OASIS: Organic Aspects for System Infrastructure Software
Easing Evolution and Adaptation through Natural Decomposition

Celina Gibbs and Yvonne Coady
University of Victoria

Abstract

It is becoming increasingly clear that we are entering a new era in systems software. As the age-old tension between structure and performance acquiesces, we can finally venture beyond monolithic systems and explore alternative modularizations better suited to evolution and adaptation. The OASIS project explores the potential of aspects to naturally shape crosscutting system concerns as they grow and change. This paper describes ongoing work to modularize evolving concerns within high-performance state-of-the-art systems software, and outlines some of the major challenges that lie ahead within this domain.

1. Introduction

With the constant demand for system change and upgrades comes the need to simplify and ensure accuracy in this process. As structural boundaries decay, non-local modifications compound the cost of system modification. Aspect-Oriented Programming (AOP) [7] aims to improve structural boundaries of concerns that are inherently crosscutting – no single hierarchical decomposition can localize both the crosscutting concern and the concerns it crosscuts. This paper explores the application and challenges of an aspect-oriented based solution to the growing problems of evolution and adaptation in system infrastructure software. We propose an approach that is incremental and organic in nature, intrinsically promoting more natural boundaries for change.

This paper begins with a high-level overview of a recently developed non-monolithic implementation of garbage collection in Java. Recent results have demonstrated that a preliminary separation of concerns into an object hierarchy does not compromise, and in fact can be augmented to improve performance relative to existing monolithic implementations [1]. The specific ways in which AOP could facilitate more cohesive system evolution when applied to domain-specific patterns is overviewed, and demonstrated by a sample aspect we have introduced to the system. We then extend these results to apply to the problem of adaptability by considering how these aspects could better facilitate change when dynamically coupled with system state and/or user demands. The precise ways in which we may be able to combine modular crosscutting with dynamism to create a more adaptable system are discussed. Though recent successes in the area of dynamic aspects provide the opportunity to immediately realize evolution of these concerns in a live system [2, 9], many challenges in extending this support to infrastructure software remain. The paper concludes with an overview of some of these challenges.

2. Background

The Jikes Research Virtual Machine (RVM) [5, 6] affords researchers the opportunity to experiment with a variety of design alternatives in virtual machine infrastructure. The project is open source and written in Java. One of the core system elements that stand to receive much attention in this testbed environment is garbage collection (GC). State of the art technologies for improving GC performance are still evolving, in particular for multiprocessor systems.

The benefits of GC are well known and have been appreciated for many years in many programming languages. GC separates memory management issues from program design, in turn increasing reliability and eliminating memory management errors. Though GC has improved significantly over the last 10 years, on going work aims to further reduce costs and meet application specific demands. Costs not only involve performance impact, but also configuration complexity. For example, in 1.4.1 JDK there are six collection strategies and over a dozen command line options for tuning GC [4]. Basic strategies, such as reference counting, mark and sweep, and copying between semi-spaces, have been augmented with hybrid strategies, such as generational collectors, that treat different areas of the heap with different collection algorithms. Collectors not only differ in the way they identify and reclaim unreachable objects but they can also significantly differ in the ways they interact with user applications and the scheduler.

3. The Dawn of Modularized, Performent GC

Recently, Blackburn et al. detailed the modularization of a monolithic memory management system for Jikes [1]. Their Memory Management Toolkit (MMTk) was designed with two goals in mind: flexibility, in support of future development, and performance, relative to a monolithic implementation and a standard C *malloc* implementation. Results demonstrate that this implementation improves modularity without sacrificing performance. In their study they measure modularity of their new MMTk by comparing metrics such as lines of code, lack of cohesion of methods (LCOM) and cyclomatic complexity with two earlier versions of the RVM. The results show MMTk as the clear winner in all of these comparisons. For example, cyclomatic complexity is a measure of branching and looping complexity within a method. MMTk displays an average ranging from 1.5 to 5 times lower in this area than the previous RVM memory management versions.

The performance tests include a comparison of allocation and tracing rates along with raw speed comparison. The tests run on standard benchmarks reveal MMTk has better performance overall than its monolithic counterpart. Though one comparison shows C outperforming MMTk by 6%, compiler inlining in the MMTk version realizes substantial performance gains up to 60% over the C implementation.

4. OASIS: Shaping Fertile Code

We have just embarked on the OASIS project, which will take the refactoring of this system one step further, modularizing crosscutting concerns within MMTk. We believe aspects will not only serve to increase the flexibility of the system, but will further facilitate system evolution and research. MMTk can now be used as a structured benchmark to measure performance and modularity of an AOP implementation against. By using the same techniques used in the comparison of MMTk to its monolithic counterpart (LCOM, cyclomatic complexity, etc), the level of modularization and performance impact incurred by aspects can be meaningfully quantifiably assessed.

In addition to improving on these metrics, added benefits of an aspect-oriented implementation are the semantic and practical value of the explicit relationship between crosscutting concerns and the concerns they crosscut. Though the internal structure creates a clearer picture of the system for developers to work with when researching new VM techniques, this factoring-out of crosscutting concerns also provides developers with the ability to plug/unplug more system elements as configuration options. Eventually employing dynamic (run-time) aspects would further allow the system to switch GC strategies on-the-fly, at run-time, enabling more effective attempts at feedback-based, system-wide optimization. This presents the possibility of a system that could dynamically switch collectors dependant on system state and/or user input. The following section overviews some of the domain-specific design patterns we have started to shape as aspects within MMTk.

5. Domain-Specific Design Patterns: Shaping Aspects of GC

Blackburn et al. use design patterns in the development of MMTk for reuse and efficiency. Four domain specific design patterns are detailed in [1]. For reuse, a pattern for *prepare and release phases* exploits collection phases to simplify the development of new collectors, and a pattern for *multiplexed delegation* passes on collector tasks to appropriate memory management policies. For efficiency, a pattern for *hot and cold paths* optimizes the common path, and a pattern for *local and global scope* minimizes contention and synchronization between multiple collector threads in multiprocessor systems.

The composition of collectors is broken down into three elements: mechanisms, policies and plans. Mechanisms are collector-neutral and shared among collectors. Policies manage contiguous regions of virtual memory, and are expressed in terms of mechanisms. Plans define the ways in which collectors are composed, as described in terms of policies.

At a top level, the algorithm used by collectors in MMTk integrates the domain-specific patterns previously introduced. This is evident by looking at the control flow through the stop-the-world superclass, extended by all plans that suspend the system while GC takes place. The implementation has three phases: *prepare*, *processAllWork*, and *release*. The prepare/release phases are further specialized by the multiplexed delegation pattern, divided according to the global and local context pattern, and optimized by hot and cold paths. For

example, given the simple case of a single collector thread on a uniprocessor, the structure of local and global scopes within prepare and release phases can be captured by the simple finite state machine in Figure 1.

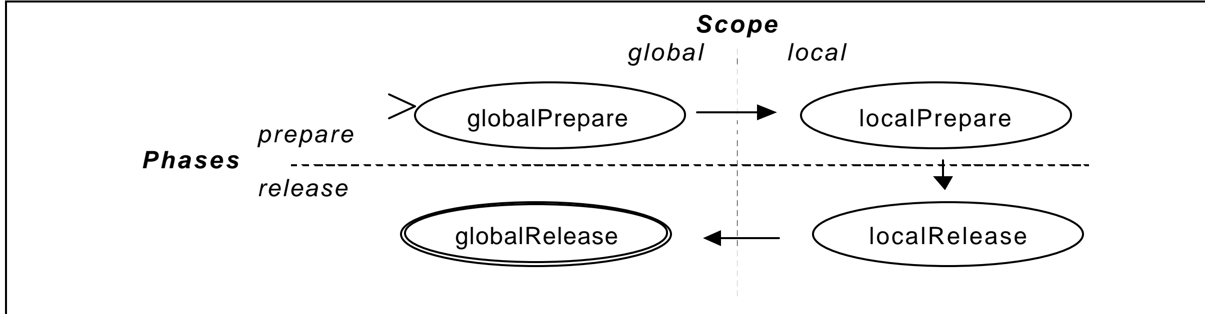

Figure 1 – Structure of Local and Global Scopes within Prepare and Release Phases

All eight plans in the Jikes RVM adhere to this overall structure. A motivating force behind MMTk was to leverage these patterns to ease the evolution of new plans. For example, division into local and global contexts in multiprocessor environments separates operations that need to be synchronized (global) from those that do not (local). Though there is always a single global state, threads on different processors can manipulate local state concurrently. This division needs to scale well across processors, and may someday be extended to include features such as dynamic join/leave of local participants.

Our experiment to date focuses on the structure of this simple FSM. Using a standard configuration for Jikes, we refactored policy related activities associated with phases and scope as an aspect within one plan. We argue that, in this form, the internal structure of each plan's use of policy becomes unpluggable and clear. It is unpluggable in this form because different combinations of policy can be specified at compile time, and clear because the internal structure associated phases, scope, and delegation can be coalesced in isolation from the rest of a given plan. By targeting these patterns, aspects can shape system elements identified to be critical during evolution.

6. Implementation Details

Figure 2 highlights three of the eight plans in Jikes (*copyMS*, *genRC* and *refCount*) and some of the contents of their four of the key abstract methods in the `StopTheWorldGC` class where plans interact with policies: *globalPrepare()*, *threadLocalPrepare()*, *threadLocalRelease()* and *globalRelease()*. In this form, similarities and differences between plans with respect to memory management policy can be more easily assessed. Some plans such as *genRC* and *refCount* in the right half of Figure 2, share the same combination of policies.

When plan is *copyMS*	When plan is *genRC* or *refCount*
and on **globalPrepare** path *copySpace* prepare *immortalSpace* prepare *markSweepSpace* prepare *treadmillSpace* prepare and on **threadLocalPrepare** path markSweepLocal prepare(*markSweepSpace)* treadmillLocal prepare(*treadmillSpace)* and on **threadLocalRelease** path markSweepLocal release(*markSweepSpace)* treadmillLocal release(*treadmillSpace*) and on **globalRelease** path *immortalSpace* release *markSweepSpace* release *treadmillSpace* release	and on **globalPrepare** path *immortalSpace* prepare *refCount* prepare and on **threadLocalPrepare** path refCountLocal prepare(*refCountSpace*) and on **threadLocalRelease** path refCountLocal prepare(*refCountSpace*) and on **globalRelease** path *immortalSpace* prepare *refCount* prepare

Figure 2 – Combinations of Policies in *copyMS, genRC* and *refCount*

Refactoring this code to re-introduce policy to plan using aspects better exposes both the symmetry between prepare/release phases and its interaction with the global/local contexts of the protocol. That is, it better captures the internal structure of policy within a given plan. Figure 3 shows the implementation of the simple FSM model for one plan, *copyMS*, using AspectJ [8].

```
package com.ibm.JikesRVM.memoryManagers.JMTk;

privileged aspect PolicyAspect {

    private int state = 0;
    private final int GLOBAL_PREPARE = 0;
    private final int LOCAL_PREPARE = 1;
    private final int LOCAL_RELEASE = 2;
    private final int GLOBAL_RELEASE = 3;

    after(Plan p):target(p) && (execution(* Plan.globalPrepare(..))
                     || execution(* Plan.threadLocalPrepare(..))
                     || execution(* Plan.threadLocalRelease(..))
                     || execution(* Plan.globalRelease(..))) {
          switch(state){
              case(GLOBAL_PREPARE):
                    CopySpace.prepare();
                    Plan.msSpace.prepare();
                    ImmortalSpace.prepare();
                    Plan.losSpace.prepare();
                    state++;
                    break;
              case(LOCAL_PREPARE):
                    p.ms.prepare();
                    p.los.prepare();
                    state++;
                    break;
              case(LOCAL_RELEASE):
                    p.ms.release();
                    p.los.release();
                    state++;
                    break;
              case(GLOBAL_RELEASE):
                    Plan.losSpace.release();
                    Plan.msSpace.release();
                    ImmortalSpace.release();
                    state = 0;
                    break;
          }
    }
}
```

Figure 3 – PolicyAspect.java

7. Adaptability: Could we do this on-the-fly?

One problem however, for this experiment and those involving other low-level software systems, is that run-time support needs to be highly generic to work with multiple different thread or process models. That is, support for language mechanisms such as that associated with control flow, or *cflow,* needs to be independent from any one definition of thread state in order to work with (and within) a range of JVMs. Techniques that exploit meta-data might be effectively applied to inspect system state and adapt the system accordingly. We believe extensive new avenues of research in VM technologies would result if recent successes in dynamic aspects, such as those presented in [2, 9], could be extended to apply to system infrastructure code as well as to the application domain [3].

8. Conclusions

Modern systems software need not be monolithic to achieve performance. As this tension between structure and performance subsides, alternative modularizations and means of extracting system state can be used to more effectively facilitate evolution and adaptation. OASIS explores the potential of aspects to naturally shape crosscutting concerns as they grow and change within system infrastructure software. Though the project is in its formative stages, preliminary results targeting domain-specific patterns have started to uncover the potential

advantages and challenges within the application of these techniques to the evolution of performance sensitive object-oriented software systems.

9. References

[1] Blackburn, S. M., Cheng, P., and McKinley, K. S. Oil and Water? High Performance Garbage Collection in Java with MMTk *ICSE 2004, 26th International Conference on Software Engineering*, (Edinburgh, Scotland, May 23-28, 2004)*(to appear)*

[2] Christoph Bockisch, Michael Haupt, Mira Mezini, Klaus Ostermann, Virtual Machine Support for Dynamic JoinPoints, Proceedings of the International Conference on Aspect-Oriented Software Development 2004.

[3] Celina Gibbs and Yvonne Coady, *Garbage Collection in Jikes: Could Dynamic Aspects Add Value?*, Dynamic Aspect Workshop, held at the International Conference on Aspect-Oriented Software Development 2004.

[4] Brian Goetz. How does garbage collection work? Developerworks, October 2003. www-106.ibm.com/developerworks/java/library/j-jtp10283/

[5] Jikes Research Virtual Machine, IBM. www-124.ibm.com/developerworks/oss/jikesrvm/

[6] Jikes Research Virtual Machine User's Guide, IBM. www-124.ibm.com/ developerworks/oss/jikesrvm/userguide/HTML/userguide.html

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, Aspect-Oriented Programming, European Conference on Object-Oriented Programming (ECOOP), 1997.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, An overview of AspectJ, Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP), 2001.

[9] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori, *A Selective, Just-In-Time Aspect Weaver*. Proceedings of the second international conference on Generative programming and component engineering, pp 189 – 208, 2003.