SOFTWARE—PRACTICE AND EXPERIENCE

Softw. Pract. Exper. 2005; **00**:1 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Profiling with AspectJ

David J. Pearce[1], Matthew Webster[2], Robert Berry[2] and Paul H.J. Kelly[3]

[1]*School of and Mathematics, Statistics and Computer Science, Victoria University of Wellington, NZ. Email: david.pearce@mcs.vuw.ac.nz. Tel: +64 (0)44635833.*
[2]*IBM Corporation, Hursley Park, Winchester, UK*
[3]*Department of Computing, Imperial College, London, UK*

SUMMARY

This paper investigates whether AspectJ can be used for efficient profiling of Java programs. Profiling differs from other applications of AOP (e.g. tracing), since it necessitates efficient and often complex interactions with the target program. As such, it was uncertain whether AspectJ could achieve this goal. Therefore, we investigate four common profiling problems (heap usage, object lifetime, wasted time and time-spent) and report on how well AspectJ handles them. For each, we provide an efficient implementation, discuss any trade-offs or limitations and present the results of an experimental evaluation into the costs of using it. Our conclusions are mixed. On the one hand, we find that AspectJ is sufficiently expressive to describe the four profiling problems and reasonably efficient in most cases. On the other hand, we find several limitations with the current AspectJ implementation that severely hamper its suitability for profiling.

KEY WORDS: AspectJ, AOP, Java, Profiling, Performance

1. INTRODUCTION

Profiling program behaviour is a common technique for identifying performance problems caused by, for example, inefficient algorithms, excessive heap usage or synchronisation. Profiling can be formalised as the collection and interpretation of program events and is a well-understood problem with a significant body of previous work. However, one area of this field has been largely unexplored

Received August 2005
Revised April 2006

in the past: *effective deployment*. That is, given a program, how can it be easily profiled in the desired manner? In some situations, this is relatively straightforward because the underlying hardware provides support. For example, time profiling can be implemented using a timer interrupt to give periodic access to the program state. Alternatively, hardware performance counters can be used to profile events such as cache misses, cycles executed and more [1, 2]. The difficulty arises when there is no hardware support for the events of interest. In this case, *instrumentation code* must be added and various strategies are used to do this. For example, `gprof` — perhaps the most widely used profiler — relies upon specific support from `gcc` to insert instrumentation at the start of each method [3]. Unfortunately, it is very difficult to capitalise on this infrastructure for general purpose profiling simply because `gcc` has no mechanism for directing where the instrumentation should be placed.

In a similar vein, binary rewriters (e.g. [4, 5]) or program transformation systems (e.g. [6, 7]) can help automate the process of adding instrumentation. While these tools do enable profiling, they are cumbersome to use since they operate at a low level. For example, binary rewriters provide only simple interfaces for program manipulation and, hence, code must still be written to apply the instrumentation. Likewise, program transformation tools operate on the abstract syntax tree and require the user provide complex rewrite rules to enable instrumentation. In a sense, these tools are too general to provide an easy solution to the profiling problem. What is needed is a simple and flexible mechanism for succinctly specifying how and where instrumentation should be deployed.

One solution is to provide support for profiling through a general purpose virtual machine interface. For example, the *Java Virtual Machine Profiler Interface (JVMPI)* enables several different types of profiling [8]. However, there are some drawbacks: firstly, it is a fixed interface and, as such, can only enable *predefined* types of profiling; secondly, enabling the JVMPI often dramatically reduces performance. The *Java Virtual Machine Tool Interface (JVMTI)* replaces the JVMPI in Java 1.5 and attempts to address both of these points [9]. However, as we will see in Section 8.3, this comes at a cost — the JVMTI no longer supports profiling directly. Instead, it simply enables the manipulation of Java bytecodes at runtime, placing the burden of performing the manipulation itself on the user.

An alternative to this has recently become possible with the advent of *Aspect Oriented Programming (AOP)* — a paradigm introduced by Kiczales *et al.* [10]. In this case, the programmer specifies in the AOP language how and where to place the instrumentation, while the language compiler/runtime takes care of its deployment. This does not require special support (e.g. from the JVM), as the target program is modified directly. However, very few works have considered AOP in the context of profiling. Therefore, we address this by taking the most successful AOP language, namely AspectJ, and evaluating whether it is an effective tool for profiling. We do this by selecting four common profiling problems and investigating whether they can be implemented in AspectJ or not. We also examine what performance can be expected in practice from the current AspectJ implementation, as this is critical to the adoption of AspectJ by the profiling community. Our reasoning is that the outcome of this provides some evidence as to whether AspectJ is suitable for general purpose profiling or not. For example, if we could not implement these straight-forward cases, we would have little hope that other, more complex types of profiling were possible. Likewise, if we were able to implement them, but the performance was poor, this would indicate AspectJ was not yet ready for the profiling community.

The outcome of our investigation is somewhat mixed. We find that, while the language itself can express the profiling examples we consider, several limitations with the current AspectJ implementation prevent us from generating results comparable with other profilers (such as `hprof`). As such, we believe these must be addressed before AspectJ can be considered a serious platform for profiling. Specifically, the main contributions of this paper are as follows:

1. We investigate AspectJ as a profiling tool — both in terms of performance and descriptive ability. This is done by evaluating four case studies across 10 benchmarks, including 6 from SPECjvm98.

2. We present novel techniques, along with source code, for profiling heap usage, object lifetime, wasted time and time-spent with AspectJ.

3. We identify several issues with the current AspectJ implementation which prohibit complete implementations of our profiling case-studies.

Throughout this paper we use the term "AspectJ" to refer to the language itself, whilst "AspectJ implementation" refers to the AspectJ implementation available from `http://www.aspectj.org` (since this is very much the standard implementation at this time).

The remainder is organised as follows. Section 2 provides a brief introduction to AspectJ. Sections 3, 4, 5 and 6 develop AspectJ solutions for profiling heap usage, object lifetime, wasted time and time-spent respectively. After this, Section 7 presents the results of an experimental evaluation into the cost of using them. Section 8 discusses related work and, finally, Section 9 concludes.

2. INTRODUCTION TO ASPECTJ

In this section, we briefly review those AspectJ constructs relevant to this work. For a more complete examination of the language, the reader should consult one of the well-known texts (e.g. [11, 12]). AspectJ is a language extension to Java allowing new functionality to be systematically added to an existing program. To this end, AspectJ provides several language constructs for describing where the program should be modified and in what way. The conceptual idea is that, as a program executes, it triggers certain events and AspectJ allows us to introduce new code immediately before or after these points. Under AOP terminology, an event is referred to as a *join point*, whilst the introduced code is called *advice*. The different join points supported by AspectJ include method execution, method call and field access (read or write). We can attach advice to a single join point or to a set of join points by designating them with a *pointcut*. The following example, which profiles the number of calls to `MyClass.toString()` versus those to any `toString()` method, illustrates the syntax:

```
aspect ToStringCountingAspect {
 private int totalCount = 0;
 private int myCount = 0;

 pointcut myCall() : call(String MyClass.toString());
 pointcut allCalls() : call(String *.toString());

 before(): myCall() { myCount++; }
 after() : allCalls() { totalCount++; }
}
```

This creates two pointcuts, `myCall()` and `allCalls()`, which respectively describe the act of calling `MyClass.toString()` and `toString()` on any class. They are each associated with advice which is executed whenever a matching join point is triggered. Here, `before()` signals the advice should be executed just before the join point triggers, while `after()` signals it should be executed immediately afterwards (although it makes no difference which is used in this example). The advice is wrapped inside an `aspect` which performs a similar role to the class construct. Aspects permit inheritance, polymorphism and implementation hiding. When the aspect is composed with a Java program — a process known as *weaving* — the program behaviour is changed such that `myCount` is incremented whenever `MyClass.toString()` is called. Likewise, `totalCount` is incremented whenever any `toString()` method is called (including `MyClass.toString()`). Note, the current AspectJ implementation does not alter the program's source code, rather the change is seen in its generated bytecode. A problem can arise with a pointcut that matches something inside the aspect itself as this can cause an infinite loop, where the aspect continually triggers itself. This would happen, for example, if our aspect had a `toString()` method that was called from the after advice. To overcome this, we can specify that a class `C` should not be advised by including `!within(C)` in the pointcut definition.

Another interesting issue is determining which particular join point triggered the execution of some advice. For this purpose, AspectJ provides a variable called `thisJoinPoint` which is similar in spirit to the `this` variable found in OOP. It refers to an instance of `JoinPoint` which contains both *static* and *dynamic* information unique to the join point in question. Here, static information includes method name, class name and type information, while dynamic information includes parameter values, virtual call targets and field values. To provide the dynamic information, the AspectJ implementation creates a fresh instance of `JoinPoint` every time a join point is triggered, passing it to the advice as a hidden parameter (much like the `this` variable). For efficiency reasons, this is only done if the advice actually references it. For the static information, the AspectJ implementation constructs an instance of `JoinPoint.StaticPart` which is retained for the duration of the program's execution. This can be accessed through either the `thisJoinPoint.StaticPart` or the `thisJoinPointStaticPart` variables. The latter is preferred as it allows greater optimisation. We can alter our example in the following way to record the total number of calls to `toString()` on a per-class basis, rather than lumping them all together:

```
private Map totalCounts = new HashMap();
...
after() : allCalls() {
 Class c = thisJPSP.getSignature().getDeclaringType();
 Integer i = totalCounts.get(c);
 if(i != null) totalCounts.put(c,new Integer(i.intValue()+1));
 else totalCounts.put(c,new Integer(1));
}
```

Note, `thisJPSP` is an abbreviation for `thisJoinPointStaticPart` and is used throughout the remainder of this paper to improve the presentation of our code examples. Also, `totalCounts` replaces `totalCount` from before. It can also be useful to access information about the *enclosing* join point. That is, the join point whose scope encloses that triggering the advice. For example, the enclosing join point of a method call is the method execution containing it and AspectJ provides `thisEnclosingJoinPoint` to access its `JoinPoint` object. The corresponding static component is accessed via `thisEnclosingJoinPointStaticPart` (henceforth `thisEJPSP`).

The final AspectJ feature of relevance is the *inter-type declaration*, which gives the ability to define new fields or methods for existing classes and/or to alter the class hierarchy. For example, the following alters `MyClass` to implement the `Comparable` interface:

```
aspect ComparableMyClass {
 declare parents: MyClass implements Comparable;
 int MyClass.compareTo(Object o) { return 0; }
}
```

This first declares that `MyClass` implements `Comparable` and, second, defines the required `compareTo()` method (which effectively adds this method to `MyClass`). Note, if `MyClass` already had a `compareTo()` method, then weaving this aspect would give a weave-time error.

3. PROFILING HEAP USAGE

In this section, we investigate AspectJ as a tool for determining which methods allocate the most heap storage. The general idea is to advise all calls to `new` with code recording the size of the allocated object. This amount is then added to a running total for the *enclosing* method (of that call) to yield exact results:

```
aspect HeapProfiler {
 Hashtable totals = new Hashtable();

 before() : call(*.new(..)) && !within(HeapProfiler) {
  MutInteger tot = getTotal(thisEJPSP);
  Class c = thisJPSP.getSignature().getDeclaringType();
  if(c.isArray()) {
   Object[] ds = thisJoinPoint.getArgs(); // dims for array
   tot.value += sizeof(c,ds);
  } else {
   tot.value += sizeof(c);
 }}

 MutInteger getTotal(Object k) {
  MutInteger s = (MutInteger) totals.get(k);
  if(s == null) {
   s = new MutInteger(0);
   totals.put(k,s);
  }
  return s;
 }
 int sizeof(Class c, Object arrayDims...) { ... }
}
```

Here, `sizeof()` computes the size of an object and, for now, assume it behaves as expected — we discuss its implementation later. Also, `getTotal()` maps each method to its accumulated total. Notice that, since the `JoinPoint.StaticPart` object given by `thisEJPSP` uniquely identifies the enclosing method, it can be used as the key. Furthermore, `getTotal()` is implemented with a `Hashtable` to provide synchronised access, although more advanced containers (e.g. `ConcurrentHashMap`) could be used here. The use of `!within(HeapProfiler)` is crucial as it prevents the advice from being applied to code within the aspect itself. Without this, an infinite loop can arise with the advice being repeatedly triggered in `getTotal()`. Notice that we have used an *anonymous* pointcut to specify where the advice should be applied, instead of an explicit designation via the `pointcut` keyword. The purpose of distinguishing between the creation of arrays and other objects is subtle. The key point is that, to determine the size of an array, we need its dimensions and AspectJ gives access to these via `getArgs()` since the arguments of an array constructor are its dimensions (Section 3.1 discusses this in more detail). As an optimisation, we call `getArgs()` only on array types, since this requires accessing the `JoinPoint` object (which is created lazily in the current AspectJ implementation). Finally, the `MutInteger` class is similar to `java.lang.Integer`, except that its value can be updated.

A subtle aspect of our approach is that bytes allocated *inside an object's constructor* are not attributed to the enclosing method creating it. Instead, they are attributed to the constructor itself and, to see that this makes sense, consider the following:

```
1. class T { T() { for(...) new X(); }}
2. int foo() { T x = new T(); }
```

This example, while perhaps somewhat contrived, highlights an important point: *if T's constructor allocates a lot of unnecessary storage, who is to blame?* By including bytes allocated by `T()` in `foo()`'s total, we are misdirecting optimisation efforts toward `foo()` rather than `T()`. Of course, we could devise situations where the problem stems from `foo()` calling `T()` too often. In this case, the *inclusive* approach seems to make more sense, since it focuses attention toward `foo()`. However, this is misleading as it is really a fundamentally different problem regarding call frequency. For example, `foo()` may call `T()` frequently because it is itself called frequently and neither will catch this. Furthermore, while both approaches could be extended to catch problems relating to call frequency, the *inclusive* approach could never catch the example highlighted above.

We now identify our first limitation with the current AspectJ implementation which affects the precision of our scheme. The issue is that the pointcut `call(*.new(..))` does not catch allocations of array objects — meaning they are not included in the heap measurements. In fact, a fix for this issue has been recently included in the AspectJ implementation (as a direct result of this work), although it is not currently activated by default (the command-line switch "`-Xjoinpoints:arrayconstruction`" is required).

3.1. IMPLEMENTING SIZEOF

At this juncture, we must briefly discuss our implementation of `sizeof` as Java does not provide this primitive. To determine an object's size, we iterate every field of its class (using reflection) and total their sizes. Of course, this is an estimate since alignment/packing issues are ignored, the size of references is unknown (we assume 4 bytes) and the object header size is also unknown (we assume 8 bytes). Also, we do not traverse references and accumulate the size of the objects they target, since only the bytes allocated by the current call to `new` are relevant (and the objects targeted by such fields must have been allocated previously). For arrays, the innermost dimension is calculated using the type held by the array, whilst the outer dimensions are assumed to be arrays of references to arrays (the dimensions themselves being obtained from the join point, as discussed previously). Again, we do not traverse the references of objects held by the innermost dimension since an array cannot be populated until after being created. To improve performance (as reflection is notoriously slow), we also employ a `Hashtable` to cache results and make subsequent requests for the same type cheaper.

One issue with this implementation is that a `Hashtable` lookup is needed to access cached type sizes. If we could eliminate this, the cost of using our heap profiling aspect might be reduced. AspectJ version 1.5.0 introduced a new primitive, `pertypewithin(..)`, which makes this possible. This allows us to specify that a separate aspect instance should be instantiated for every type matching a given type pattern. For example:

```
aspect TestAspect pertypewithin(mypkg..*) {
 ...
}
```

This results in a separate instance of `TestAspect` (created lazily) for every class within the package `mypkg`, rather than just a single instance of `TestAspect` being created (as for normal aspects). Thus, `pertypewithin` allows us to associate state (in our case, `sizeof` information) with a type. The information associated with a type `C` is stored as a `static` class variable of `C`. This (in theory at least) allows us to access cached `sizeof` information using a static field lookup, rather than a `Hashtable` lookup. Unfortunately, we find in practice that using `pertypewithin` to implement the cache actually gives worse performance that using a `Hashtable`. The reason for this appears to be that, although the information is stored in a static field, the current AspectJ implementation accesses it via a reflective call. We expect future optimisation of the AspectJ implementation will eliminate this overhead, leading to better performance of `pertypewithin`. A complete implementation of `sizeof` using `pertypewithin` is given in Appendix B for reference.

4. PROFILING OBJECT LIFETIME

In this section, we look at profiling object lifetime, where the aim is to identify which allocation sites generate the longest-living objects. This can be used, for example, to help find memory leaks as long-lived objects are candidates [13]. Another application is in generational garbage collectors, where it is desirable to place long lived objects immediately into older generations, often known as *pretenuring* (see e.g. [14, 15]).

As we have already demonstrated how allocation sites can be instrumented with AspectJ, the remaining difficulty lies in developing a notification mechanism for object death. In Java there are two obvious constructs to use: *weak references* and *finalizers*. An implementation based on the latter would rely upon introducing special finalizers for all known objects to signal their death. This poses a problem as introducing a method `foo()` into a class which already has a `foo()` is an error in AspectJ. To get around this, we could *manually* specify which classes need finalizers introduced into them (i.e. all those which don't already have them) with a pointcut. At which point, we could advise all finalizers to signal object death. Note that, while the process of determining which classes don't have finalizers could be automated, this cannot be done within AspectJ itself making this approach rather inelegant. In light of this, we choose weak references and, indeed, they have been used for this purpose before [14].

```
aspect lifetimeProfiler {
 static int counter = 0;
 static int period = 100;
 static Set R; static Monitor M;
 static ReferenceQueue Q;

 after() returning(Object o) : call(*.new(..)) &&
    if(++counter >= period) && !within(lifetimeProfiler) {
   MyRef mr = new MyRef(thisJPSP, System.currentTimeMillis(),o,Q);
   R.add(mr);
   counter = 0;
 }

 lifetimeProfiler() {
  HashSet tmp = new HashSet();
  R = Collections.synchronizedSet(tmp);
  Q = new ReferenceQueue();
  M = new Monitor(); M.start();
 }

 class MyRef extends PhantomReference {
  public JoinPoint.StaticPart sjp;
  public long creationTime;

  MyRef(JoinPoint.StaticPart s, long c, Object o, ReferenceQueue q) {
   super(o,q); sjp = s; creationTime = c;
 }}

 class Monitor extends Thread {
  public void run() { while(true) { try {
    MyRef mr = (MyRef) Q.remove();
    R.remove(mr);
    long age = System.currentTimeMillis() - mr.creationTime;
    getSample(mr.sjp).log(age);
   } catch(InterruptedException e) {
 }}}}

 class AvgSample {
  double avg = 0; int num = 0;
  public void log(long v) { avg = ((avg * num) + v) / ++num; }
 }
 AvgSample getSample(Object k) { ... }}
```

Figure 1. The outline of our lifetime profiler aspect. The key feature is the `after() returning(..)` notation, which gives access to the newly allocated object returned by `new`. The advice then attaches an extended phantom reference containing the creation time and allocation site. When an object dies, its reference is removed from `R` by the `Monitor` and its lifetime logged. Here, `getSample()` is similar to `getTotal()` from before. `AvgSample` is used to maintain the average lifetime of all objects created at a given allocation site. Additional code is needed to catch immortal objects: on program termination this would iterate through `R` to identify and log the lifetime of any unclaimed objects. Finally, counter-based sampling is used to reduce the number of objects being tracked. This lowers overhead and causes less perturbation on the target program.

The `java.lang.ref` package was introduced to give a degree of control over garbage collection and provides three types of weak reference. These share the characteristic that they do not prevent the referenced object, called the *referent*, from being collected. That is, if the only references to an object are weak, it is open to collection. The different weak reference types provide some leverage over when this happens:

1. *Soft references*. The garbage collector must free softly referenced objects before throwing an `OutOfMemoryError` exception. Thus, they are useful for caches, whose contents should stay for as long as possible. Soft references are "cleared" (i.e. set to `null`) before finalisation, so their referents can no longer be accessed.
2. *Weak references*. Their referents are always reclaimed at the earliest convenience. Weak references are also "cleared" before finalisation.
3. *Phantom references*. Again, phantomly referenced objects are always reclaimed at the earliest convenience. However, they are not "cleared" until after finalisation.

To see which is best suited to our purpose, we must understand the relevance of clearing. When creating a reference, we can (optionally) indicate a `ReferenceQueue` onto which it will be placed (by the garbage collector) when cleared. Thus, this is a form of callback mechanism, allowing notification of when the reference is cleared. Note, if it was not cleared before placed on the queue, our application could *resurrect* it by adding new references. In fact, objects are not truly dead until after finalisation because their finalizer can resurrect them [14]. From these facts, it follows that phantom references give the most accurate indication of object lifetime.

The basic outline of our scheme is now becoming clear: at object creation, we attach a phantom reference and record a timestamp and an identifier for the allocation site. The phantom reference is associated with a global reference queue, monitored by a daemon thread. This is made efficient by `ReferenceQueue.remove()`, which sleeps until a reference is enqueued. Thus, when an object is garbage collected, the daemon thread is awoken (by the reference queue) to record the time of death and, hence, compute the object's lifetime. Figure 1 provides the core of our implementation.

5. PROFILING WASTED TIME

In the last section, we developed a technique for profiling object lifetime. In fact, we can go beyond this by breaking up the lifetime into its *Lag, Drag* and *Use* phases [16]. Under this terminology, *lag* is the time between creation and first use, *drag* is that between last use and collection, while *use* covers the rest. Thus, we regard lag and drag as wasted time and the aim is to identify which allocation sites waste the most.

An important question is what it means for an object to be *used*. In this work, we consider an object is used when either of the following occurs: a public, non-static method is called; or a public, non-static field is read or written. We ignore read/writes to private fields and methods, since these must have arisen from a call to a public method, in which case the object use has been registered. Methods which run for a long period of time updating the internal state of some object may cause imprecision if there is sufficient difference between the time of method entry and the actual last use. This is really a trade-off as, by ignoring changes to the internal state of an object, the profiling data associated with it needs to be updated less frequently, leading to greater performance in practice. We wanted a more complex definition of object use, which additionally ignored changes to public fields from within the object's own methods. As it turned out, we could not express this constraint efficiently in AspectJ.

The main difficulty in this endeavour actually lies in efficiently associating state with an object. Here, the state consists of timestamps for the first and last use which, upon object death, can be used to determine *lag*, *drag* and *use*. This state is updated by advice associated with the get/set join points as the program proceeds. As such advice will be executed frequently, access to the state must be as fast as possible. We considered, when embarking upon this project, that there should be three possible approaches:

1. *Using a Map*. This is the simplest solution — state is associated with each object using a HashMap (or similar). The downside, of course, is the need to perform a lookup on every field access (which is expensive).

2. *Member Introduction*. In this case, we physically add new fields to every object to hold the state using *member introduction*. The advantage is constant time access, while the disadvantage is an increase in the size of every object.
3. *Using pertarget*. The `pertarget` specifier is designed for this situation. It indicates that one aspect should be created for every object instance, instead of using a singleton (which is the norm). Again, a disadvantage is that every object is larger.

The issue of increasing object size is important as it reduces the advantages of sampling — where the aim is to reduce overheads by monitoring only a few objects, rather than all. In particular, sampling should dramatically reduce the amount of additional heap storage needed, but this is clearly impossible if the size of *every* object must be increased. Now, approach 3 gives something like:

```
aspect ptWaste pertarget(call(*.new(..))){
 State theState = new State();
 before(Object o) : target(o) && (set(public !static * *.*) ||
                                  get(public !static * *.*) ||
                                  call(public !static * *.*(..))) {
  ...
}}
```

The `pertarget(X)` specifier declares that a separate instance of the aspect should be created for every object that is the target of the join points identified by `X`. Thus, a separate instance of the `ptWaste` aspect will be created for every constructible object. Each would be created the first time its corresponding object is the target of some invoked advice. This allows `theState` to be shared between invocations of advice on the same object. We have already seen that the `call` join point captures method invocation. In this case, we have annotated it to specify that only public, non-static methods should be captured. Likewise, the `get`/`set` join points capture all public, non-static field read / writes. Thus, these join points taken together define what it means for an object to be used. Unfortunately, this approach of using `pertarget` fails as there are no valid target objects for a `call(*.new(..))` — meaning the `pertarget(call(*.new(..)))` specifier does not match anything. This arises because the target object is not considered to exist until after the `call(*.new(..))` pointcut. Using other pointcuts for the `pertarget(...)` specifier (such as `get(), set()` and `call()`) does not help because these will not match objects which are created but not used. This constitutes our second limitation with the current AspectJ implementation. As this behaviour was intentional, it is perhaps more significant than the others identified so far. In particular, it remains uncertain whether or not it can be resolved.

The second approach, which uses *Inter-Type Declarations (ITD)*, provides a manual implementation of the above:

```
1. aspect itdWaste {
2.  private interface WI { }
3.  declare parents: * && !java.lang.Object implements WI;
4.  State WI.theState = new State();
5.  before(Object o): target(o) &&
6.                       (set(public !static * *.*) ||
7.                        get(public !static * *.*) ||
8.                        call(public !static * *.*(..))) {
9.  if(o instanceof WI)
9.   WI w = (WI) o;
10.    ... // access WI.thState directly
11. } else {
12.    ... // use map
13.}}
```

Here, line 3 is an ITD which declares every class to implement interface WI (except `java.lang.Object`, as this is prohibited by the current AspectJ implementation), while line 4 introduces `theState` into WI. The effect of all this is to introduce a new instance variable `theState` into every user-defined class in the class hierarchy (see Section 7.1 for more on why only user-defined classes are affected). This ensures that every corresponding object has exactly one copy[*] of `theState` and, through this, we can associate each object with a unique instance of `State`. Only user-defined classes are affected by the ITD because, in practice, classes in the standard library cannot be altered using the current AspectJ implementation (Section 7.1 discusses the reasons for this in more detail). The pointcut for the advice matches all uses (including method invocation) of any object. In the case of a user-defined object (i.e. an object implementing WI), we obtain constant-time access to `theState`

[*]Note, AspectJ does not introduce a field F into a class whose supertype is also a target for the introduction of F. Thus, an instance of any class can have at most one copy of F, rather than potentially one for every supertype in its class hierarchy.

(since it is a field). For other objects, we use a map to associate the necessary state as a fall back (this is outlined in more detail below) . To complete the design, we must also advise all `new` calls to record creation time and include our technique from the previous section for catching object death.

We now consider our final design, which uses a map to associate state with each object. A key difficulty is that the map must not prevent an object from being garbage collected. Thus, we use weak references to prevent this, which adds additional overhead. The main body of our implementation is detailed in Figure 2 and the reader should find it similar to those discussed so far. Note the use of sampling to reduce the number of objects being tracked. This improves space consumption as fewer state objects are instantiated, although it has little impact upon runtime overhead. In fact, our ITD implementation also uses sampling for this reason, although it must still pay the cost of an extra word per user-defined object (for `theState`).

The astute reader may notice something slightly odd about our implementation of Figure 2 — it contains a bug! The problem is subtle and manifests itself only when the target program contains objects with user-defined `hashCode()` implementations that read/write public fields. It arises because `WeakKey` invokes an object's `HashCode()` method, which is needed to ensure different `WeakKey`s referring to the same object match in the `Hashtable`. This invocation will correspond to a use of the object if `hashCode()` reads/writes public fields. The invocation itself is not a use, since it occurs within `WasteAspect` and this is explicitly discounted using `!within(..)`. The problem causes an infinite loop where looking up the state associated with an object is a use of it, which triggers the `before()` advice, which again tries to lookup the state and so on. To get around this is not trivial. We cannot use an alternative map, such as `TreeMap`, since this uses the object's `compareTo()` method, leading to the same problem. We could, however, employ AspectJ's `cflow()` construct to include in our definition of an object use the constraint that a method within `WasteAspect` cannot be on the call stack. Unfortunately, this would almost certainly impose a large performance penalty [17]. Thus, we choose simply to acknowledge this problem, rather than resolving it, since it is unlikely to occur in practice.

```
public final aspect WasteAspect {
 static int counter = 0;
 static int period = 100;
 static Map R = new Hashtable();
 static ReferenceQueue Q = new ReferenceQueue();

 after() returning(Object newObject) : call(*.new(..))
         && !within(WasteAspect) && if(++counter >= period) {
  WeakKey wk = new WeakKey(newObject,Q);
  R.put(wk,new State(thisJPSP, System.currentTimeMillis()));
  counter = 0;
 }

 before(Object o) : target(o) && !within(WasteAspect) && (
     call(public !static * *.*(..)) ||
     set(public !static * *.*) || get(public !static * *.*)) {
  Object t = R.get(new WeakKey(o));
  if(t != null) {
   State s = (State) t;
   s.lastUse = System.currentTimeMillis();
   if(s.firstUse == -1) { s.firstUse = s.lastUse; }
 }}

 class WeakKey extends WeakReference {
  int hash;
  WeakKey(Object o) { super(o); hash = o.hashCode(); }
  WeakKey(Object o, ReferenceQueue q) { super(o,q); hash = o.hashCode();}
  public int hashCode() { return hash; }
  public boolean equals(Object o) {
   if (this == o) return true;
   Object t = this.get();
   Object u = ((WeakKey) o).get();
   if ((t == null) || (u == null)) { return false; }
   return t == u;
 }

 private final class State {
  long lastUse,firstUse = -1;
  long creationTime;
  JoinPoint.StaticPart sjp = null;
  State(JoinPoint.StaticPart s, long c) { creationTime = c; sjp = s; }
 }}
```

Figure 2. The core parts of an aspect for profiling wasted time. The key features are the `Hashtable` which associates state with an object and the use of counter-based sampling to reduce overhead. Note that, while sampling does help reduce storage, it does not prevent a table look up on each field access. To complete this design, a daemon thread must monitor `Q` to catch object death and log usage information, as for Figure 1. Finally, `WeakKey.equals()` deals with two awkward problems: firstly, its hashcode must be identical for identical referents to allow correct look up from the `after()` advice; secondly, look up must still be possible after the referent is cleared.

6. PROFILING TIME SPENT

In this section, we consider time sampling, where the idea is to periodically log the currently executing method. Thus, on termination, those methods with the most samples (i.e. logs) accrued are considered to be where most time was spent. Our approach is to track the currently executing method with AspectJ, so it can be periodically sampled by a daemon thread. This is done by updating a global variable whenever a method is entered (through normal entry/call return) or left (through normal exit/calling another):

```
aspect CurrentMethodAspect {
 static JoinPoint.StaticPart current;
 before() : (execution(* *.*(..))  || execution(*.new(..)))
            && !within(CurrentMethodAspect) {
   current = thisJPSP;
 }

 after() returning : (execution(* *.*(..)) || execution(*.new(..)))
                     && !within(CurrentMethodAspect) {
   current = null;
 }

 before() : (call(* *.*(..)) || call(*.new(..)))
            && !within(CurrentMethodAspect){
   current = null;
 }

 after() returning : (call(* *.*(..)) || call(*.new(..)))
                     && !within(CurrentMethodAspect) {
   current = thisEJPSP;
 }}
```

Here, the unique `JoinPoint.StaticPart` object is used to identify the currently executing method. Notice that `current` is assigned to null when a method is left. This may seem redundant, since it will be overwritten as soon as the next method is (re-)entered. Indeed, if we could guarantee that all methods were advised, this would be the case. Unfortunately, we cannot necessarily make this guarantee for reasons discussed in Section 7.1. With regard to multithreading, our approach can be inaccurate as, following a context switch, a sample may be taken before `current` is updated by the newly executing method. This results in time being incorrectly charged to the method which was running before the switch. Following [18], we argue that this is not a serious cause for concern as sampling is inexact anyway and it is unlikely that such behaviour would consistently affect the same method. Certainly, synchronizing on `current` would cause greater problems and making it a thread local means a thread lookup before/after every method call/execution. Thus, we choose against either of these on the grounds of efficiency.

Another interesting point is the use of `after() returning`, instead of just `after()` advice. The former only catches normal return from a method, whilst the latter also catches thrown exceptions. Our reason then, for choosing `after() returning` is that we have observed it offers better performance (up to 20% in some cases), while the issues of missing return by exception seem negligible. Note, if this were considered important, then `after()` could simply be used in place of `after() returning` to ensure `current` was updated correctly after an exception.

7. EXPERIMENTAL RESULTS

In this section, we present and discuss the results of an experimental evaluation into the costs of using the profiling aspects developed in the previous sections. We also introduce `djprof`, a command-line tool which packages these aspects together so they can be used without knowledge of AspectJ. This was used to generate the results presented later on and we hope it will eventually find future use as a non-trivial AspectJ benchmark. Indeed, previous work has commented on the lack of such benchmarks [17]. The `djprof` tool itself is available for download under an open source license from `http://www.mcs.vuw.ac.nz/˜djp/djprof`.

The benchmark suite used in our experiments consisted of 6 benchmarks from the SPECjvm98 suite [19] as well as 4 candidates which were eventually dropped from inclusion in it[†]. Table I details these. Where possible, we also compared the performance and precision of `djprof` against `hprof`

[†]Note, the `_222_mpegaudio` benchmark is also part of the SPECjvm98 suite. This could not be used due to a bug in the current implementation of the new array join point.

Benchmark	Size (KB)	Time (s)	Heap (MB)	SPECjvm98	Multi-threaded
`_227_mtrt`	56.0	4.7	25	Y	Y
`_202_jess`	387.2	6.5	17	Y	N
`_228_jack`	127.8	5.5	17	Y	N
`_213_javac`	548.3	12.3	31	Y	N
`_224_richards`	138.5	7.4	18	N	Y
`_210_si`	18.2	9.0	16	N	N
`_208_cst`	23.2	17.8	30	N	N
`_201_compress`	17.4	21.1	24	Y	N
`_209_db`	9.9	35.1	28	Y	N
`_229_tsgp`	7.7	36.5	26	N	N

Table I. The benchmark suite. Size indicates the amount of bytecode making up the benchmark, excluding harness code and standard libraries. Time and Heap give the execution time and maximum heap usage for one run of the benchmark.

(a well-known JVMTI profiler — see [8]) and `pjprof`, a pure Java time profiler described below. In doing this, our aim was twofold: firstly, to ascertain whether the current AspectJ implementation is competitive, compared with alternative approaches; secondly, to validate the results produced by our profiling aspects. We now provide further discussion of `djprof` and `pjprof`, detail the experimental procedure used and present the results themselves.

7.1. THE DJPROF TOOL

In this section, we consider issues related to the deployment of our aspects as part of a general purpose profiling tool. We believe it desirable that such a tool can be used on existing Java programs without knowledge of AspectJ. One solution is for the tool to statically weave aspects behind the scenes. In this case, they are combined with the original binaries to produce modified binaries in some temporary location. However, the current AspectJ implementation allows a more efficient mechanism, through a feature known as *load-time weaving*. In this case, all weaving is performed by a special classloader allowing it to be done lazily — thereby reducing costs. Therefore, we used this to implement `djprof` — a command-line tool which encompasses the profiling aspects considered in the previous sections.

Unfortunately, there is one significant drawback with the current load-time weaving implementation: *code in the standard libraries cannot be woven against*. This is very restrictive and constitutes our third limitation with the current AspectJ implementation. In fact, while in theory it is possible to statically weave against the standard libraries, we find this impractical due to the massive amounts of time and storage required. Furthermore, static weaving requires every class in the standard libraries be woven against, regardless of whether it is used or not. Thus, it seems clear that, if this limitation with the load-time weaver were overcome, then it would offer the best approach as only classes actually used by the program would be woven.

At this point, we must clarify how this limitation affects the results produced by our tool. The inability to weave against the standard libraries means that `djprof` cannot report results for methods within the libraries themselves. For heap and lifetime, accurate results are still obtained for all objects allocated in the target application. However, for wasted-time profiling, uses of objects allocated in the target application which occur in library methods are missed. This, in theory at least, could affect the precision of the wasted-time results (if a significant number of uses occur in library methods), although it remains unclear whether this really happens in practice or not. Finally, for time-spent profiling, accurate results are obtained for all methods in the target application (subject to the issues of multi-threading already discussed in Section 6).

Aside from issues of imprecision, the inability to weave against the standard libraries also gives `djprof` an inherent advantage over `hprof` and `pjprof`, since they must pay the cost of profiling all methods where `djprof` does not. While this does compromise our later performance comparison of `djprof` against `hprof` and `pjprof`, it does not render it completely meaningless. This is because we are still able to make general observations about the performance of `djprof` and, hence, the current AspectJ implementation (namely, that it is not outrageously slow in most cases and, most likely, will be competitive should this limitation be overcome).

The output produced by `djprof` consists of a list of methods, along with the amount of the profiled quantity (e.g. bytes allocated) used by them. The output is ordered so that methods consuming the most appear first. As such, `djprof` does not provide any additional context (i.e. stack-trace) information. In contrast, `hprof` is capable of providing context-sensitive information, where information is reported for individual stack-traces (to a given depth). This information can be more useful in practice, as it can help determine the circumstances (if there are any) under which a method performs badly. In fact, `djprof` can easily be extended to record such information, although we have opted against doing this to simplify our evaluation. Since recording this additional context information can be expensive, we restrict the amount of context recorded by `hprof` to a depth of one (which is equivalent to that recorded by `djprof`) to ensure a fair comparison. This is achieved using the `depth=1` command-line switch to `hprof`.

7.2. PJPROF - A PURE JAVA TIME PROFILER

The ability to write a time profiler without AspectJ is made possible in Java 1.5 with the new `Thread.getAllStackTraces()` and `Thread.getState()` methods. The former allows a daemon thread to iterate, at set intervals, the stack trace of all other threads to record their currently executing method. In doing this, `Thread.getState()` is used to ignore those which are blocked, as samples should not be recorded for them [20, 8]. This was developed by us in the course of this work and is the first pure Java time profiler we are aware of. A complete implementation, which we refer to as `pjprof`, can be found in Appendix A.

7.3. EXPERIMENTAL PROCEDURE

The SPECjvm98 benchmark harness provides an autorun feature allowing each benchmark to be run repeatedly for N iterations in the same JVM process. Generally speaking, the first run has higher overhead than the others as it takes time before JIT optimisations are applied and it also includes the weaving time. Therefore, we report the average of five runs from a six iteration autorun sequence (we discard the first run), using a problem size of 100. We believe this reflects the overheads that can be expected in practice, since most real world programs are longer running than our benchmarks (hence, these startup costs will be amortised) and, for short running programs, such overheads will be insignificant anyway. In all cases, the variation coefficient (i.e. standard deviation over mean) for the five measured runs was ≤ 0.15 — indicating low variance between runs.

To generate time-spent profiling data using hprof, we used the following command line:

```
java -Xrunhprof:cpu=samples,depth=1,interval=100 ...
```

The interval indicates hprof should record a sample every 100ms (the same value was used for djprof and pjprof). The depth value indicates no context information should be recorded (as discussed in Section 7.1), whilst the cutoff indicates the precision (as a percentage) of information which hprof should report. To generate heap profiling data, we used the following:

```
java -Xrunhprof:heap=sites,depth=1,cutoff=0.0 ...
```

The hprof tool produces a breakdown per stack trace of the live bytes allocated (i.e. those actually used), as well as the total number of bytes allocated. The results are ranked by live bytes allocated, rather than total bytes allocated. However, hprof does not report stack traces where the number of live bytes allocated, relative to the total number of live bytes allocated overall, is below a certain threshold (this is the cutoff value). Since djprof reports total bytes allocated only, a discrepancy can occur between the profilers when a method allocates a large number of bytes which are not considered live by hprof (since these will be reported by djprof, but cut off by hprof). Setting cutoff=0.0 ensures a fair comparison with djprof, since it forces hprof to report all results. Note, this does not in any way affect the performance of hprof.

The output of djprof and pjprof is similar, providing a breakdown of the total allocated by each method. A script was used to convert hprof's output into a form identical to that of djprof and pjprof. A slight complication is that, in the case of heap profiling, using a depth of 1 with hprof also does not provide comparable information with djprof. This is because hprof charges storage allocated for a type X to its constructor (indeed, its supermost constructor), rather than the method calling new X(..) (as djprof does). Therefore, to ensure the fairest comparison possible, we ran hprof twice for each benchmark when generating the heap profiling data: the first had depth=1 and was used to generate the performance data (since `djprof` does not record context information); the second had `depth=5` and was used to compare the outputs of `hprof` and `djprof` (since the additional context allowed the true calling method to be determined).

Finally, to determine the maximum amount of heap storage used by the VM (used to measure a profiler's space overhead), we used a simple program to periodically parse `/proc/PID/stat` and record the highest value for the Resident Set Size (RSS). The experiments were performed on a 900Mhz Athlon based machine with 1GB of main memory, running Mandrake Linux 10.2, Sun's Java 1.5.0 (J2SE 5.0) Runtime Environment and AspectJ version 1.5.2.

7.4. DISCUSSION OF RESULTS

Before looking at the results, we must detail our metrics. Time overhead was computed as $\frac{T_P - T_U}{T_U}$ for each benchmark, where T_P and T_U are the execution times of the profiled and unprofiled versions respectively. Space overhead was computed in a similar way.

7.4.1. HEAP PROFILING

Figure 3 looks at the overheads (in time and space) of the heap profiling implementation developed in Section 3, as well as those of `hprof`. Regarding `djprof`, perhaps the most important observations are: firstly, time overhead is quite low — especially on the longer-running benchmarks; secondly, space overhead is comparatively higher. We suspect the latter stems from our implementation of `sizeof`, which indefinitely caches the size of previously seen types. For `hprof`, we see significantly higher time overheads, while the space overheads are (roughly) of the same magnitude as `djprof`. The exact reasons behind `hprof`'s poor runtime performance remain unclear. A very likely explanation is that the additional costs of instrumenting standard libraries (which are not profiled by `djprof`) is to blame.

Figure 4 details our attempts to validate the output of the heap profiling aspect against `hprof`. To do this, we compare the profilers against each other using a metric called *overlap percentage* [18]. This

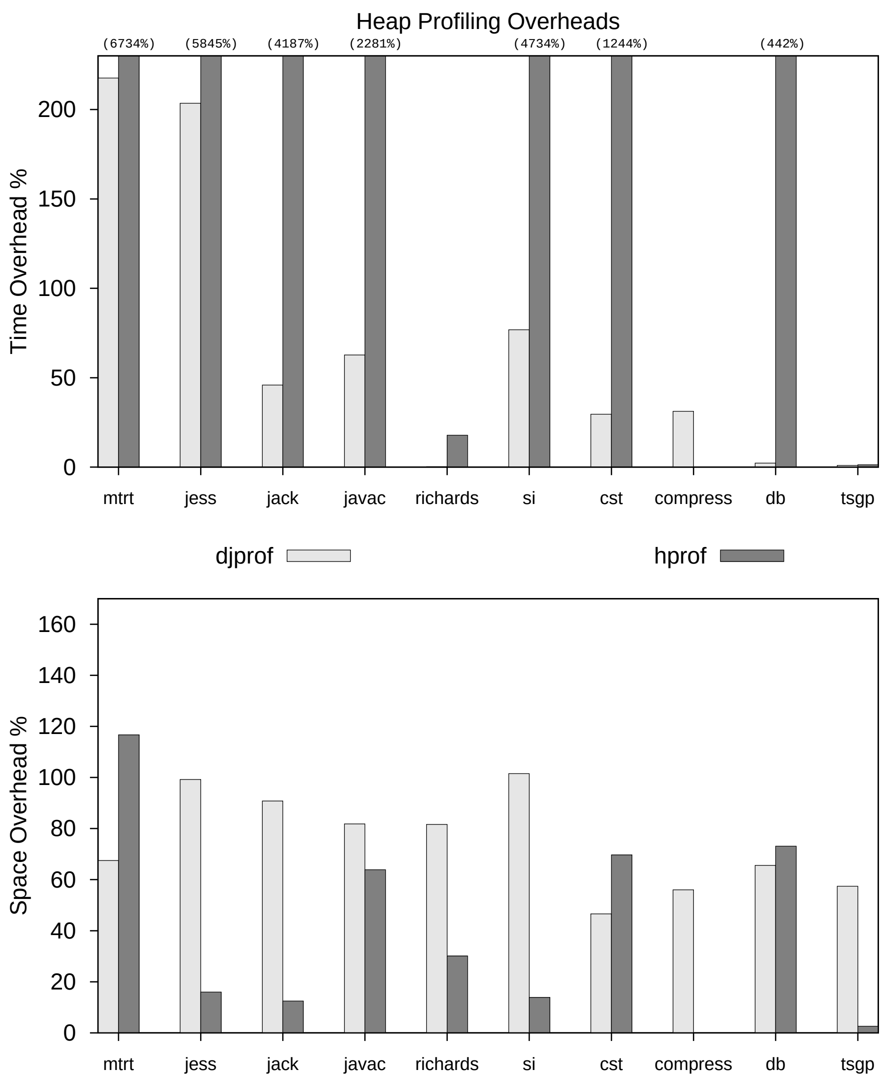

Figure 3. Experimental results comparing the overhead (in time and space) of our heap profiling implementation against `hprof` using the `heap=sites` switch. Note, empty columns (e.g. for `compress`) do not indicate missing data — only that the relevant value was very small.

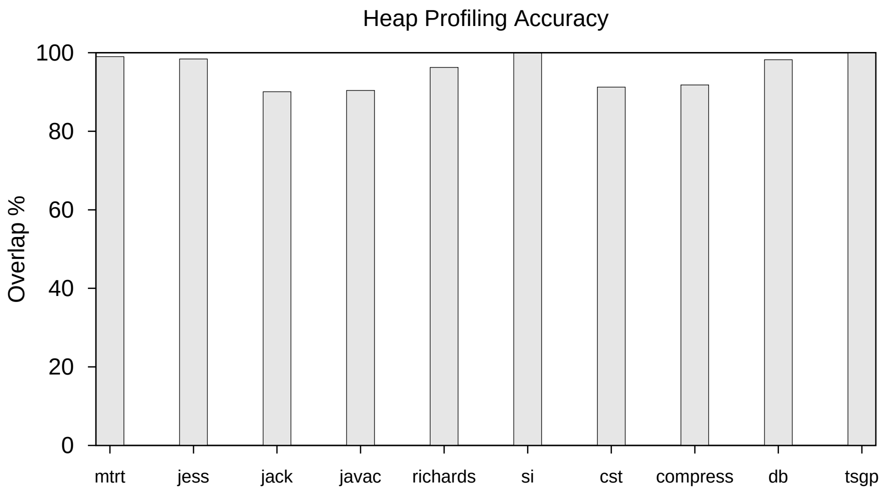

Figure 4. Experimental results comparing the precision of our heap profiling implementation against `hprof` using the `heap=sites` switch. The overlap metric indicates the amount of similarity between the output of the two profilers (see Section 7.4.1 for more discussion on this). A higher value indicates greater similarity, with the maximum being 100% overlap.

works as follows: first, the output of each profiler is normalised to report the amount allocated by each method as a percentage of the total allocated *by any method in the target application, not including the standard libraries*; second, each method is considered in turn and the minimum score given for it by either profiler is added to the overlap percentage. For example, if `djprof` reports that `Foo.bar()` accounts for 25% of the total storage allocated, whilst `hprof` gives it a score of only 15%, then the lower value (i.e. 15%) is counted toward the overlap percentage. Thus, two profilers with identical results produce an overlap of 100%, whilst completely different results have no overlap. We can think of the overlap percentage as the *intersection* of the scores given by the two profilers. Methods in the standard libraries are not included in the calculation because `djprof` cannot profile them (see Section 7.1 for more on why). In general, we find this is a useful way to evaluate profiler precision.

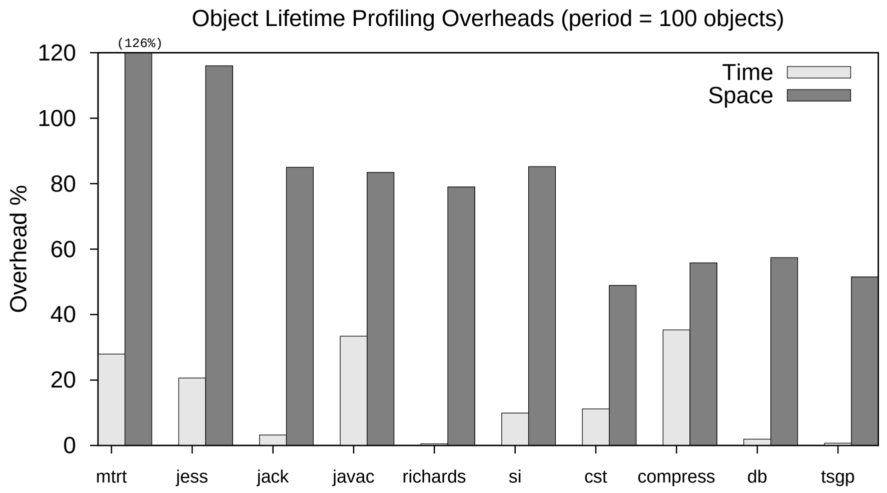

Figure 5. Experimental results looking at the overheads of our object lifetime implementations. The period indicates that every 100^{th} object was monitored. Again, empty columns (e.g. for `jack`) do not indicate missing data — only that the relevant value was very small.

Looking at Figure 4 we see that on all benchmarks `hprof` and `djprof` have an overlap of over 90%, indicating an excellent correlation between them. We would not expect identical results since `djprof` estimates object size (recall Section 3.1), where `hprof` does not.

Our overall conclusions from these results are mixed. Clearly, the inability to profile the standard libraries makes it difficult to properly compare the performance of `djprof` and `hprof`. In spite of this, the results are still interesting since they indicate that: firstly, the performance of `djprof` is not outrageously bad, compared with `hprof`, and, hence, could well be competitive should this limitation be overcome; secondly, that the precision obtained by `djprof` (when ignoring methods in the standard libraries) is good.

7.4.2. OBJECT LIFETIME

Figure 5 looks at the overheads of the lifetime profiling technique developed in Section 4. The main observations are, firstly, that similar but consistently lower time overheads are seen compared with heap profiling. Secondly, that the space overheads are also similar, but consistently higher. The first point can most likely be put down to the cost of using `sizeof` (as the lifetime aspect does not use this) which is non-trivial, especially for previously unseen types. The second point almost certainly arises because we are associating additional state with individual object instances.

7.4.3. WASTED TIME

Figure 6 details the overheads of using the two wasted-time implementations of Section 5. The main observation is that the Member Introduction (MI) approach generally performs better than just using a Map (`java.util.HashTable` in this case). Indeed, although its overhead is still large, we feel the MI approach works surprisingly well considering it is advising every public field and method access. As expected from its implementation (where an extra field is added to every used-defined object), the storage needed for the MI approach is consistently greater than for the Map approach.

7.4.4. TIME-SPENT PROFILING

Figure 7 compares the overheads of our time-spent profiling implementation against `hprof` and `pjprof`. The results show that the overheads of `djprof` are much higher than for either of the other two profilers. However, there are several other issues to consider: firstly, `pjprof` only works in Java 1.5; secondly, in other experiments not detailed here, we have found the performance of `hprof` on Java 1.4 environments to be significantly worse than `djprof`. The reason for this latter point is almost certainly due to the fact that, under Java 1.4, `hprof` uses the JVMPI whilst, under Java 1.5, it uses the more efficient JVMTI (see Section 8.3 for more on this). While these points are only relevant

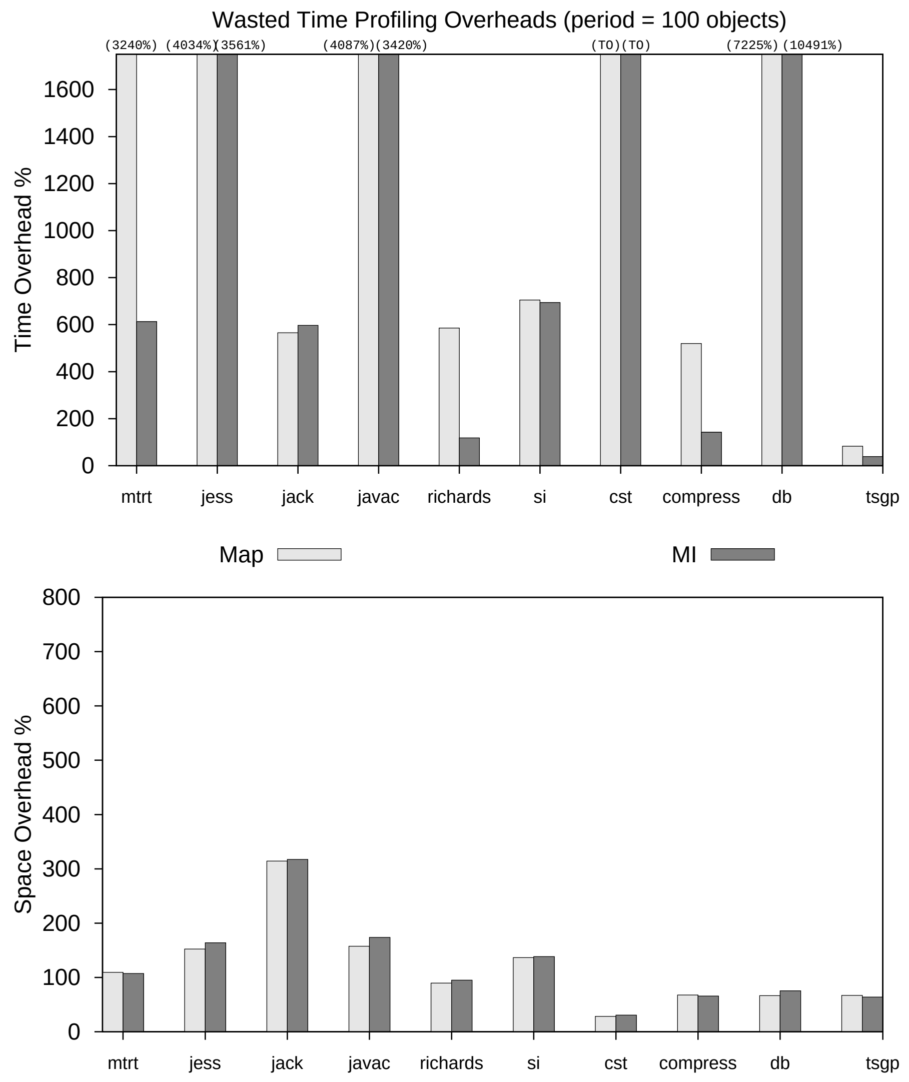

Figure 6. Experimental results looking at the overheads of our wasted-time implementations. Here, *Map* corresponds to approach 1 from Section 5, whilst *MI* (short for Member Introduction) corresponds to approach 2. TO indicates the benchmark had not completed after 1 hour (i.e. timeout) and the period indicates that every 100^{th} object was monitored.

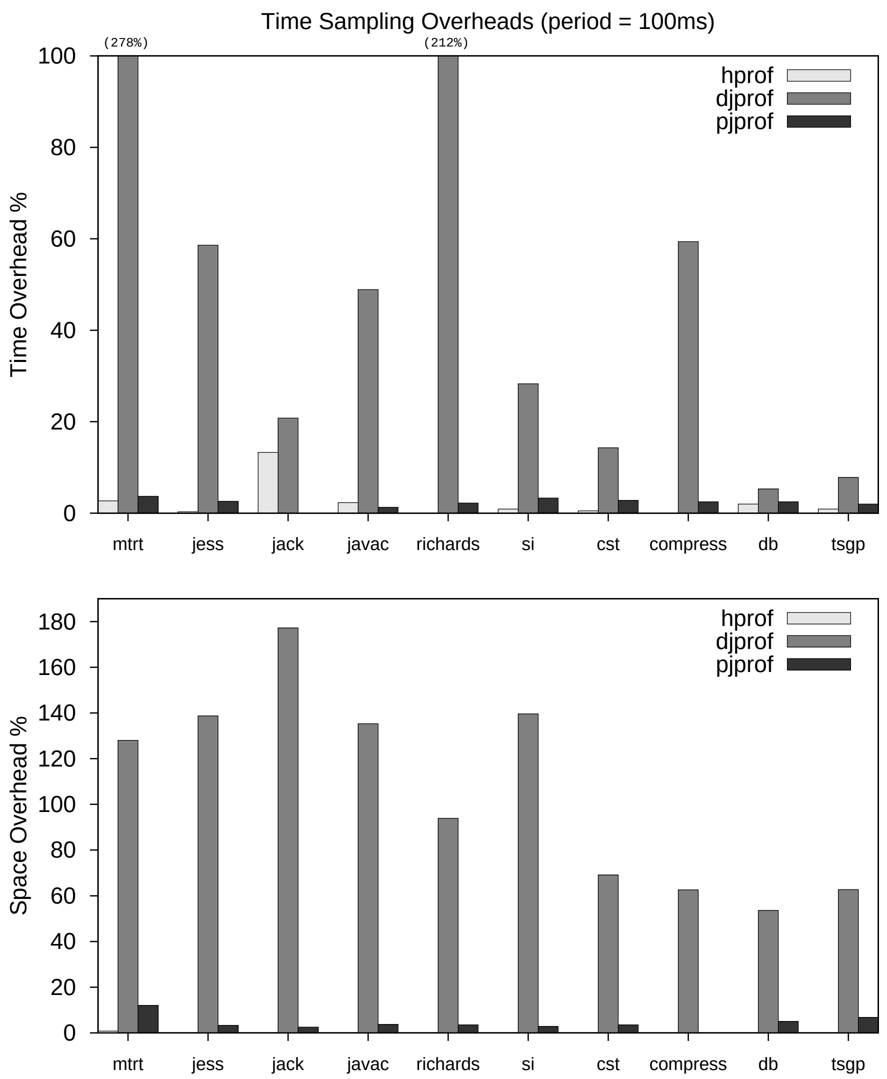

Figure 7. Experimental results looking at the overheads in time (top) and space (bottom) of our time profiling implementation, compared with `hprof` and `pjprof`. Again, empty columns (e.g. for `cst`) do not indicate missing data — only that the relevant value was very small.

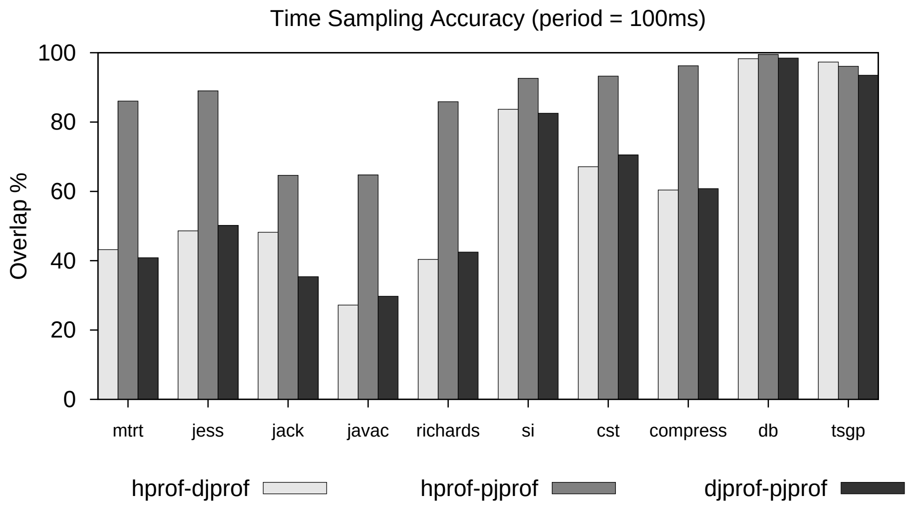

Figure 8. Experimental results looking at the precision of our time profiling implementation, compared with `hprof` and `pjprof`. The overlap metric indicates the amount of correlation between the output of the two profilers (see Section 7.4.1 for more discussion on this). A higher value indicates a better correlation, with the maximum being 100% overlap.

to those using the older Java 1.4 VM's, we expect this user-base to remain significant for some time to come.

Figure 8 details our attempts to validate the output of the time profiling aspect against `hprof` and `pjprof`. Again, overlap percentage is used to make the comparison, with each profiler normalised to report the time spent by each method as a percentage of the total spent by any in the target application, not including the standard libraries. As there are three time profilers, we compared each against the others separately in an effort to identify their relative accuracy. Looking at Figure 8, we see that `hprof` and `pjprof` have consistently higher overlaps when compared with each other. This suggests `djprof` is the least precise of the three. Since time spent in the standard libraries is not included in the overlap scores, `djprof`'s inability to profile them does not explain this observation. While the other inaccuracies mentioned in Section 6 may be a factor, we believe the main problem is simply that `djprof` causes the most perturbation on the target program. To see why, recall that our time profiling implementation adds advice before and after every method execution and method call. Even if this advice is inlined, its effect on short, frequently executed methods will still be quite high and this would skew the results significantly.

Our overall conclusion from these results is that `djprof` and, hence, the current AspectJ implementation is not well-suited to this kind of profiling. In fact, should the restriction on profiling the standard libraries be overcome, we would only expect the performance of `djprof` to deteriorate further.

8. RELATED WORK

We now consider two categories of related work: AspectJ/AOP and profiling. We also examine the JVMPI/JVMTI in more detail.

8.1. ASPECTJ AND AOP

Aspect-Oriented Programming was first introduced by Kiczales *et al.* [10] and, since then, it has received considerable attention. Many language implementations have arisen, although their support for AOP varies widely. Some, such as AspectC [21], AspectC++ [22] and AspectC# [23], are similar to AspectJ but target different languages. Others, like Hyper/J [24] and Jiazzi [25], are quite different as they do not integrate AOP functionality into the source language. Instead, they provide a separate configuration language for dictating how pieces of code in the source language compose together. AspectWerkz [26] and PROSE [27] focus on run-time weaving, where aspects can be deployed (or removed) whilst the target program is executing. The advantage is that, when the aspect is not applied, no overheads are imposed. While static weaving techniques can enable/disable their effect at runtime, there is almost always still some overhead involved. In fact, the ideas of run-time weaving share much in common with Dynamic Instrumentation (e.g. [5, 28, 29]) and Metaobject Protocols [30].

Several works have focused on AspectJ itself. In particular, Dufour *et al.* investigated the performance of AspectJ over a range of benchmarks [17]. Each benchmark consisted of an aspect applied to some program and was compared against a hand-woven equivalent. They concluded that some uses of AspectJ, specifically `pertarget` and `cflow`, suffered high overheads. In addition, `after() returning()` was found to outperform `around()` advice when implementing equivalent things. A detailed discussion on the implementation of these features can be found in [31], while [32] focuses on efficient implementations of `around()` advice.

Hanenberg *et al.* [33] consider *parametric introductions*, which give member introductions access to the target type. Without this, they argue, several common examples of crosscutting code, namely the singleton, visitor and decorator patterns, cannot be properly modularised into aspects. In fact, introductions share much in common with mixins [34] and open classes [35], as these also allow new functionality to be added at will. Another extension to AspectJ is investigated by Sakurai *et al.*, who propose a variant on `pertarget` which allows aspect instances to be associated with groups of objects, instead of all objects [36].

8.2. PROFILING

Profiling is a well known topic which has been studied extensively in the past. Generally speaking we can divide the literature up into those which use sampling (e.g. [14, 1, 37, 18, 3]) and those which use exact measurements (e.g. [28, 38]). However, exact measurements are well known to impose significant performance penalties. For this reason, hybrid approaches have been explored where exact measurements are performed on a few methods at a time, rather than all at once [28, 38]. However, it remains unclear what advantages (in terms of accuracy) are obtained. Most previous work has focused on accounting for time spent in a program (e.g. [3, 37, 1, 39, 20, 28, 38]). As mentioned already, `gprof` is perhaps the best known example [3]. It uses a combination of CPU sampling and instrumentation to approximate a call-path profile. That is, it reports time spent by each method along with a distribution of that incurred by its callees. To generate this, `gprof` assumes the time taken by a method is constant regardless of calling context and this leads to imprecision [39].

DCPI uses hardware performance counters to profile without modifying the target program [1]. These record events such as cache misses, cycles executed and more and generate hardware interrupts on overflow. Thus, they provide a simple mechanism for sampling events other than time and are accurate to the instruction level. More recent work has focused on guiding Just-In-Time optimisation of frequently executed and time consuming methods [37, 40].

Techniques for profiling heap usage, such as those developed in Sections 3, 4 and 5, have received relatively little attention in the past. Röjemo and Runciman first introduced the notions of *lag*, *drag* and *use* [16]. They focused on improving memory consumption in Haskell programs and relied upon compiler support to enable profiling. Building on this, Shaham *et al.* looked at reducing object drag in Java programs [41]. Other works use lifetime information for pretenuring (e.g. [14, 15]). Of these, perhaps the most relevant is that of Agesen and Garthwaite who use phantom references (as we do) to measure object lifetime. The main difference from our approach is the use of a modified JVM to enable profiling.

The heap profiler `mprof` requires the target application be linked against a modified system library [42]. Unfortunately, this cannot be applied to Java since the JVM controls memory allocation. Another heap profiler, `mtrace++`, uses source-to-source translation of C++ to enable profiling [43]. However, translating complex languages like C++ is not easy and we feel that building on tools such as AspectJ offers considerable benefit.

We are aware of only two other works which cross the boundary between AOP and profiling. The first of these does not use AOP to enable profiling, but instead is capable of profiling AOP programs [44]. The second uses AspectJ to enable profiling, but focuses on the visualisation of profiling data rather than the intricacies of using AspectJ in this context [45]. Finally, there are many other types of profiling which could be explored in conjunction with AspectJ in the future. These include *lock* *contention profiling* (e.g. [8, 44]), *object equality profiling* (e.g. [46]), *object connectivity profiling* (e.g. [47]), *value profiling* (e.g. [48]) and *leak detection* (e.g. [13]).

Conversely, others such *path*, *vertex* or *edge profiling* (e.g. [49, 50]) are perhaps not well suited to AspectJ, since they require instrumentation at the basic block level.

8.3. COMPARISON WITH JVMPI/JVMTI

An interesting question is what is gained with AspectJ over what can already be achieved through the standard profiling interface found in JVM's. Prior to Java 1.5, this was the *Java Virtual Machine Profiler Interface (JVMPI)*. With Java 1.5 this has been replaced by the *Java Virtual Machine Tool Interface (JVMTI)* [9]. The latter is a refined version of the JVMPI, designed to be more flexible and more efficient than its predecessor.

The JVMTI comprises two main features: an event call-back mechanism and a *Byte Code Insertion (BCI)* interface. The former provides a number of well-defined events which can be set to automatically call the JVMTI client when triggered. The latter allows classes to be modified at the bytecode level during execution. However, the BCI does not itself provide support for manipulating bytecodes and, instead, the user must do this manually (perhaps via some third-party library such as BCEL [51]). Example events supported by JVMTI include: `MonitorWait`, triggered when a thread begins waiting on a lock; `MethodEntry`, triggered on entry to a Java/JNI method; and `FieldAccessed`, triggered when a predetermined field is accessed. In general, events supported by the JVMTI tend to be those which cannot otherwise be implemented via the BCI interface. In particular, there is no event for catching objects allocated by Java programs. Furthermore, while the `MethodEntry` and `MethodExit` events exist, they are not recommended because they can severely impair JVM performance. Instead, using the BCI to catch method entry/exit should be preferred since this gives *full-speed* events. That is, since the triggers inserted to catch method entry/exit are simply bytecodes themselves, they can be fully optimised via the JVM. Note, this applies to other events such as

`FieldAccessed` and `FieldModified`. Thus, it becomes apparent that AspectJ and the JVMTI actually complement each other, rather than providing alternate solutions to the same problem.

The older JVMPI did not provide a Byte Code Insertion interface, making it less flexible. However, it did provide some events not found in the JVMTI. In particular, there was direct support for heap profiling via the `OBJECT_ALLOC` event type, which caught all object allocations made by the JVM. Finally, both the JVMPI/JVMTI can be used to profile synchronisation issues. This is not currently possible in AspectJ, as there is no join point for `synchronized` blocks. However, this feature has been requested as an enhancement and we hope this work will help further motivate its inclusion in the language.

9. CONCLUSION

Profiling tools typically restrict the user to a set of predefined metrics, enforce a particular profiling strategy (e.g. sampling or exact counting) and require the whole program be profiled regardless of whether this is desired or not. Aspect-oriented programming languages offer an alternative to this dogma by allowing the user to specify exactly what is to be profiled, how it is to be profiled and which parts of the program should be profiled. In this work, we have investigated how well an aspect-oriented programming language (namely AspectJ) lives up to this claim. We have developed and evaluated solutions to four well-known profiling problems in an effort to answer this question. The results of our investigation are mixed. On the one hand, we found AspectJ was sufficiently flexible to support the four profiling examples and that it was reasonably efficient in most cases; on the other hand, we uncovered several limitations, some of which are quite severe, which would need to be addressed before AspectJ could be considered a serious profiling platform. To summarise, these issues are:

1. **Load-time weaving standard libraries** - the inability to perform load-time weaving against the standard libraries severely handicaps any profiler (see Section 7.1). One solution to this problem

may be possible through the Java 1.5 `instrument` package and we hope this is explored in the future.

2. **State association** - We found that associating state through `pertarget` failed as `pertarget(call(*.new(..)))` does not match any join points (see Section 5). This prevented us from using `pertarget` to provide a more natural implementation of the wasted time aspect.
3. **Synchronisation** - We would like to have explored the possibility of profiling lock contention (among other things), but this is not possible as `sychronized` blocks have no join point associated with them. As this feature has already been requested by others, we feel this adds further support for its inclusion.
4. **Array allocation join point** - The current version of AspectJ (1.5.2) does not support the array join point by default — meaning array objects are not profiled (see section 3). A fix for this has been recently included in the AspectJ implementation (as a direct result of this work) and we hope this will be activated by default in future releases.

The limitations identified here are limitations with the current AspectJ implementation, rather than Aspect-Oriented Programming (AOP). Nevertheless, we believe that AspectJ — and AOP in general — has much to offer the profiling community. For example, two of the problems studied (namely, object lifetime and wasted time) are not generally supported by profilers in the main (such as `hprof`), and yet were easily expressed in AspectJ. Furthermore, building upon our techniques to develop more powerful profilers should be straightforward and opens up many possibilities that would otherwise be hard to achieve.

We do not expect AspectJ will excel at all types of profiling, since it operates on a fairly abstracted program model which, most notably, ignores many details of a method's implementation. Profiling for branch prediction is, therefore, impossible (since there is no branch join point). Likewise, profiling the flow of values (e.g. reference values) through a program is hampered by the inability to monitor value flow through local variables (this would require, at the very least, an assignment join point). Of course, both of these would be possible with a language supporting a richer variety of join points. Thus, these issues are not with AOP in general, rather they are artifacts of a particular language (i.e. AspectJ) and we could easily imagine an AOP language tailored more specifically to profiling.

In the future, we would also like to investigate how well AspectJ applies to other profiling problems, such as those discussed in Section 8.2. We would also like to investigate the amount of perturbation caused by `djprof`, although this is well-known to be a difficult undertaking [52].

ACKNOWLEDGEMENTS

We thank everyone on the AspectJ team at IBM Hursley for their help, as well as all members of the `aspectj-users` mailing list. We also thank Prof. James Noble and the anonymous SP&E (and other) referees for helpful comments on earlier drafts of this paper. Finally, Dr. Paul H. J. Kelly is supported by an Eclipse Innovation Grant.

REFERENCES

1. J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandervoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the Symposium on Operating Systems Principles*, pages 1–14. ACM Press, 1997.
2. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *ACM Conference on Programming Language Design and Implementation*, pages 85–96. ACM Press, 1997.
3. S. L. Graham, P.B. Kessler, and M.K. McKusick. gprof: a call graph execution profiler. In *ACM Symposium on Compiler Construction*, pages 120–126. ACM Press, 1982.
4. A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 196–205. ACM Press, 1994.
5. D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder. Gilk: A Dynamic Instrumentation Tool for the Linux Kernel. In *Proceedings of the International TOOLS Conference*, pages 220–226. Springer-Verlag, 2002.
6. I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: program transformations for practical scalable software evolution. In *Proceedings of the IEEE International Conference on Software Engineering*, pages 625–634. IEEE Computer Society Press, 2004.
7. E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proceedings of the International Conference on Rewriting Techniques and Applications*, pages 357–362. Springer-Verlag, 2001.

8. S. Liang and D. Viswanathan. Comprehensive profiling support in the Java Virtual Machine. In *Proceedings of the USENIX Conference On Object Oriented Technologies and Systems*, pages 229–240. USENIX Association, 1999.
9. K. O'Hair. The JVMPI transition to JVMTI, `http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition/`, 2004.
10. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242. Springer-Verlag, 1997.
11. J. D. Gradecki and N. Lesiecki. *Mastering AspectJ : Aspect-Oriented Programming in Java*. Wiley, 2003.
12. R. Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.
13. N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 351–377. Springer-Verlag, 2003.
14. O. Agesen and A. Garthwaite. Efficient object sampling via weak references. In *Proceedings of the international Symposium on Memory Management*, pages 121–126. ACM Press, 2000.
15. P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 162–173. ACM Press, 1998.
16. N. Röjemo and C. Runciman. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. In *Proceedings of the ACM International Conference on Functional Programming*, pages 34–41. ACM Press, 1996.
17. B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 150–169. ACM Press, 2004.
18. M. Arnold and B.G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 168–179. ACM Press, 2001.
19. The Standard Performance Corporation. SPEC JVM98 benchmarks, `http://www.spec.org/osg/jvm98`, 1998.
20. T. E. Anderson and E. D. Lazowska. Quartz: a tool for tuning parallel program performance. In *Proceedings of the ACM Conference on Measurement and modeling of computer systems*, pages 115–125. ACM Press, 1990.
21. Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of Path-Specific customization in operating system code. In *Proceedings of the Joint European Software Engeneering Conference and ACM Symposium on the Foundation of Software Engeneering*, pages 88–98. ACM Press, 2001.
22. O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings of the Conference on Technology of Object Oriented Languages and Systems*, pages 53–60. Australian Computer Society, Inc., 2002.
23. H. Kim. AspectC#: An AOSD implementation for C#. Master's thesis, Department of Computer Science, Trinity College, Dublin, 2002.
24. H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the International Conference on Software Engineering*, pages 734–737. ACM Press, 2000.

25. S. Mcdirmid and W. C. Hsieh. Aspect-oriented programming with Jiazzi. In *Proceedings of the ACM Conference on Aspect Oriented Software Development*. ACM Press, 2003.
26. J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proceedings of the Conference on Aspect-oriented software development*, pages 5–6. ACM Press, 2004.
27. A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the Conference on Aspect-Oriented Software Development*, pages 141–147. ACM Press, 2002.
28. H. W. Cain, B. P. Miller, and B. J.N. Wylie. A callgraph-based search strategy for automated performance diagnosis. In *Proceedings of the European Conference on Parallel Processing (Euro-Par)*, pages 108–122. Springer-Verlag, 2001.
29. K. Yeung, P. H. J. Kelly, and S. Bennett. Dynamic instrumentation for Java using A virtual JVM. In *Performance Analysis and Grid Computing*, pages 175–187. Kluwer, 2004.
30. J. des Rivires G. Kiczales and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
31. E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the ACM Conference on Aspect-Oriented Software Development*, pages 26–35. ACM Press, 2004.
32. S. Kuzins. Efficient implementation of around-advice for the aspectbench compiler. Master's thesis, Oxford University, 2004.
33. S. Hanenberg and R. Unland. Parametric introductions. In *Proceedings of the Conference on Aspect-Oriented Software Development*, pages 80–89. ACM Press, 2003.
34. G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 303–311. ACM Press, 1990.
35. C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145. ACM Press, 2000.
36. K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *Proceedings of the ACM Conference on Aspect-Oriented Software Development*, pages 16–25. ACM Press, 2004.
37. J. Whaley. A portable sampling-based profiler for Java Virtual Machines. In *Proceedings of the ACM Java Grande Conference*, pages 78–87. ACM Press, 2000.
38. D. J. Brear, T. Weise, T. Wiffen, K. C. Yeung, S. A.M. Bennett, and P. H. J. Kelly. Search strategies for Java bottleneck location by dynamic instrumentation. *IEE Proceedings — Software*, 150(4):235–241, 2003.
39. M. Spivey. Fast, accurate call graph profiling. *Software — Practice and Experience*, 34:249–264, 2004.
40. M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 111–129. ACM Press, 2002.
41. R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 104–113. ACM Press, 2001.
42. B. Zorn and P. Hilfinger. A memory allocation profiler for C and Lisp programs. In *Proceedings of the USENIX Conference*, pages 223–237. USENIX Association, 1988.

43. W. H. Lee and J. M. Chang. An integrated dynamic memory tracing tool for C++. *Information Sciences*, 151:27–49, 2003.
44. R. Hall. CPPROFJ: aspect-capable call path profiling of multi-threaded Java applications. In *Proceedings of the IEEE Conference on Automated Software Engineering*, pages 107–116. IEEE Computer Society Press, 2002.
45. M. Hull, O. Beckmann, and P. H. J. Kelly. MEProf: Modular extensible profiling for eclipse. In *Proceedings of the Eclipse Technology eXchange (eTX) Workshop*. ACM Digital Library, 2004.
46. D. Marinov and R. O'Callahan. Object equality profiling. In *Proceedings of the Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 313–325. ACM Press, 2003.
47. M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *Proceedings of the ACM symposium on Memory management*, pages 36–49. ACM Press, 2002.
48. S. A. Watterson and S. K. Debray. Goal-directed value profiling. In *Proceedings of the Conference on Compiler Construction*, pages 319–333. Springer-Verlag, 2001.
49. T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Language Systems*, 16(4):1319–1360, 1994.
50. D. Melski and T. W. Reps. Interprocedural path profiling. In *Computational Complexity*, pages 47–62, 1999.
51. Markus Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin, 2001.
52. A. D. Malony and S. Shende. Overhead compensation in performance profiling. In *Proceedings of the European Conference on Parallel Processing (Euro-Par)*, pages 119–132. Springer-Verlag, 2004.

APPENDIX A — PJPROF IMPLEMENTATION

```java
class PureJavaTimeProfiler {
 Hashtable totals = new Hashtable();
 TimerThread timer = null;
 long startTime; int period = 100; // 100ms

 PureJavaTimeProfiler() {
  startTime = System.currentTimeMillis(); timer = new TimerThread();
  timer.setDaemon(true); timer.start();
 }

 void sample() { // do the sampling
  Map<Thread,StackTraceElement[]> m = Thread.getAllStackTraces();
  Iterator<Thread> i = m.keySet().iterator();
  while(i.hasNext()) {
   Thread t = i.next();
   if(t != timer && t.isAlive() && t.getThreadGroup().getName() != "system"
    && t.getState() == Thread.State.RUNNABLE) {
    StackTraceElement ste[] = m.get(t);
    if(ste.length > 0) {
     // discard line number
     StackTraceElement s = new StackTraceElement(ste[0].getClassName(),
          ste[0].getMethodName(), ste[0].getFileName(),-1);
     getTotal(s).value++;
 }}}}

 MutInteger getTotal(Object k)
  MutInteger s;
  s = (MutInteger) totals.get(k);
  if(s == null) { s = new MutInteger(0); totals.put(k,s); }
  return s;
 }

 class TimerThread extends Thread {
  public void run() {
   while(true) { try { Thread.sleep(period); sample(); }
    catch(InterruptedException e) {}
 }}}

 public static void main(String argv[]) {
  new PureJavaTimeProfiler();
  try {
   Class clazz = Class.forName(argv[0]);
   Method mainMethod = clazz.getDeclaredMethod("main",argv.getClass());
   // construct args array for target
   String nArgv[] = new String[argv.length-1];
   Object args[] = new Object[1];
   for(int i=1;i<argv.length;++i) { nArgv[i-1]=argv[i]; }
   args[0] = nArgv;
   mainMethod.invoke(null,args);
  } catch(Exception e) {}
 }}
```

APPENDIX B — SIZEOF ASPECT

```
aspect SizeOf pertypewithin(*) {
 static final private Hashtable cache = new Hashtable();
 private int size = -1;

 after() returning(): staticinitialization(*) && !within(SizeOf) {
  size = sizeof(thisJPSP.getSignature().getDeclaringType()) + 8;
 }
 public static int get(Object o) {
  Class c = o.getClass();
  if(SizeOf.hasAspect(c)) {
   SizeOf a = SizeOf.aspectOf(c);
   return a.size;
  } else { // for classes which AspectJ cannot weave
   Integer r = (Integer) cache.get(c);
   if(r != null) { return r.intValue(); }
   else {
    int x = sizeof(c,o) + 8;
    cache.put(c,new Integer(x));
    return x;
 }}}
 static public int sizeof(Class c, Object dims...) {
  int tot = 0, m = 1;
  if(c.isArray()) {
   for(int i=0;i!=dims.length;++i) {
    c = c.getComponentType(); // move toward type held by array
    int d = ((Integer) dims[i]).intValue();
    if(i != (dims.length-1)) { tot += m * ((d*4) + 8); }
    else { tot += m * ((d*primitiveSize(c)) + 8); }
    m = m * d;
  }} else {
   Field fs[] = c.getDeclaredFields();
   for(int i=0;i!=fs.length;++i) {
    Field f = fs[i];
    if(isInstance(f)) {
     Class ft = f.getType ();
     tot += primitiveSize(ft);
   }}
   Class s = c.getSuperclass();
   if(s != null) { tot += sizeof(s); }
  }
  return tot;
 }
 static private boolean isInstance(Field f) {
  return !Modifier.isStatic(f.getModifiers());
 }
 static private int primitiveSize(Class pt) {
  if (pt == boolean.class || pt == byte.class) return 1;
  else if (pt == short.class || pt == char.class) return 2;
  else if (pt == long.class || pt == double.class) return 8;
  else { return 4; } // object references, floats + ints
 }}
```