Introduction of Program Instrumentation using Aspects *

Morgan Deters and Ron K. Cytron
{mdeters, cytron}@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130

Abstract

Compilers and other program-analysis tools often rely on profiling information obtained from the programs they analyze. Obtaining such information can be a tedious task. The most detailed information is often obtained by arranging for code generators to instrument the code they generate or by collecting such information through instrumented interpretation. Access to code generators and interpreters is often beyond the capability of ordinary users. In this paper, we examine the advantages of using *aspects* for this purpose and present code that harvests runtime information for subsequent offline analysis.

1 Introduction

For software with real-time or small-footprint requirements, dynamic analysis[1] is becoming increasingly important. Dynamic analysis and other profiling techniques [1] allow a developer to reason about the *execution* of code rather than the code itself. Information can be harvested concerning the state of the program, the state of its memory heap, or its real-time or footprint requirements. Further, dynamic analysis is often necessary to obtain this information for programs implemented in reflective languages like Java [2], where the relevant body of code may not be known until runtime. Data gathered with dynamic analysis is often used to improve the program on later runs.

Dynamic analysis requires instrumenting the target program or its platform. But unlike adaptive techniques that are shipped as part of the final software's code, dynamic analysis concerns aren't typically part of the program itself. Any instrumentation added for this purpose must therefore be removed from the software before it is released.

Dynamic analysis is particularly useful in determining memory usage [3] and scheduling information [4], as well as object behavior in an object-oriented system. But because these aspects of a program do not occur at one particular place in the code, instrumentation is often pervasive and tends to be distributed throughout the code. In this sense, dynamic analysis cuts across the concerns of a program, requiring instrumentation through selective macros or the explicit introduction of hooks. This process is error-prone and may leave artifacts of the analysis in the code long after such analysis is needed. Further, each introduction of additional features to the code base requires concomitant additions to the instrumentation for the analysis to be complete.

A technique is needed to define analysis concerns modularly and to allow such concerns to spread automatically throughout the code. We propose the use of aspect-oriented programming (AOP) [5]—in particular the AspectJ [6] system—for this purpose. Introducing aspect code to perform the analysis makes it completely self-contained: no analysis code or hooks are present in the final, released code, and the specification of all analysis and data harvesting is relegated to a modular unit.

In this paper, we demonstrate the use of aspects in dynamic analysis and in the instrumentation of a source program to take advantage of insights learned in analysis. We present this in the context of our current research dealing with the translation of Java code into code aware of the real-time memory enhancements and constraints as provided in the Real-Time Specification for Java (RTSJ) [7].

The rest of the paper is organized as follows: Section 2 provides some background information on our ongoing research work using aspects in Java translation; Section 3 describes the two data-harvesting, dynamic-analysis aspects used in our current RTSJ translation work; Sections 3.1 and 3.2 discuss and present our *reference-probing* and *death-probing* aspects, respectively; Section 3.3 describes some offline analysis we perform on data gathered with our analysis aspects; Section 4 discusses the instrumentation of the program to use information learned during analysis; Section 5 itemizes other ideas for potential uses of dynamic analysis aspects; and Section 6 offers a few concluding remarks and planned future directions of our research.

We advise the reader that the syntax used to express aspects and storage-management directives for AspectJ and RTSJ, respectively, is based on the current version of those tools. For example, the RTSJ specification is currently undergoing revision, and the revised reference implementation has not been

* Sponsored by DARPA under contract F33615-00-C-1697

[1] Our use of the term *dynamic* analysis refers to analysis that is based upon a program's *behavior* rather than upon its instructions, the latter generally being called *static* analysis.

released as of this writing. Moreover, the current version of the AspectJ compiler (0.8b4) weaves only into source and does not easily compile the source of the standard Java class library packages.

2 Background

Our work involves the translation of Java source code into RTSJ-compliant code that takes advantage of the real-time memory capabilities of RTSJ. RTSJ programs can avoid garbage collection by defining scopes of memory (similar to memory *regions* [8, 9]) that are attached to certain scopes of execution—when the associated scope of execution exits, the associated memory scope can be reclaimed by the virtual machine. This is achieved through the `ScopeMemory` type and its subclasses, which are part of RTSJ's `MemoryArea` class hierarchy. Figure 1 shows a few classes in this hierarchy.

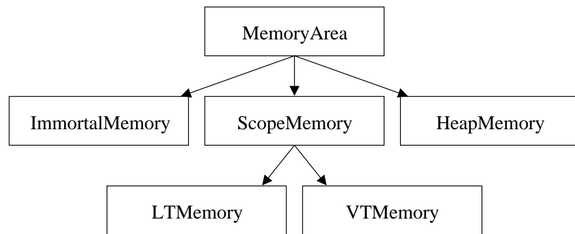

Figure 1: Part of the RTSJ `MemoryArea` class hierarchy

`ScopeMemory` types are associated with particular scopes of execution. `ScopeMemory` subclasses `LTMemory` and `VTMemory` offer a linear- and variable-time allocator, respectively. A singleton `ImmortalMemory` area provides storage that is live for the length of the entire program and never becomes collectible, and a singleton `HeapMemory` area provides a garbage-collected heap.

Objects are allocated in `ScopeMemory` areas in two ways:

1. Explicit allocation through `MemoryArea`'s `newInstance()` method

2. Through Java's standard `new` operation within an execution scope that has been associated with a `ScopeMemory` instance.

Figure 2 demonstrates these two procedures. The `createSomeObjects()` method demonstrates the use of `MemoryArea`'s `newInstance()` and `enter()` methods and the creation of a new `LTMemory` instance. First, the `LTMemory` instance is constructed with an initial and maximum size of one kilobyte. The call to `newInstance()` returns a new object of type `Foo` allocated in this `LTMemory`, and the call to `enter()` takes the given object (which must be `Runnable`) and calls its `run()` method with the `LTMemory`'s storage as its current memory area. During `run()`'s execution, all standard `new` operations allocate objects from this memory area.[2]

```
class Foo implements Runnable {
  class Bar { }

  void createSomeObjects() {
    ScopeMemory sm = new LTMemory(1024, 1024);
    // explicit allocation
    Foo f = (Foo)sm.newInstance(Foo.class);
    // entering a scope
    sm.enter(this);
  }

  // run() is called by sm.enter(this) above
  void run() {
    // b is now allocated in the LTMemory
    Bar b = new Bar();
  }
}
```

Figure 2: Ways to allocate an object in an RTSJ `ScopeMemory`

Scoped memory areas may be "nested" by associating them with nested scopes of execution. It makes sense to design such parent-child memory scope relationships when certain methods generate many intermediate objects in computing their final result—this garbage can be collected upon completion of the method's execution.

To avoid referencing dead storage, RTSJ places strict rules on references between objects in different memory scopes. An exception is thrown if an object x tries to reference an object y that is shorter-lived than x. Thus, an object in some memory area may always reference objects that are in the same memory area, or the singleton `ImmortalMemory` or `HeapMemory`. It may also reference an object in a parent (or ancestor) `ScopeMemory` area; it may not refer to an object in one of its memory area's nested `ScopeMemory` areas.

Use of the `ScopeMemory` type requires the introduction of unwieldy constructs, and can break the encapsulation of classes that under RTSJ must know about the memory requirements of other classes they reference. Therefore, it is desirable to have a modular memory component that, when introduced into a system of classes, can manage memory requirements from a central location—in effect, we wish to modularize the memory concerns of the system. We therefore turn to aspects for this purpose.

However, it is non-trivial to generate memory aspects by hand, since they must follow RTSJ's safety concerns. Further,

[2] In this example, the call to `enter()` is blocking—`run()` is executed in the caller's thread. Alternatively, a `RealtimeThread` object can be associated with a particular memory area when it is instantiated. This can be seen as a scoped execution where the entire `RealtimeThread`'s execution (concurrent with the execution of the thread that created it) is the affected execution scope.

maintaining them would require changing the large body of existing Java code. We therefore use aspects to perform two main tasks:

1. Perform the dynamic analysis necessary to determine what RTSJ memory scopes can be used in a Java program

2. Introduce advice into the original program to utilize scoped memory areas using the information learned about the program through dynamic analysis.

In this paper, we concentrate primarily on (1) above.

3 Analysis Aspects

By determining the execution-flow join points of interest and then advising those points, we can collect and act on analysis data. We have written a *reference-probing aspect* to keep track of potential inter-references between different objects and a *death-probing aspect* to determine when objects become unreachable (and therefore eligible for collection by the garbage collector).

Both analysis aspects inherit from a simple, abstract aspect (Figure 3) that defines a few useful pointcuts. `withinUs()` restrains aspects from acting on the analysis aspects themselves—this is generally not desired, but could be useful in real-time scheduling or performance analysis, where the effect of the analysis aspects is intentionally removed from timing calculations. `methodExecutions()` and `constructorCalls()` simply make the concretized analysis aspects more readable.

```
package autoscope.probe;
abstract aspect Probe {
  /* A pointcut to exclude analysis aspects from analyzing
     themselves */
  pointcut withinUs():
           within(autoscope.probe..*)
        || within(autoscope.runtime..*);

  /* Convenience pointcuts for method and constructor join
     points that interest us */
  pointcut methodExecutions():
           !withinUs() && executions(* *..*.*(..));
  pointcut constructorCalls():
           !withinUs() && calls(*..*.new(..));
}
```

Figure 3: A simple abstract probing aspect

3.1 Reference-Probing Aspect

It is an error for an RTSJ object to reference an object that is not in an ancestor's scope. We have developed a reference-probing aspect to harvest object-referencing activity from a set of program executions. Figure 4 shows a sketch of the reference-probing aspect's implementation.

We are presently concerned with grouping objects together based on the location of their instantiation—that is, the source code location of their associated `new` operation. To track the objects that are created in our system, we use "before" advice on constructor executions. This advice sets up an unresolved `SourceLocationImpl` object. When the constructor call is completed, our "after" advice on constructor calls resolves the source location of the instantiation.

We also have advice to detect when one object references another. "Around" advice captures *field assignment* join points and calls the `reference()` method, registering the reference. One or both of the objects involved in the reference may have an unresolved instantiation source location—its constructor may still be executing and, thus, our resolution advice has not yet run. In this case, the reference is kept in a stack of references that are not yet resolved. A subsequent call to `reference()` will detect that the instantiation locations have since been resolved and output the reference data.

The information thus collected can then be analyzed to determine legal scoping hierarchies for RTSJ. Section 3.3 describes the scope-forming process in more detail.

3.2 Death-Probing Aspect

In Java, one can use the `java.lang.ref.WeakReference` class to keep references to objects without inhibiting their collection. Further, one can use a `WeakReference` together with an instance of the `ReferenceQueue` class to carry out processing when the object becomes collectible in a more flexible way than is available with object finalization.

Our *death-probing aspect* uses Java's `ReferenceQueue` and `WeakReference` types to receive notification of each object's collectibility. Each method- and constructor-exit is advised with a collection cycle, and the set of such collectible objects can thus be determined. A sketch of the death-probing aspect is shown in Figure 5.

A *frame* field is introduced into `Threads` to track birth and death "times" of objects created by the thread, as well as the execution join points at which they become collectible. Using this information, we can determine how many stack frames particular objects remain live after being instantiated.

`DeathProbe` also maintains a list of references to outstanding live objects, `refs`,[3] and the `ReferenceQueue` instance. "Around" advice captures all method executions and constructor calls, maintains the frame number, and detects dead objects. "After" advice is used on constructor calls to capture the birth frame of instantiated objects and store them in

[3]These references are necessary. If a `WeakReference` object becomes itself collectible, it is never placed on the `ReferenceQueue` upon its referent's collection.

```
package autoscope.probe;
aspect ReferenceProbe extends Probe {
  protected static Hashtable newLocHash = new Hashtable();
  protected static Stack printStack = new Stack();

  /* Sets up an "unresolved" SourceLocationImpl */
  before(Object o): executions(o.new(..)) && !withinUs() {
    if(newLocHash.get(o) == null)
      newLocHash.put(o, new SourceLocationImpl());
  }
  /* Resolves a SourceLocationImpl */
  after(Object o) returning(Object o2):
                calls(o.new(..)) && !withinUs() {
    try {
      SourceLocationImpl loc =
          (SourceLocationImpl)newLocHash.get(o2);
      if(loc != null) {
        /* We don't have a SourceLocation for class
           creations outside our system (since we can't
           weave into their executions()).  We resolve
           such situations here. */
        newLocHash.put(o2,
           new SourceLocationImpl(thisStaticJoinPoint
               .getCorrespondingSourceLocation()));
      }
    } catch(ClassCastException e) { }
  }
  /* Detect references between two objects */
  before(Object x): sets(* *..*.*)[][x] && !withinUs() {
    Object a = ((FieldAccessJoinPoint)thisJoinPoint)
               .getTargetObject();
    if(x != null) {
      if(a == null)
        reference(new StaticClass(thisStaticJoinPoint
                  .getSignature().getDeclaringType()), x);
      else reference(a, x);
    }
  }
  /* Register a reference from "a" to "x" */
  static protected void reference(Object a, Object x) {
    // 1. Push a and x onto printStack, then
    // 2. Pop reference pairs off printStack and print them
    //    until a pair is found that is not resolved
  }
  static protected void flushReferences() {
    // flush remaining references from printStack
  }
  static protected String getSrcLoc(Object o) {
    // Generate a String that encodes source location
    // information
  }
  static protected void printReference(Object a,
                                       Object x) {
    // Print reference information
  }
}
```

Figure 4: A reference-probing aspect

a `DeathProbeReference`. `DeathProbeReference`, defined as a member class, extends `WeakReference` and provides for the storage of objects' birth information.

```
package autoscope.probe;
aspect DeathProbe extends Probe {
  static final ReferenceQueue Q = new ReferenceQueue();
  static final Vector refs = new Vector();

  private long Thread.frame = 0;

  around() returns Object:
        constructorCalls() || methodExecutions() {
    Thread thr = Thread.currentThread();
    ++thr.frame;
    Object retval = proceed();

    DeathProbeReference r;
    System.gc();
    System.runFinalization();
    System.out.println(thisJoinPoint);
    while((r = (DeathProbeReference)Q.poll()) != null) {
      refs.remove(r);
      // Record the death of this object
    }

    --thr.frame;
    return retval;
  }

  after() returning(Object o): constructorCalls() {
    Thread thr = Thread.currentThread();
    new DeathProbeReference(o, Q, thr, thr.frame - 1);
  }

  protected static class DeathProbeReference
                extends WeakReference {
    protected String str = null;
    protected long birthFrame;
    protected Thread birthThread;

    DeathProbeReference(Object referent, ReferenceQueue Q,
                        Thread thr, long frame) {
      super(referent, Q);

      birthFrame = frame;
      birthThread = thr;
      if(referent != null)
        str = referent.getClass().getName()
            + "@" + System.identityHashCode(referent);
      refs.add(this);
    }

    public long getBirthFrame() { return birthFrame; }
    public Thread getBirthThread() { return birthThread; }
    public String toString() { return str; }
  }
}
```

Figure 5: A death-probing aspect

3.3 Offline Analysis

We must determine two critical pieces of information from the data collected by the analysis aspects:

- The scoping hierarchy to use in the RTSJ translation
- The particular execution join points of the program that should enter particular memory scopes.

To determine the scoping hierarchy, we construct a *doesReference graph* from the reference information obtained by the reference-probing aspect, where each node of the *doesReference* graph corresponds to a well-defined set of objects in

the original program.[4] We collapse the strongly-connected components of the *doesReference* graph, as any strongly-connected object-reference chain implies that all of the participating objects must be in the same memory scope. Collapsing of strongly-connected components results in a directed acyclic graph (DAG). Because legal references may only point to objects in the same scope or longer-lived scopes, the reverse of this collapsed *doesReference* graph is a DAG representing parent-to-child scope nesting relationships.

We then select a tree over this DAG to be the memory scope hierarchy of the translated program. All references observed in dynamic-analysis runs of the program point upward in this generated scoping tree and are therefore consistent with RTSJ's reference rules.

Having decided on the particular memory scoping hierarchy to use, we can take the information gleaned from the death-probing aspect to determine the particular execution scopes—methods and constructors—on which we wish to root the discovered memory scopes.

4 Runtime Enforcement

Once a particular scoping hierarchy has been selected, and the exact join points that should signal the entrance into scopes have been determined, we must enforce the scoping strategy on the running program.

We can achieve this using AOP as well. We advise join points that should enter memory scopes to do so. "Around" advice is used, as shown in Figure 6.[5] The scope is entered by a `RealtimeThread`, which dispatches control back to the advised join point. No special treatment is required to close the scope, as that will automatically follow the execution scope's exit as guaranteed by RTSJ.

The enforcement aspect makes use of our `ScopeMap` class. `ScopeMap` implements Java's `Map` interface and maps AspectJ `JoinPoints` to `ScopeMemory` instances.

5 Other Aspect Analysis Techniques

A variety of data could be collected with aspect-characterized dynamic analysis:

- *Instantiation and collection behavior* We operate in this realm of analysis, but other possibilities exist. A developer may be interested in knowing the maximum amount of live storage in the system at one time, or the maximum number of live objects. Thread-based analysis could be performed to determine the amount of work done by each thread in a thread pool.

- *Inter-class behavior* This paper examines inter-class references, but inter-class call behavior could be detected also. Particularly high method-execution frequencies could be detected to indicate the most important places to optimize or inline code.

- *Dead code removal* Method and constructor implementations never used by a running system can be detected and replaced by a no-op in bytecode files to reduce the size of the software.

- *Class preloading* Classes used by an embedded system application could be determined and either preloaded or placed in flash memory in the running system to buffer against unexpected pauses during runtime. In some circumstances it can be impossible to determine all loadable classes statically (*e.g.*, in a reflective language like Java).

Of course, these are just a few possibilities among a large number of potential applications.

[4] As mentioned in Section 3.1, we are currently grouping objects together into *doesReference* nodes based on the source code location of their construction—their associated `new` operation. In the future, we will be investigating more flexible and adaptive ways of grouping objects into *doesReference* nodes.

[5] Of course, the actual structure of the generated memory aspects will depend on the analyzed program.

```
package autoscope.runtime;
aspect EnforcerAspect {
  ScopeMap scopes = new ScopeMap();
  void enterScope(JoinPoint jp, Runnable logic) {
    scopes.getScope(jp).enter(logic);
  }

  around() returns Object:
        executions(public void Foo.foo()) {
    Runnable r = new Runnable() {
        public Object retval;
        public void run() {
            retval = proceed();
        }
      };
    // The run() method of "r" executes in our thread,
    // not its own.
    enterScope(thisStaticJoinPoint, thr);
    // Scope no longer exists after call to enterScope()
    return thr.retval;
  }

  // etc.
}
```

Figure 6: A sample enforcement aspect

6 Conclusion

We have demonstrated the suitability of aspects for dynamic analysis and for the application of insights gleaned from analysis. While our examples have been limited chiefly to the collection of information processed in offline analysis, aspects are sufficiently expressive to allow for program analysis and

optimization at runtime. As such, more powerful, adaptive dynamic analysis [10] could be performed during runtime.

Our future work in this area will focus on the characterization of dynamic analysis and related aspect design patterns. While these dynamic-analysis concerns are often orthogonal, they may not be in the case of real-time schedule feasibility (aspect advice requires time to execute), so the composition of dynamic analysis into the same run of the program also warrants future research.

References

[1] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1992.

[2] J. Gosling, B. Joy, and G. Steele, *The Java Programming Language Specification*. Reading, Massachusetts: Addison-Wesley, 1996.

[3] D. J. Cannarozzi, M. P. Plezbert, and R. K. Cytron, "Contaminated garbage collection," *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pp. 264–273, 2000.

[4] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, vol. 20, March 2001.

[5] G. Kiczales, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[6] The AspectJ Organization, "Aspect-Oriented Programming for Java." `www.aspectj.org`, 2001.

[7] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[8] D. Gay and A. Aiken, "Language support for regions," in *Proceedings of ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)*, (Snowbird, Utah), pp. 70–80, ACM, May 2001.

[9] M. Tofte and J.-P. Talpin, "Region-based memory management," *Information and Computation*, vol. 132, pp. 109–176, February 1997.

[10] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek, "'C and tcc: A language and compiler for dynamic code generation," *ACM Transactions on Programming Languages and Systems*, vol. 21, pp. 324–369, March 1999.