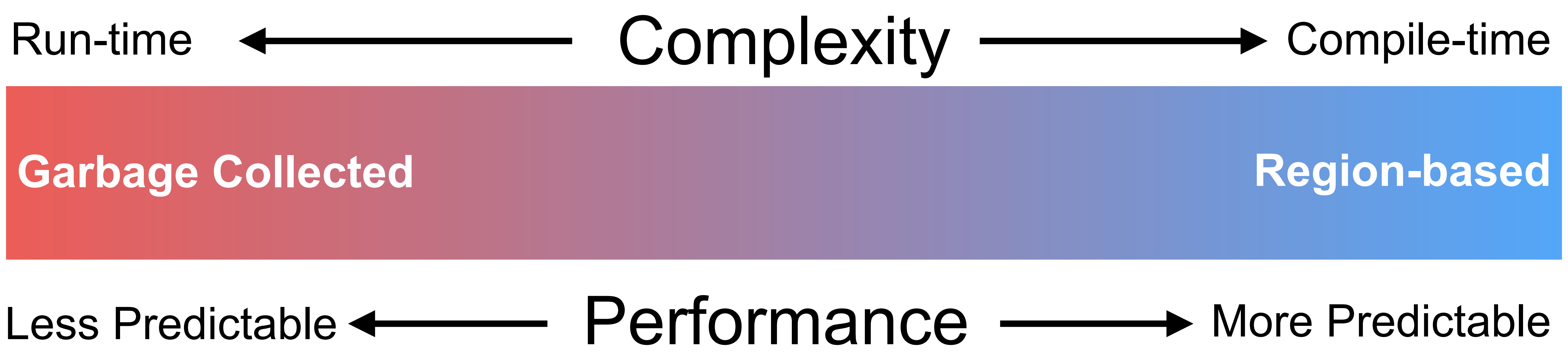
Interaction with Virtual Memory

- Virtual memory subsystems page out unused data in an LRU manner
- Garbage collector scans objects, paging data back into memory
- Leads to thrashing if the working set of the garbage collector larger than memory
  - Open research issue: combining virtual memory with garbage collector

Garbage Collection for Weakly-typed Languages

- Collectors rely on being able to identify and follow pointers, to determine what is a live object – they rely on strongly-typed languages
- Weakly typed languages, such as C, can cast any integer to a pointer, and perform pointer arithmetic
  - Implementation-defined behaviour, since pointers and integers are not guaranteed to be the same size
  - Difficult, but not impossible, to write a garbage collector for C:
    - Need to be conservative: any memory that might be a pointer must be treated as one
    - Boehm-Demers-Weiser collector can be used for C and C++ (http://www.hboehm.info/gc/) – works for strictly conforming ANSI C code, but beware that much code is not conforming
  - Other weakly typed languages may suffer from similar problems
    - Strongly typed, but dynamic, languages, such as Python, not an issue

Memory Management Trade-offs



- Rust pushes memory management complexity onto the programmer
  - Predictable run-time performance, low run-time overheads
  - Uniform resource management framework, including memory
  - Limits the programs that may be expressed – matches common patterns in good C code
- Garbage collection imposes run-time costs and complexity, simpler for programmer

Summary

- Garbage collection
  - Mark-sweep
  - Mark-compact
  - Copying collectors
  - Generational algorithms
  - Incremental algorithms
- Real-time garbage collection
- Practical factors