THE COMPILE- AND RUN-TIME PERFORMANCE CONSIDERATIONS OF IMPLEMENTING CROSS-CUTTING CONCERNS AS ASPECTS

JEFF DALTON[°], SAKET RUNGTA[§], LAKSHMI SHANKAR[§], AND MATTHEW WEBSTER[§]

[°] *Department of Computer Science, Union College, 807 Union Street, Schenectady, NY 12308 U.S.A.*
daltonj@union.edu

[§] *IBM Labs, Hursley Park, Winchester, Hampshire, SO21 2JN, United Kingdom*
{saket_rungta, shankarl, matthew_webster}@uk.ibm.com

Abstract: Aspects are frequently cited as an ideal solution to large cross-cutting concerns such as logging and first failure data capture. The flexibility of the AspectJ™ language with its use of wild-cards combined with a powerful runtime reflection API allows the creation of simple and extensible aspects. The paper outlines the performance implications of weaving aspects with existing programs and the relative run-time performance of woven programs in terms of throughput and scalability. This paper is an extension to our previous work [Dalton et al, 2003]; the focus is on the new developments in AspectJ technology and deeper empirical analysis.

Keywords: Performance, Aspect-Orientation, AspectJ, Compilation, Weaving.

1 INTRODUCTION

The motivation for our study comes from our initial work to understand how Separation of Concerns can reduce the complexity of IBM® Middleware from different perspectives [Bodkin et al, 2003]. Separation of Concerns[α] provides an abstraction that reduces the ever growing complexity of large software systems. It is essential to express the 'notion of Concerns' or aspects in a language to produce an implementation. Several such implementations exist, most of which provide extensions to popular programming languages or are frameworks defined in these programming languages[β].

Performance is an important consideration for large software systems [Berry, 2003; Jain, 1991]. It is vital to understand the performance considerations of deploying aspect oriented technologies into production environments and to ensure that any trade-off for simplification over performance does not make the adoption of such technologies impractical.

1.1 Performance Measurement

Our choices for workloads in the study are programs written in the Java™ language[χ], since Java underpins the majority of IBM middleware solutions; and also due to our experience with Java Virtual Machines (JVMs) [Lindholm and Yellin, 1999] and the AspectJ[δ] language.

The metric of interest for our compile-time performance analysis is the measure of elapsed wall clock time spent in deploying aspect oriented technologies to existing programs of varying sizes. Time is spent in compilation of the aspects and, more significantly, in weaving the resulting binary representations with pre-compiled programs. Such analysis is critical for large software systems that have several minutes or hours of compilation time and any significant increase in this time is considered unacceptable.

Our run-time performance analysis is a comparison of the performance of IBM's Portable Business Object Benchmark (pBOB[ε]) with a version woven

[α] Multi-Dimensional Separation of Concerns, IBM Research, http://www.research.ibm.com/hyperspace/

[β] Aspect Oriented Software Development, http://www.aosd.net

[χ] Java Technology, Sun Microsystems™, http://java.sun.com/

[δ] AspectJ project, http://eclipse.org/aspectj/

[ε] pBOB was developed by IBM and submitted to SPEC® as the basis for SPECjbb2000®. Although SPEC has ex-

with a simple logging aspect. pBOB is a Java benchmark that reports number of successful business transactions per second [Baylor et al, 2000]. The metrics of interest here are both throughput and scalability as reported by pBOB. Such analysis is critical for production systems where any change resulting in a performance loss is considered objectionable.

The performance tools used in this study to verify measurements are tprof[φ] and arcflow [Alexander et al, 2000], each based on one of the two fundamental profiling techniques – statistical sampling and code instrumentation, respectively.

1.2 Workload: pBOB

pBOB is a Java program emulating a three-tier system with emphasis on the middle tier. All three tiers are implemented within the same JVM. These tiers mimic a typical business application, where users in the first tier generate inputs that result in the execution of business logic in the middle tier, which calls to a database on the third tier. In pBOB, the user tier is implemented as random input selection. pBOB fully implements the middle tier business logic. The third tier is represented by in-memory binary trees rather than as a separate database.

pBOB is a sensitive and well understood benchmark. It exercises the JVM, Just-in-time compiler (JIT) [Suganuma et al, 2000], garbage collector, thread implementation [Dimpsey et al, 2000] and some features of the underlying operating system and hardware. However, the pBOB benchmark performs no disk or network I/O and therefore does not accurately represent most real world workloads.

2 GOALS AND APPROACH

Our previous work to measure the performance of aspects was based on the AspectJ 1.1 compiler. That version was the first to adopt a new architecture based on the open source Java compiler[γ] in Eclipse[η] and featuring a two stage compile and binary weave process. This contrasted with the behaviour of previous versions of the compiler where source code for both the base application and any aspects was required.

AspectJ 1.2 introduces several new performance related features. A benchmark suite has been created to facilitate the measurement of weave times for a set of standard pointcuts. A comparison of results for versions 1.2 and 1.1.1 of the compiler were published in the readme file[ι] and show considerable improvements have been made. The new `-XlazyTjp` compiler option enables the optimization of AspectJ reflection. In particular this allows the efficient use of dynamic context in a logging aspect which was not possible using AspectJ 1.1.

2.1 AOP and AspectJ

Aspect-Oriented Programming (AOP) is an evolution of Object-Oriented Programming. As Java is the leading OO language so AspectJ, a small extension to the Java language, is the leading AO language. OO languages like Java allow you to define classes to encapsulate properties and behavior in your system. AO languages take this capability a step further. Using AspectJ you can define an *aspect* which not only encapsulates the implementation of a feature but also its policy i.e. how it interacts with the rest of the system. This is achieved by declaring one or more *pointcuts* which match specific *join points* in the running system e.g. method call or execution, field access or exception handling. A pointcut is accompanied by *advice* which describes additional behavior that should occur before, after or before and after (around) the matched join points. For example to implement trace entry/exit trace using an aspect you would first declare before and after advice and then associate it with a pointcut that matched method execution for every class in the system. The body of the advice calls an existing logging infrastructure such as Jakarta Commons Logging[φ] passing context from the join point e.g. method name and arguments obtained using the powerful AspectJ runtime library.

Each aspect defined in source code becomes a Java class in the runtime system. By default a singleton aspect instance is created on first use i.e. at the

tensively modified both the design and the code during the development of SPECjbb2000, the performance characteristics of these two applications is similar but not the same. Standard Performance Evaluation Corporation, http://www.spec.org

φ IBM internal tool, Similar to tprof for AIX®, Performance Management Guide, http://publib16.boulder.ibm.com/pseries/en_US/aixbman/prftungd/prftungdtfrm.htm

γ Java Development Tools (JDT), http://www.eclipse.org/jdt/

η Eclipse, A universal tools platform, http://www.eclipse.org

ι AspectJ 1.2 readme is available at http://download.eclipse.org/technology/ajdt/aspectj-1.2.jar

φ Jakarta Commons Logging, http://jakarta.apache.org/commons/logging.html

first join point matched by a pointcut declared in the aspect, and advice is invoked as an instance method on the aspect. All of the aspects used for the work described in this paper are singleton aspects.

2.2 Aspect Specification

A logging aspect, a first failure data capture (FFDC) aspect and an aspect composition of logging and FFDC aspects are used to measure compile-time cost. The logging aspect alone is used to measure run-time overhead. This is because the code associated with an FFDC concern only impacts the performance of an error path which itself carries the large overhead of exception instantiation and dispatch. As a result, determining any performance differences between the original pBOB and the pBOB variant woven with FFDC aspect would not be possible because it does not use exceptions as part of its normal processing.

2.2.1 The Logging Aspect

The abstract aspect `Logging` shown below declares pointcuts to enable the logging of public methods while excluding `toString`. This method is often used for problem diagnosis and logging it can lead to unwanted recursion. An abstract pointcut allows sub-aspects to declare the scope and the implementation of the logging.

```
public abstract aspect Logging {

  public abstract pointcut
  loggingScope ();

  pointcut toStringMethod () :
    execution(* *.toString());

  pointcut publicMethods () :
    loggingScope()
    && execution(public * *(..))
    && !toStringMethod();
}
```

The concrete sub-aspect `JakartaCommons1` declares the logging scope based on package, class or methods. Each application or component within an application that uses the logging aspect defines its own sub-aspect with an appropriately declared scoping pointcut. In this example, we have selected to include classes in the `com.ibm` package and all sub-packages. Advice is declared to pass context to a separate logging infrastructure in this case Jakarta Commons Logging v1.0.3 (JCL). JCL is a lightweight framework that provides an interface for logging method entry and exit and several logger implementations for outputting data to the console or a file.

```
public aspect JakartaCommons1 extends
Logging {
  public pointcut loggingScope () :
    within(com.ibm..*);

  before () : publicMethods() {
    if (log.isDebugEnabled()){
      log.debug("Entering: "
      + thisJoinPointStaticPart);
    }
  }

  after() returning : publicMethods(){
    if (log.isDebugEnabled()){
      log.debug( "Exiting: "
      + thisJoinPointStaticPart);
    }
  }

  private static final Log log =
    LogFactory.getLog("pBOB");
}
```

Calls to JCL are only made if logging is enabled which is determined by call to `log.isDebugEnabled()`. This guard method returns true if logging is enabled and false if it is disabled. All measurements for run-time performance are made with logging disabled. This aspect was used in our earlier work but by using `thisJoinPointStaticPart` it has the disadvantage of extracting only static context i.e. defining class and method name. Dynamic context is available through `thisJoinPoint`; however, when using AspectJ 1.1, instances are created aggressively and passed to the aspect regardless of whether logging is enabled or disabled. A short test run of pBOB involves millions of method calls and hence millions of objects were needlessly created and discarded. The resulting GC overhead caused a 100 fold reduction in the throughput of the benchmark.

In AspectJ 1.2 the `-XlazyTjp` option, when used in conjunction with an `if()` pointcut, allows `thisJoinPoint` instances to be created only when logging is enabled. A modified version of the logging aspect `JakartaCommons2`, which logs arguments to the method as well as class and method names, is shown below.

```
public aspect JakartaCommons2 extends
Logging {

  public pointcut loggingScope () :
    within(com.ibm..*);

  pointcut shouldLog () :
```

[κ] Apache AXIS, Web Services, http://ws.apache.org/axis/

```
    if(loggingEnabled);

  before () : publicMethods()
  && shouldLog() {
    if (log.isDebugEnabled()){
      log.debug("Entering: "
      + thisJoinPoint
      + " " + thisJoinPoint.getArgs());
    }
  }

  after() returning : publicMethods()
  && shouldLog() {
    if (log.isDebugEnabled()){
      log.debug( "Exiting: "
      + thisJoinPointStaticPart);
    }
  }

  private static final Log log;
  public static boolean loggingEnabled;

  static {
    log = LogFactory.getLog("pBOB");
    loggingEnabled =
      log.isDebugEnabled();
  }
}
```

The difference between `JakartaCommons1` and `JakartaCommons2` is that a static flag `loggingEnabled` is set during aspect initialization depending on whether logging is enabled. The use of an `if()` pointcut results in a test on the new flag in the body of the method being logged instead of the advice. This guard "promotion" means that a call is only made to the advice when logging is enabled.

2.2.2 The FFDC Aspect

The FFDC aspect simply instruments exception handlers (catch blocks) to record exceptions when they are caught.

```
public abstract aspect FFDC {

  public abstract pointcut ffdcScope();

  final pointcut staticContext () :
    !this(Object);
  final pointcut nonStaticContext
  (Object o) : this(o);
  final pointcut caughtThrowable
  (Throwable t) :
    handler(Throwable+) && args(t);

  before (Throwable t) :
  caughtThrowable(t)
  && ffdcScope() && staticContext() {
    processStaticFFDC(t,
    thisJoinPointStaticPart );
  }

  before (Throwable t, Object o) :
  caughtThrowable(t) && ffdcScope()
  && nonStaticContext(o) {
    processNonStaticFFDC(t,o,
    thisJoinPointStaticPart );
  }

  private void processStaticFFDC(
  Throwable t,
  JoinPoint.StaticPart tjp ) {

    log.error(
      makeProbeId(
        tjp.getSourceLocation()),t);
    }

  private void processNonStaticFFDC(
  Throwable t,
  Object o,
  JoinPoint.StaticPart tjp ) {
    log.error(
      makeProbeId(
      tjp.getSourceLocation()),t);
  }

  private String makeProbeId(
  SourceLocation sl) {
    …
  }

  private static final Log log;

  static {
    log = LogFactory.getLog("pBOB");
  }

}
```

As with the logging aspect a concrete sub-aspect is used to define the FFDC scope.

```
public aspect pBOB_FFDC extends FFDC {

  public pointcut ffdcScope() :
    within(com.ibm..*);
}
```

2.3 Aspect Characterization

The time spent in weaving aspects with a base application may be affected by a number of factors - application size, aspect scope, pointcut complexity and aspect-class interaction (advice, aspect instantiation, inter-type declarations). It is important to reduce the number of dimensions in the problem space - the approach in this study is to relate each dimension to one or more example concerns.

Tables 1 to 4 characterize the logging aspect, the FFDC aspect and a composition of logging and FFDC aspects when woven with base applications of varying size and functionality. The four base applications are the pBOB benchmark itself, Apache AXIS[κ] SOAP engine v1.1, the set of core class libraries in IBM SDK[λ] and the set of all class libraries in IBM SDK.

The tables show the total number of classes[μ] and methods[ν] in each application respectively and the scope of the aspects in terms of number of affected classes and number of affected methods. The number of exception handlers instrumented by the FFDC aspect is also shown. This choice of applications and aspects allows us to explore how the number of classes, aspect scope and aspect-class interaction affect time spent in weaving. The data supports the intuition that logging concern is more pervasive than FFDC.

Table 1. pBOB – Scope of aspects

	Classes	Methods	Handlers
Total	110	990	
Logging	92 (84%)	778 (79%)	
FFDC	45 (41%)	104 (11%)	154
Logging & FFDC	95 (86%)	821 (83%)	154

Table 2. AXIS – Scope of aspects

	Classes	Methods	Handlers
Total	753	4549	
Logging	487 (65%)	2941 (65%)	
FFDC	306 (41%)	663 (15%)	1272
Logging & FFDC	533 (71%)	3297 (72%)	1272

Table 3. Core class libraries – Scope of aspects

	Classes	Methods	Handlers
Total	2700	17074	
Logging	1947 (72%)	9767 (57%)	
FFDC	563 (21%)	1283 (8%)	1920
Logging & FFDC	2015 (75%)	10424 (61%)	1920

[λ] IBM® Developer Kit for Windows®, Java™ 2 Technology Edition, Version 1.4.2, 32-bit version. Available as a part of IBM Development Package for Eclipse at http://www.ibm.com/developerworks/java/jdk/eclipse/

[μ] Number of classes is defined as the number of .class files produced by the ajc compiler 1.2 for a given source tree.

[ν] Number of methods for a class is defined as the total number of methods in a class excluding the constructor methods.

Table 4. All class libraries – Scope of aspects

	Classes	Methods	Handlers
Total	8336	61582	
Logging	6001 (72%)	33823 (55%)	
FFDC	1756 (21%)	4082 (7%)	6145
Logging & FFDC	6184 (74%)	35866 (58%)	6145

2.4 Compile-Time Performance

The goal of compile-time performance analysis is to provide empirical evidence of the impact on overall build time when weaving aspects or aspect compositions with large applications. Figure 1 illustrates our use of aspects. The source representations for a base application are complied into bytecodes in binary .class files as normal, typically using the javac compiler. The source representations for aspects are compiled into bytecodes in binary .class files, using the ajc compiler. The collection of base application .class files in a JAR and the collection of aspect .class files in a JAR file (aspect library) are woven together using the ajc[ο] compiler. The time taken for each step may be measured.

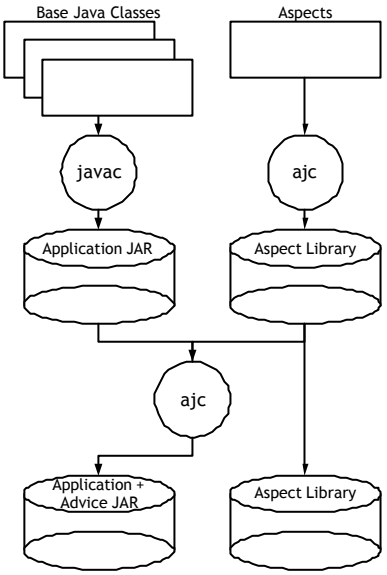

Fig. 1. Compiling Classes and Aspects

[ο] The ajc compiler uses the Byte Code Engineering Library or BCEL for byte code modification during the weaving process. The Byte Code Engineering Library, http://jakarta.apache.org/bcel/

2.5 Run-time Performance

The primary goal of run-time perfomance analysis is not to measure the overhead of the logging concern but to make a comparison between an aspect-oriented implementation and its hand-coded equivalent. Our choice of application, pBOB, has no existing logging support. This removes the possibility of interference between an existing logging implementation and the logging aspect.

To ensure the functional equivalence of the two logging aspects and the hand-coded equivalent, a 30 second run of pBOB was made for each with logging enabled. Three separate analyses were made:

1. The logs produced by each version of pBOB were compared to ensure exactly the same methods were covered.
2. Method entries and exits were counted to ensure the numbers were exactly the same. This is an important consideration for the hand-coded approach because methods may contain more than one return. This problem is handled automatically by AspectJ.
3. Each pBOB run measures a random workload so the total number of log records will be different each time. To test functional equivalence 3 separate runs were made for each version of pBOB and an average number of records determined. The variation was found to be < 0.01%.

The listing below is an example of the hand-coded logging implementation for a single method.

```
public static Address createAddress(Base
nearobj,
  AccessMode amode) {
    if (log.isDebugEnabled()){
      log.debug(
        "Entering: Address Address”
        + ”.createAddress(Base,
          AccessMode)");
    }

  // Method body

  if (log.isDebugEnabled()){
    log.debug(
      "Exiting: Address Address”
      + ”.createAddress(Base,
        AccessMode)");
  }

  return newAddress;
}
```

For comparison we decompiled the code generated by the AspectJ compiler for the first logging aspect `JakartaCommons1`:

```
public static Address createAddress(
Base nearobj,
AccessMode arg1) {

  JakartaCommons1
  .aspectOf()
  .ajc$before
  $logging_JakartaCommons1$1$e62c3fca(
  ajc$tjp_0);

  // Method body

  JakartaCommons1
  .aspectOf()
  .ajc$afterReturning
  $logging_JakartaCommons1$2$e62c3fca(
  ajc$tjp_0);

  return newAddress;
}
```

What is clear is the apparent overhead of separating the base application from the logging implementation. An aspect instance must first be located through the `aspectOf()` method call and the generated advice method invoked before the logging infrastructure is actually called within the aspect. What is not apparent is whether this impacts the overall performance of the application.

To see the effect of the `-XlazyTjp` option we decompiled the code generated for the second logging aspect `JakartaCommons2`:

```
public static Address createAddress(
Base nearobj,
AccessMode amode) {

  Address newAddress;
  Base base = nearobj;
  AccessMode accessmode = amode;

  if(JakartaCommons2.ajc$if_13())
    JakartaCommons2
    .aspectOf()
    .ajc$before
    $logging_JakartaCommons2
    $1$4f3a56f9(
    org.aspectj.runtime.reflect.Factory
      .makeJP(ajc$tjp_0, null, null,
              base, accessmode));

  // Method body
```

```
  if(JakartaCommons2.ajc$if_13())
    JakartaCommons2
    .aspectOf()
    .ajc$afterReturning
    $logging_JakartaCommons2
    $2$4f3a56f9(ajc$tjp_0);
  return newAddress;
}
```

The most important difference between `JakartaCommons1` and `JakartaCommons2` is the two `if()` tests which determine whether to call advice in the aspect. This implementation of the `if()` pointcut calls a static method defined by the aspect which in turn tests the `loggingEnabled` flag. This promoted guard ensures `thisJoinPoint` objects, containing references to the method arguments, are only created and passed to the advice when logging is enabled. The side effect is a much more efficient implementation because calls to both the `aspectOf` method and the advice to test the logging guard are avoided when logging is disabled.

Measurements with pBOB variants were made with both `JakartaCommons1` and `JakartaCommons2` logging aspects as well as the hand-code equivalent. A fourth measurement with the original pBOB was chosen as the reference point. All measurements were taken with logging disabled. This is the standard practice when making an assessment of the performance impact of serviceability concerns. This is because:

1. It is the most frequent mode of operation: logging is typically enabled when diagnosing problems.
2. Enabling logging can significantly impact performance because of the relative cost of the underlying logging infrastructure compared to the method being logged.

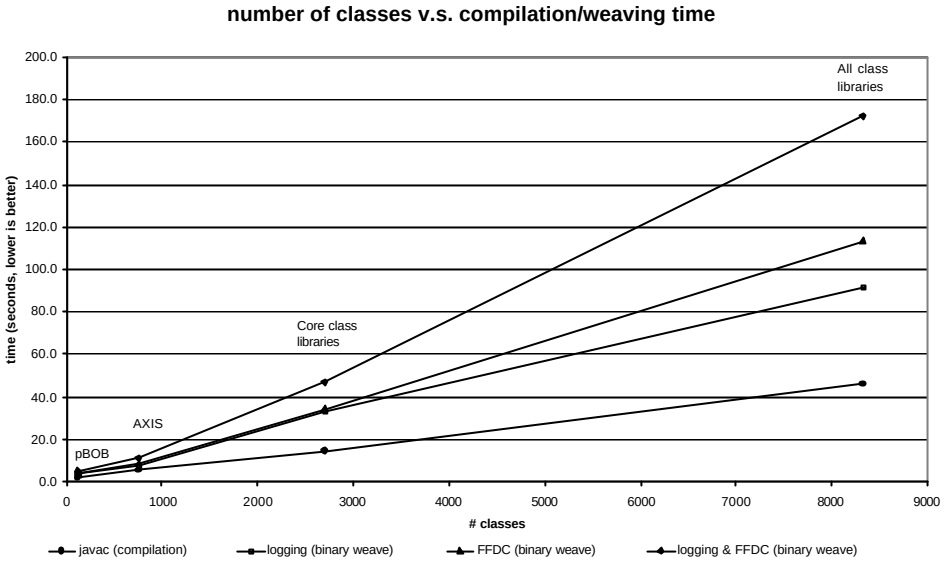

Fig. 2 Compile-Time Performance Results

3 RESULTS

Results for both compile-time and run-time tests were collected using the IBM JRE[π] running on Windows 2000 Advanced Server™ Service Pack 4 on a hyper-threaded 8-way (physical 4-way) Intel® Pentium® 4 Xeon™ 2.8 GHz machine with 512KB Level 2 cache and 4096KB Level 3 cache.

3.1 Compile-Time Performance

The compilation of base classes takes a relatively significant time whereas the compilation of aspects is relatively trivial. However, weaving takes a relatively significant time. We ignore the aspect compilation time and compare the other two timings in our study. Three aspects (logging, FFDC, logging and FFDC) are woven with four applications of increasing size (pBOB, Apache AXIS, IBM SDK core class libraries, IBM SDK class libraries).

Figure 2[ρ] illustrates the compilation or weaving times[ρ] for different application sizes in terms of number of binary classes. The compilation time for base classes as well as the weaving times for both logging and FFDC seem to be linear with respect to application size. However the weaving time for the combined aspect appears to be non-linear with respect to application size. In addition while the weaving time for the combined aspect is greater than that for the individual aspects it is not the sum of the individual times. For example the combined time for the core class libraries is 50 seconds while the sum of the individual times is 73 seconds (68%). The same comparison for all class libraries gives times of 175 and 210 seconds respectively (88%).

[π] IBM® Runtime Environment for Windows®, Java™ 2 Technology Edition, Version 1.4.2, 32-bit version

[θ] Reproducibility, repeatability and convergence of the results have been checked. All results are arithmetic mean with 95% confidence level and on average 1.26% confidence intervals (worst case 4.98% confidence interval) with a reasonable number of iterations. Results were collected with a large fixed heap of 1024MB to minimise Garbage Collection overhead.

[ρ] Measurements were made using a high-resolution (microseconds) timer. It is justified as our measurements are several orders of magnitude higher.

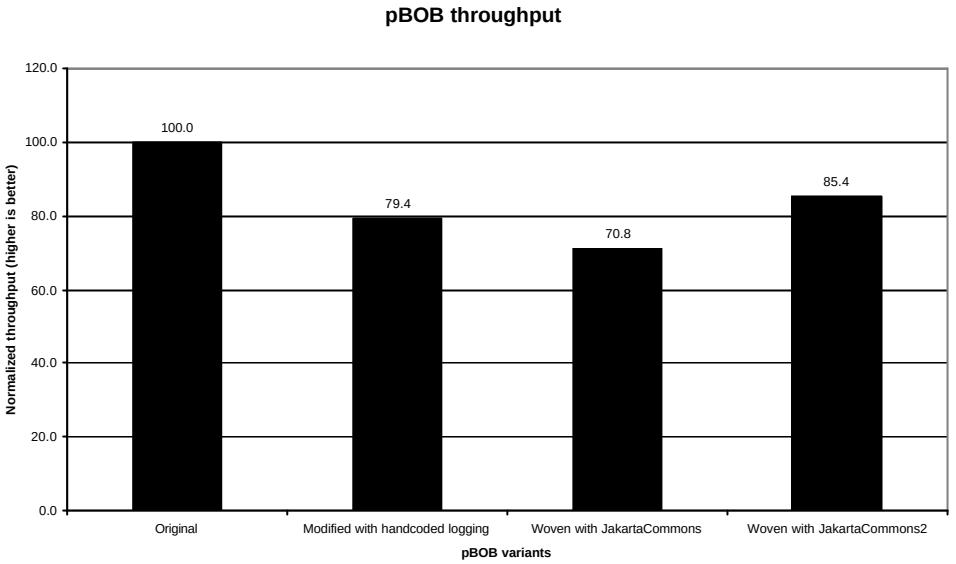

Fig. 3. Run-Time Performance Results (throughput)

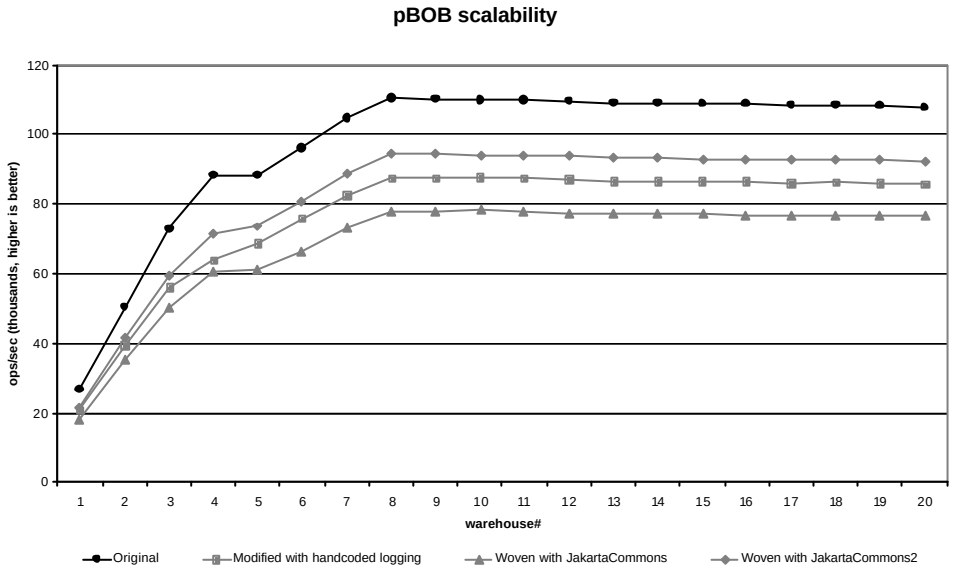

Fig. 4. Run-Time Performance Results (scalability)

3.2 Run-time Performance

The pBOB benchmark measures the throughput of the underlying Java platform, which is the rate at which business operations are performed per second. The benchmark progresses by incrementally increasing the workload unit or number of warehouses by one. The workload unit is referred to as a warehouse in pBOB context. Each warehouse workload may saturate a single processor. The overall throughput or score as reported by pBOB is a composite number, which is the arithmetic mean of throughput for warehouses n to 2n, where warehouse n has the peak performance. Theoretically, the peak performance is expected when the number of warehouses is the same as the number of processors in the system.

We compare the performance of original pBOB, a pBOB variant with hand-coded logging, and two further variants of pBOB woven with logging aspects `JakartaCommons1` and `JakartaCommons2`. For all three variants the logging infrastructure is in place, but is disabled.

The results[σ] in Figure 3 and Figure 4 show a significant difference in throughput when logging is introduced, even though it is disabled. This is due to the need to test the guard on entry and exit to logged methods in the application. The throughput for each of the logging implementations is very different, but all demonstrate good scalability

The difference in throughput between the pBOB variant with hand-coded logging and that woven with logging aspect `JakartaCommons1` suggests that there is an overhead associated with locating an aspect and calling advice. However, the use of the promoted guard greatly improves the throughput for the variant woven with the `JakartaCommons2` logging aspect.

The slope in Figure 4 is different for the line between warehouses 1 to 4 than between warehouses 5 to 8; the former is saturating the four physical processors, while the latter is exploiting the four virtual processors. The virtual processors are enabled by using the hyper-threading[•] technology in new Intel processors.

4 CONCLUSIONS

4.1 Compile-Time Performance

The results in Figure 2 suggest that the weaving time is of the same order (a factor of 1.3 up to a factor of 3.7) as the compilation time for base classes and more importantly, the aspect composition does not result in a timing that is the sum of the timings when each aspect is woven individually.

The results show that the time to weave logging and FFDC is similar for small applications but FFDC becomes relatively more expensive as application size increases. This reflects a difference in the exception handling policy of the applications chosen.

4.2 Run-time Performance

The pBOB benchmark is not an ideal application for assessing the overall performance cost of logging in a typical business application. This is clear from the throughput results which show an overhead of up to 30% when logging is disabled. The reason for using pBOB is that it is highly sensitive to changes in performance and our goal is to find the difference between an AO implementation of logging and its hand-coded equivalent rather than performance overhead of either implementation.

The results show that there is a difference in performance between an aspect-oriented logging implementation (see `JakartaCommons1`) and its hand-coded equivalent. However, this can be greatly improved by a small change to the implementation of the aspect (see `JakartaCommons2`) albeit at the cost of flexibility: the logging state cannot be modified after application initialization. While a similar change to the hand-coded implementation could also result in a performance improvement, applying and maintaining such a change would be considerably more difficult in terms of effort. Therefore, we do not pay a performance penalty for the modular improvements afforded us by the use of AOSD.

5 RELATED WORK

[Hilsdale and Hugunin, 2004] describe the compile-time performance of different pointcut designators on a single project with the aim of improving the performance of the compiler. The measurement of run-time performance compares different logging aspects to identify the most efficient implementation. In particular we have used the `if()` pointcut to facilitate the extraction of dynamic context and improve the overall performance of the logging aspect.

6 FUTURE WORK

We have concluded that the time taken to weave two pervasive aspects is less than the sum of the times taken to weave the aspects separately. This argument would be better supported by studying more complex aspect compositions (more than two aspects woven together with an application). It would also be interesting to explore whether weave time could be determined by the characteristics of the constituent aspects.

An alternative approach to using different applications for this study would be to vary the scope (as defined by pointcut declaration) for a single application and verify that the aspect characteristics remain similar.

[σ] All results are arithmetic mean with 95% confidence level and on average 0.29% confidence intervals (worst case 0.42% confidence interval) with a reasonable num-ber of iterations. Results were collected with a fixed heap of 1550MB, the maximum permitted by the JVM on this platform, to maximise the performance.

[•] Intel Hyper-Threading Technology http://www.intel.com/technology/hyperthread/

Weaving aspects requires a considerably larger Java heap compared to that needed to compile the base classes. In addition, the profiles reveal a large number of objects are being created and discarded. An analysis of the types of objects involved may lead to an improvement in weave times.

The results for weaving the logging and FFDC aspects show similar times despite considerable differences in both the number of affected classes and methods: the scope of the logging aspect is much larger. The advice defined in each aspect affects different join points: logging advises method execution while FFDC advises exception handlers. Further profiling of ajc may reveal how different types of advice affect weave times.

The aspects used in this project to test the performance of logging were extremely simple and in general do not reflect the implementation of logging in large applications. Firstly, a single log object is used for the whole system. This greatly reduces the control a user has over logging. Many applications, the Apache AXIS open source project for example, create a separate log object for each class allowing logging to be configured for a single class or group of classes. Secondly, the use of the `loggingEnabled` flag, which is set once during application initialization, prevents dynamic configuration of the logging concern i.e. we cannot enable or disable it during execution. Most large applications allow the user to do this.

Future work to measure the performance of a logging aspect should address these differences. One approach would be to use features of the AspectJ language and runtime library; however a per-type aspect instantiation model is not available in AspectJ 1.2. A future version of AspectJ may include this feature.

One consequence of the use of aspects not addressed by this work is code bloat. Initial observation suggests an increase of up to 10% for .class files on disk when using a logging aspect as compared to a hand-code equivalent implementation. While this may not be a primary consideration for middleware deployment it may be a concern for J2ME[τ] products. Another consideration is Java heap occupancy. The use of AspectJ reflection results in the allocation of large numbers of static join point objects; for example one per exception handler for an FFDC aspect. The size of each object and whether it is allocated aggressively or on first use will both effect the overall memory consumption of an application using aspects.

[τ] Java 2 Platform, Micro Edition (J2ME™), http://java.sun.com/j2me/

AspectJ 1.2 supports load-time weaving whose function is similar to the binary weaving used in the work described by this paper except that the weaving of base classes and aspects occurs as they are loaded into the JVM. This approach further decouples the application from any aspects, allowing greater scope for them to be modified independently or the use of environment-specific aspects such as those used in middleware; however the cost of weaving is transferred from compilation to application initialization and execution. The impact on application server environments, where load-time weaving will most likely used, should be assessed.

Classes that have been woven may not subsequently be rewoven: all aspects must be applied during the same compiler invocation. AspectJ 1.2 introduces the `-Xreweavable` option which allows classes to be woven more than once by saving a copy of the original bytecodes. An interesting future analysis would be to assess the footprint and weaving time characteristics of this new feature.

ACKNOWLEDGEMENTS

We are indebted to Dr. Robert F. Berry, IBM Distinguished Engineer, for his valuable advice and guidance. We would also like to thank Dr. David Siegwart for his help in interpreting the results.

REFERENCES

Alexander W.P. et al 2000, “A unifying approach to performance analysis in the Java environment”. In IBM Systems Journal, Volume 39, Number 1.

Baylor S.J. et al 2000, “Java server benchmarks”. In IBM Systems Journal, Volume 39, Number 1.

Berry R. 2003, “Trends, Challenges and Opportunities for Performance Engineering with Modern Business Software”. In UKPEW'03, July 9-10, University of Warwick, UK.

Bodkin R., Colyer A. and Hugunin J. 2003, “Applying AOP for Middleware Platform Independence”. In AOSD 2003, March 17-21, Boston, Massachusetts, USA.

Dalton J. et al 2003, “The Compile- and Run-time Performance Considerations of Implementing Cross-Cutting Concerns as Aspects”. In UKPEW'04, July 7-8, University of Bradford, UK.

Dimpsey R. et al 2000, "Java server performance: A case study of building efficient, scalable JVMs". In IBM Systems Journal, Volume 39, Number 1.

Hilsdale E. and Hugunin J. 2004, "Advice Weaving in AspectJ". In AOSD 2004, March 22-26, Lancaster, UK.

Jain R. 1991, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling", Wiley, New York, USA.

Lindholm T., Yellin F. 1999, "The Java Virtual Machine Specification", Addison-Wesley, 2nd edition, USA.

Suganuma T. et al 2000, "Overview of the IBM Java Just-in-Time Compiler". In IBM Systems Journal, Volume 39, Number 1.

TRADEMARKS

Jeff Dalton

Jeff Dalton received his B.Sc. (Hons) in Computer Science at Union College (Schenectady, NY) in 2004. He completed an internship in 2003 researching AspectJ with the Java Performance Team at the IBM UK Hursley Lab. His research interests include focused search engine technology, automated text classification, aspect-oriented programming, and software engineering practices. He also spends his time applying information science to the science of cooking.

Saket Rungta

Saket Rungta joined IBM in 2002, after receiving his Bachelors in Software Engineering from The University of Manchester. He has been part of the Java performance team since his one year internship at IBM's Software Development Laboratory in Hursley, UK, in 2000. He has worked on the performance of Serially Reusable Java Virtual Machine for z/OS and Decimal arithmetic in Java. Recently, he has been actively involved with Garbage Collection performance of IBM Java Virtual Machines. He is pursuing a part-time Masters in Software Engineering at the University of Oxford.

Lakshmi Shankar

Lakshmi Shankar is a Software Engineer at IBM Hursley Labs, UK. After completing his Masters in Computing (Software Engineering) from Imperial College, he joined IBM, where he has worked for more than two years in a range of areas including Java performance, test, and development. He is currently the Class Loading component owner for IBM's Java technology. He has co-authored several articles, including one on Java Shared Classes, recently published on IBM developerWorks.

Matthew Webster

Matthew Webster joined IBM in 1989 with a degree in Physics with Computer Science from Southampton University and since then has worked on a number of software projects at the Hursley lab. He moved to the Java Technology Centre in 1997 initially as a technology evangelist then working on the restructure of the IBM JVM and leading the development of advanced Garbage Collection and Class Loading features. Matthew is a senior software engineer developing AOP technology for use in IBM software products and is co-author of a book on AspectJ and Eclipse that is published later this year.