Virtual Machine-based Control Flow Matching

Michael Haupt Sebastian Kanthak Mira Mezini
Software Technology Group
Darmstadt University of Technology, Germany
{haupt,mezini}@informatik.tu-darmstadt.de, kanthak@rbg.informatik.tu-darmstadt.de

ABSTRACT

Aspect-oriented programming languages allow for quantifying, using pointcuts, over the execution of programs, and for attaching additional functionality to join points in the execution where the quantifications match. Frequently, join points cannot be statically resolved, i. e., mapped to locations in program code (join point shadows). In these cases, pieces of conditional logic called residues are used to determine whether join point shadows actually yield join points at run-time. Certain pointcuts allow for quantifying over the dynamic control flow of programs. The implementation of the associated residues is comparatively expensive with respect to performance. This paper presents the implementation of control flow residues in a virtual machine supporting aspect-oriented programming natively. Extensive performance measurements show that implementing such support at virtual machine level is beneficial and yields results that are at least comparable to static optimisations.

1. INTRODUCTION

In complex systems, concerns tend to "cross-cut" each other. In terms of object-oriented decomposition, this means that the implementations of most concerns are clearly mappable to classes and collaborations thereof. The implementations of other concerns, however, are *scattered* over modules of the aforementioned kind, and *tangled* with their code.

The aspect-oriented programming (AOP) paradigm [23, 12] introduces a new kind of modules called *aspects* that allow for capturing crosscutting concerns in a localised way and with explicit interfaces to the rest of the system. Aspect-oriented programming languages introduce the following notions [22]. Crosscutting behaviour, encapsulated in aspects, is regarded as functionality that is to be executed whenever the application it cuts across reaches certain points in its execution. These points in the execution graph of an application are called *join points* (e. g., method calls, field accesses, etc.). They are quantified over by means of so-called *pointcuts*, which thus are queries over the execution of a program.


```
1  class A {
2      B b = new B();
3      void m() { b.x(); }
4      void n() { b.y(); }
5      void o() { b.x(); b.y(); }
6  }
7
8  class B {
9      void x() { ... this.k(); ... }
10     void y() { ... this.k(); ... }
11     void k() { ... }
12 }
```

Listing 1: Sample classes for cflow.

Whenever a pointcut matches, crosscutting behaviour, represented in the form of *advice*, which are method-like constructs, can be executed.

To implement such a model, a process called *weaving* is often used. It inserts advice invocations into application code at locations called *join point shadows* [24]. Join point shadows are code structures (expressions, statements or blocks) that yield join points during execution. For example, the shadow of a method call is a call instruction. Join point shadows are determined by evaluating pointcuts.

Not all join point shadows can be fully determined statically, because join points are dynamic by nature. Pointcuts that quantify over dynamic properties of join points cannot definitely be mapped to code locations. For example, in AspectJ [22, 4] (an aspect-oriented extension to Java), pointcuts exist that filter method calls based on the type of the target object, or on the types of arguments to the call. When such dynamic conditions apply to a join point shadow, the weaver generates pieces of conditional logic called *residues* that are woven in at the shadow along with advice invocations.

A commonly used dynamic pointcut is **cflow**. It quantifies over control flows. To illustrate its meanings, consider, for example, the two classes in Lst. 1. Both methods in A invoke methods on an instance of B, and both of those methods in turn invoke, at some time during their execution, B.k().

Next, assume that the call to k() is to be advised by some aspect. The advice is to be executed *only* if k() is called in the course of an execution of A.m(). The pointcut expressing this looks as follows:

```
1  call(void B.k()) && cflow(execution(void A.m()))
```

In other words, "match all calls to B.k() that occur in the control flow of the execution of A.m()". Normally, it cannot be determined statically whether a particular call to k() is

inside that control flow. This is why dynamic residues are needed.

Different AOP implementations have different ways of implementing these residues (details will be given in Sec. 2). The vast majority of existing AOP systems implements them as calls to the particular system's AOP infrastructure that are woven at join point shadows. This has the effect that the residues have to be executed by the virtual machine as part of the running application, which induces performance penalties due to the overhead associated with maintaining, updating, and querying the data structures connected with the residues.

One way to reduce the amount of residues needed to be executed is by static analysis. The `abc` compiler [25, 7, 1] of the popular AspectJ language introduces static inter-procedural analysis to optimise away a large amount of residue overhead. While the performance gain earned is considerable [7], the downside of the approach is that the `abc` compiler performs a *whole-program analysis*. This has two important implications. Firstly, it increases compilation time significantly even for simple programs. Secondly, it places Java applications under a closed-world assumption that contradicts Java's dynamic class loading capabilities. This is especially hard to bear in the area of middleware containers, for which AOP has been recognised as a great tool to reduce the complexity of transparent service injection, because such containers heavily rely on dynamic deployment of business applications.

Another approach to deal with **cflow** residue performance is in the focus of this paper. The idea is to provide support for **cflow** at the level of the execution layer, i. e., in the virtual machine itself. The underlying rationale that has driven our work has been as follows. First, the VM maintains some dynamic model of the execution as it executes the code, of which we can directly make use for some kind of quantification over control flows. Second, when the needed information is not directly accessible and we need to construct it by integrating residues into the bytecodes, the latter can be implemented more efficiently within the VM.

To validate our hypothesis, we have implemented native support for **cflow** residues in Steamloom [16, 8], a Java virtual machine with dedicated support for AOP mechanisms, which is based on IBM's Jikes Research Virtual Machine [3, 2, 20]. We present and evaluate three different implementation strategies for **cflow** in the context of Steamloom and analyse their performance characteristics. This side by side discussion of three alternative implementations with the evaluations of when each of them is best applicable constitutes one of the contributions of this paper. The second contribution is to show that the residue mechanisms for **cflow** can be implemented more efficiently when exploiting direct access to VM internals. This claim is backed by performance evaluations.

The structure of this paper is as follows. In Sec. 2, we will first abstractly outline three implementation strategies for **cflow**, and then briefly present current existing AOP implementations implementing the strategies. All of these strategies have also been realised in Steamloom, which implementations are presented in Sec. 3. The performance of these approaches, and that of other systems, is evaluated in Sec. 4. This section is also where the different approaches are discussed. Sec. 5 concludes the paper.

2. CFLOW IMPLEMENTATIONS

When **cflow** is used, the idiom **cflow**(pc1) && pc2 is frequently met, denoting that the pointcut shall match join points pertaining to pc2 *only* if they occur in the control flow of some join point matched by pc1. In the following, join points matched by pc1 will be called *control flow constituents*. A control flow constituent's shadows mark *entries* and *exits* of control flows. Shadows pertaining to join points matched by pc2 will be called *dependent* shadows.

In general, an implementation of **cflow** needs to address the following two issues:

1. At control flow entries and exits, action needs to be taken to monitor the state of the control flow, i. e., whether it is active or not.

2. At dependent shadows, it must be checked whether the control flow is currently active to determine whether the advice attached to the join point shadow needs to be invoked.

It is usually possible in AOP implementations to access the context at constituent join points, and to use this context in advice attached to dependent join points. In this paper, we do not deal with **cflow** pointcuts that do this. The focus of this work solely is on *matching* control flows, not on accessing their constituent join points' contexts.

We shall now outline three different approaches in generalised form before AOP implementations employing them and their implementation in Steamloom are introduced.

2.1 Counters

When this approach is adopted, residues are attached to control flow entries and exits that update counters. When a control flow is entered, the counter is incremented, and it is decremented when the control flow is left. At dependent shadows, residues are woven that check whether the counter is greater than zero. If so, the control flow is active and the advice can be executed.

Control flow counters exist once per control flow. Furthermore, they must exist once per thread for this approach to work. Did they not, different threads entering and leaving the same control flow could easily corrupt the control flow counter state. For example, AspectJ [22, 4] uses ThreadLocal instances to maintain control flow counters.

Using counters imposes a constant overhead at control flow entries and exits as well as at dependent shadows.

2.2 Stack Walking

The stack walking approach does not need any residues at control flow entries and exits. Instead, it gets hold of the current call stack at dependent shadows and iterates over the methods on the stack to check whether the control flow in question is currently active.

This approach does not need to regard thread locality, because the call stack that a residue accesses is *always* the one of the currently executing thread. Depending on the language used, there are different approaches to access the call stack. In Java, the call stack can be accessed by creating an instance of Throwable, which can be queried for the stack frames via its getStackTrace() method. In Smalltalk, the call stack is immediately accessible due to the reflective nature of the language. Any approach based on C can operate on the machine level directly.

There is no cost at control flow entries and exits connected with stack walking. However, the cost imposed on dependent shadows is directly dependent on the depth of the call stack. In the most inauspicious case, the entire stack must be parsed only to determine that a particular control flow is *not* active at present.

On the other hand, stack walking could be beneficial when complex nested control flows are to be matched, stating, for example, that a given sequence of methods must be on the stack in a given order. It is possible to regard such a nested control flow as a regular expression that can be matched by an automaton that walks the stack.

2.3 Continuous Weaving

Using continuous weaving[1], it is possible to leave dependent join point shadows *completely* unaffected while the control flow in which they should be decorated with advice is inactive. Instead, the control flow entries and exits are decorated with residues that trigger continuous weaving of advice invocations at dependent shadows. In this case, there is still an element of residual logic required at dependent shadows: the advice must only be invoked when the shadow is reached in the same thread in which the control flow is currently active.

The simplest approach to implementing **cflow** using continuous weaving is to decorate *all* dependent shadows at once, when the control flow is entered by the first thread, and to withdraw the woven code when the control flow is left by all threads. In the spirit of continuous weaving, more fine-grained approaches are imaginable.

The cost imposed on control flow entries and exits, or on parts of the dependent shadows, is as high as that of dynamic weaving. At dependent shadows, an additional overhead is introduced to check for thread applicability.

2.4 AOP Implementations

We will now briefly introduce a number of prominent AOP implementations with special regard to the way each of them implements support for **cflow**. Most of the systems are AOP implementations for the Java platform. We do not claim the list to be complete; it is restricted to typical representatives. The approaches will not be described in depth; for more detailed descriptions, we refer to the particular literature, or to a survey of AOP languages and their implementations [10].

AOP implementations employing the **counter** approach described above are AspectJ [22, 4] and AspectWerkz [9, 6]. AspectJ compiles an extended version of Java to bytecode. It uses ThreadLocal instances to encapsulate **cflow** counters. AspectWerkz supports dynamic weaving, i. e., weaving aspects into a running application. To that end, it prepares applications at at load-time by inserting hooks to which advice invocations can later be added. AspectWerkz follows, in principle, the counters approach, but it always uses a stack to monitor control flows. Control flow checks are implemented by querying the stack's size. The stack is used by default to allow for accessing state from the constituent join points.

The abc compiler [25, 7, 1] also belongs to this group of systems. However, it supports static analysis for the optimisation of **cflow**. The interprocedural analysis implemented in abc [7] exploits a call graph of the entire application, which is why *all* classes of the application must be known at compile-time. For each pointcut expression containing a **cflow** designator, it yields three sets of join point shadows that are then further processed by the weaver.

Based on the example **cflow**(pc1) && pc2, the three sets computed are as follows (in the following, "residues" and "advice invocations" mean those pertaining to the sample pointcut only). The first set contains those shadows of pc2 that *may* occur in a control flow constituted by a shadow of pc1. At the shadows contained in this set, advice invocations must be guarded by residues. At those shadows of pc2 that are *not* contained in the first set, neither residues nor advice invocations need to be woven because they are guaranteed to never be executed inside a control flow pertaining to pc1.

The second set contains those shadows of pc2 that are guaranteed to occur *only* in a control flow constituted by a shadow of pc1. At these shadows, the advice invocation can be woven without being guarded by a residue. At those shadows of pc2 that are *not* contained in the second set, residues are required.

In the third set, those shadows of pc1 (sic) are contained that *may* influence the evaluation of residues at shadows of pc2. At these shadows, residues for counter or stack maintenance must be woven.

The **stack walking** approach is adopted by JAsCo [27, 18] and AspectS [17, 5]. JAsCo is an extention to Java. The residues it weaves at dependent join point shadows create Throwable instances and iterate over the stack trace that can be retrieved from such an instance. AspectS is implemented in Smalltalk and accesses the call stack by means of the thisContext pseudo variable.

Continuous weaving is offered as an alternative in AspectS. It does not only support **cflow**. In fact, it was implemented to support the notion of *morphing aspects* [14] that use continuous weaving in a much wider fashion than it is described above.

3. IMPLEMENTATION IN STEAMLOOM

Steamloom [16, 8] is a Java virtual machine with dedicated support for AOP language mechanisms. It is based on IBM's open source Jikes Research Virtual Machine ("Jikes" for short) [3, 2, 20]. The support Steamloom has for AOP is provided through an API, whose classes are members of the VM itself.

Steamloom supports fully dynamic weaving. That is, it allows for weaving aspects in and out while an application is running. Classes are not prepared with hooks (like in AspectWerkz, for example) at load-time. All weaving takes place entirely at run-time.

In Steamloom, all of the three approaches mentionend in Sec. 2 have been implemented. All of the implementations exploit the fact that Steamloom is a VM extension in that they rely on specific functionality offered by the virtual machine, or in that they themselves integrate part of their functionality in the VM. The three implementations will be presented in the same order as above.

3.1 Counters

For every **cflow** pointcut, Steamloom manages a thread-local counter. It deploys a residue at control flow entries and exits that increments or decrements, respectively, this counter for the current thread. So far, Steamloom's ap-

[1]The term "continuous weaving" was introduced by Hanenberg et al. [14].

proach does not differ from other counter-based implementations.

The difference lies in the nature of the residues. Residues woven at both control flow entries/exits and dependent shadows are calls to methods that are *part of the virtual machine* rather than other application methods. Thus, Steamloom's **cflow** residues are not subject to execution by the VM, but they are executed as a part of the VM's inherent functionality.

Control flow counters are also not maintained at application level. They are stored directly in arrays that are themselves stored in the VM's *internal* representation of Java threads. Storing control flow counters in an array allows for very fast access to them. The array indices for a given **cflow**'s counters are fixed at the time the corresponding aspect is woven into the application code and do not change while the aspect is active. The arrays are resized dynamically and the handles are recycled so that the maximum array size is bounded by the maximum number of control flow pointcuts that are deployed at a given moment in time. Since a particular thread's array is only accessed by that thread, no synchronisation is needed, enabling a lock-free implementation of counter updating and checking residues.

This approach is further optimised by making the residues deployed at dependent join point shadows *thread-local* so that they are, in a given thread, only ever executed when the corresponding control flow has been entered in that thread. Steamloom supports scoping entire aspects to individual threads. Such aspects are active in a given thread *only*. Residues for **cflow** woven at dependent shadows are treated as parts of the advice that are attached to the shadows. Making them thread-local implies that, whenever the control flow counter is incremented from 0 to 1, the thread-local advice is enabled for the current thread and when it is decremented back to 0 it is disabled again.

3.2 Stack Walking

As mentioned in Sec. 2.2, no residues are required at control flow entries and exits when stack walking is used to implement **cflow** checks. Consequently, Steamloom only weaves residues at dependent join point shadows. The residue is, like seen above with the counter approach, a direct call into the virtual machine.

A so-called "stack frame matcher" is created for a **cflow** designator when the aspect containing a pointcut with that designator is woven into the application. From the pointcut designator, the matcher builds, internally, a stack pattern that represents the stack layout (in terms of methods on the call stack) that must be met in order for the constituent pointcut to match. In case of nested control flows, the pattern contains the methods constituting the nested control flow in the given order. Each entry of the pattern can – if the corresponding constituent pointcut contains wildcards – match multiple methods.

The matching process extracts the IDs of compiled methods from the VM-internal stack frames. From the ID of a compiled method, the VM-internal representation of the method is resolved. The methods retrieved from the stack frames are subsequently matched against the elements of the stack pattern to check. As soon as the pattern is safely identified, the process stops, and the advice can be invoked.

3.3 Continuous Weaving

Continuous weaving as implemented in Steamloom follows the simple approach mentioned in Sec. 2.3, where all dependent shadows are immediately decorated when the control flow is entered.

When an aspect unit containing a **cflow** pointcut is deployed, the entry and exit shadows are decorated with residues that notify Steamloom's dynamic weaver of the entry or exit. The residues are, as in the two preceding solutions, VM methods. Apart from triggering continuous weaving, the entry and exit residues also have to update counters. Counters are still required because a control flow may be entered recursively; the dependent shadows must be kept decorated until the last activation of the respective control flow is left.

Unlike the constituent shadows, *most* of the dependent shadows are *not* decorated. *Some* are, which is due to a small optimisation that will be explained below. Normally, dependent shadows are completely unaffected by residues pertaining to a **cflow** pointcut until the first thread enters the control flow, which triggers the deployment of residues and advice invocations at dependent shadows. All other threads that enter the control flow merely lead to the respective aspects being marked as applicable to those threads (see the above remarks for thread-local aspects). Consequently, code woven at the dependent shadows is not removed as long as *any* thread is inside the control flow. When the last thread leaves the control flow, residues and advice invocations at dependent shadows are removed.

The aforementioned optimisation applied in the implementation of continuous **cflow** weaving is applied when a situation like the following one is met. When a pointcut like

```
1  cflow(execution(void X.m())) && call(void Y.n())
```

is used *and* the method X.m() contains calls to Y.n(), then the method whose execution constitutes the control flow also contains dependent shadows.

Normally, woven code inserted before and after the execution of X.m() would trigger dynamic weaving of residues and advice invocations at dependent shadows. However, since the method X.m() itself contains such shadows, they can right away be decorated as well. This is just what the optimisation is about. It applies a very simple form of static analysis and determines exactly those dependent shadows that are contained in methods constituting the control flow. That way, an unnecessary weaving step can be avoided for dependent shadows.

4. EVALUATION

This section evaluates and discusses several implementations of **cflow**. We do not only regard the three implementations that have been achieved in Steamloom, but also take other AOP systems into account.

For the evaluation, several performance benchmarks are used. They have various levels of complexity and target different characteristics of **cflow** implementations. The first benchmark measures the cost of executing join point shadows *inside* versus *outside* a control flow. The second benchmark highlights the scalability of **cflow** implementations with regard to varying numbers of (a) control flow entries/exits, (b) dependent join point invocations, and (c) threads. The third benchmark targets nested control flows.

The Java 5 standard VM was used to run the benchmarks on the following systems: AspectJ 5 milestone 4, `abc` 1.1.0,

JAsCo 0.8.6, and AspectWerkz 2.0. Jikes 2.3.1 was used to run the benchmarks on the following systems: Steamloom (static counters, stack walking, continuous weaving), AspectJ 1.2.1, and `abc` 1.1.0. AspectJ and `abc` were measured on Jikes in addition to the Java 5 standard VM to gain results that are better comparable to the performance of Steamloom. AspectS is excluded from the measurements, because it does not run on the Java platform and thus delivers no comparable results. All measurements were made on a Dual Xeon workstation (2x3 GHz) with 2 GB RAM running Linux 2.4.23.

4.1 Simple Micro-Measurements

Based on the JavaGrande benchmark framework [11, 19], we have implemented a micro-measurement suite [15] that measures the performance of certain kinds of join points when advice are attached to them. Measurement results are expressed in operations per second, i. e., in the number of join point executions including advice invocations that are performed per second. All advice just increment a counter. The measured join points are minimal as well; in the case of method calls and executions, the corresponding methods also only increment a counter.

This micro-measurement suite is used here to measure the throughput of method execution join points in the presence of **cflow** pointcuts. Method executions were chosen because they are supported by all AOP implementations in focus. The entry point of the measurement is the JGFrun() method in the measurement class. The entire measurement is run twice: the first run is started by simply invoking JGFrun(), the second by invoking CFlow(), which calls the former. In the measurement, a pointcut is used that matches the measured method executions only if they occur in the control flow of the execution of CFlow().

This benchmark is indeed very simple in that the control flow is only ever entered and left exactly once. So, the performance measured here is basically that of residues at dependent shadows.

Results from the measurement are shown in Fig. 1. The standard JVM-based systems are represented by the left bar chart; the systems running on Jikes by the right. The "execution plain" bar is given for all systems to show how many method executions the plain run-time environment (Java 5 or Jikes 2.3.1) can perform in a second.

It is immediately visible that the approaches using stack walking perform worst. Still, the stack walking implementation based on Steamloom (abbreviated "SL" in the figure) performs considerably better than JAsCo, which creates Throwable instances to access the call stack. Inside the virtual machine, the call stack is immediately available for access, while a representation must be expensively created when it is accessed at application level.

Even though stack walking clearly benefits from VM integration, it still performs significantly worse than *all* other approaches. The conceptual benefit of stack walking, requiring residues *only* at dependent join point shadows, is annulled by the high cost of the residues.

The counter-based approaches all perform better. Steamloom has the most efficient of all counter implementations. It even outperforms AspectJ running on the standard VM. Basically, both approaches use thread-local counters to monitor control flows, but AspectJ does so at application level using ThreadLocal instances, while Steamloom directly associates the counters with the VM's *internal* representation of threads. Residues checking the counters also are immediate calls *into the VM* in Steamloom, while AspectJ executes all checks at application level.

When compared to other approaches running on Jikes, Steamloom's counters still perform better than those of `abc`, though only slightly. Apparently, the VM-integrated support for counters still has a larger positive impact on performance than static optimisation. Still, this benchmark is, due to its simple nature, not ideal for judging about Steamloom's counters versus `abc`'s optimisation.

Continuous weaving performs extremely well. Given the nature of this benchmark (the control flow is only ever entered and left once), this was to be expected; this benchmark is not suited to yield accurate results concerning continuous weaving. The benchmark results described in the following subsection will give better insights on its performance, since they enter and leave control flows more often.

4.2 Variability Benchmark

The purpose of this benchmark application is to measure how the performance of the introduced **cflow** implementations varies depending on the number of control flow entries/exits and on the number of dependent join point shadow occurrences in- and outside of the control flow. In addition, it evaluates the scalability of **cflow** implementations with an increasing number of threads.

The Benchmark. The core measurement class is shown in Lst. 2. The aspect applied to the benchmark class is also contained in the listing, starting at line 44. As can be seen from its pointcut, the control flow entries and exits in the benchmark are defined by executions of the method CflowBenchmark.m(). The dependent join point shadows are calls to CflowBenchmark.x().

The benchmark consists of two nested loops. The outer loop is implemented in CflowBenchmark.test() and controls the number of control flow entries and exits during a benchmark run. The inner loop controls the number of executions of the dependent join point shadow. It is implemented in CflowBenchmark.m() and m0(), respectively, for reasons that will be described below.

The CflowBenchmark.test() method is the entry into the benchmark. It accepts four parameters, each of which controls a certain facet of the benchmark behaviour.

The outer value denotes the number of iterations of the outer loop. The additional parameter freq_cflow defines the fraction of those iterations that actually enter and exit the control flow. The inner value controls the number of iterations in the inner loop. Finally, the freq_dep parameter controls the actual number of executions of the dependent join point shadow, by defining the fraction of entered/exited control flows in whose context a dependent shadow is actually reached.

Fig. 2 shows an abstracted sample run of the measurement in one thread. Each box in the vertical bar represents one iteration of the outer loop. Similarly, a box in one of the horizontal bars represents an iteration of the inner loop. Both outer and inner are set to 10. The other two parameters freq_cflow and freq_dep are set to 2 and 3, respectively. Where an iteration of the outer loop enters the control flow, the corresponding box in the vertical bar is filled grey. For the given values, every second iteration actually enters the control flow; every third iteration of the outer loop leads to

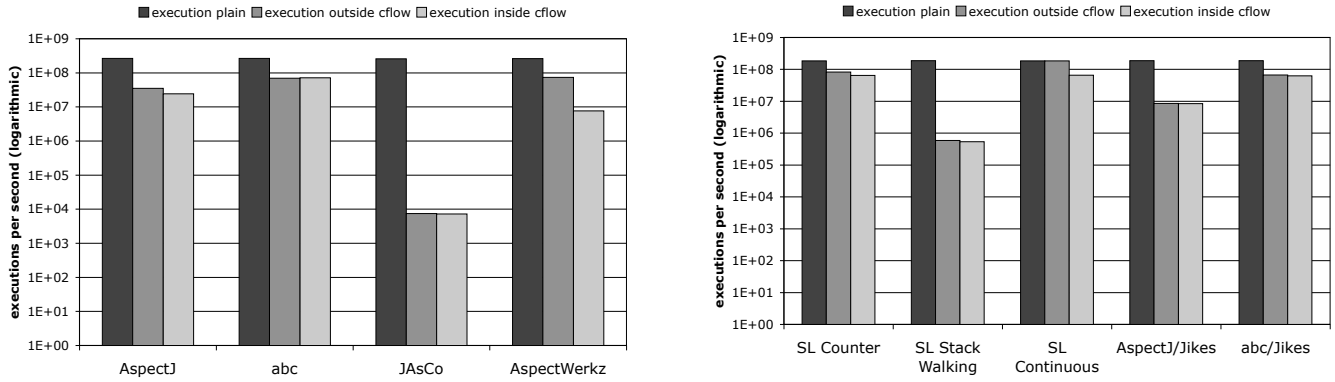

Figure 1: Results from micro-measurements for Java 5 (left) and Jikes-based (right) systems.

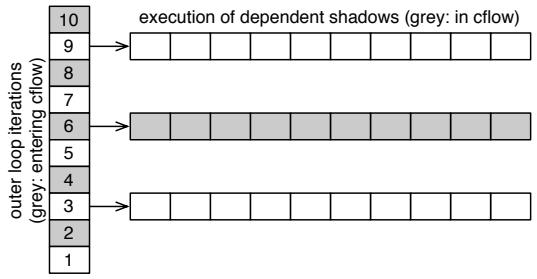

Figure 2: Execution of the variability benchmark.

actual executions of the inner loop, and thereby to potential executions of the dependent shadow. Where the latter are actually executed inside the control flow, the corresponding boxes are also marked grey.

The measurements were not run on all systems mentioned in Sec. 2.4. Those with extremely poor performance due to stack walking, namely JAsCo, PROSE, and Spring AOP, were excluded. Of the stack walking approaches, only Steamloom's was kept as a representative with better performance. The measurements were run with 1,000 outer and inner loop iterations, and for all combinations of 1, 2, 5, 10, 100, and 1,000 for the two frequencies and 1, 5, 10, 15, and 20 for the number of threads. A representative selection of the results is discussed in the following.

The diagrams pertaining to this discussion (Figs. 3–6) should be read as follows. Each diagram represents the results for a certain measurement point $(freq\_cflow, freq\_dep)$. In each diagram, the logarithmic y axis denotes the time (in milliseconds) needed to execute the benchmark, while the x axis denotes the number of threads. In addition to the results for the measured AOP implementations, results for plain Java are given as a reference.

Discussion of Results: Continuous Weaving and Stack Walking. From the figures, it is immediately evident that continuous weaving (indicated by "SL Continuous") is prohibitively expensive when control flows are entered and left frequently. The only scenario in which it ranges in the average is the measurement point set in Fig. 4, where the frequency of dependent shadows is significantly higher than that of control flows. This was to be expected: recompilation is expensive, and when it is done frequently, overall performance suffers. Stack walking ("SL Stack W.") is extremely expensive in exactly this scenario. This was also to be expected: when the control flow is not active, the entire stack must be walked to yield a negative result. Matching is done much faster when it is successful, as in the first scenario.

Continuous weaving and stack walking are suited for opposite scenarios with extreme differences between the frequencies of control flows and dependent shadows. This can also be seen from the results pertaining to these two strategies in scenarios with the same frequency of control flows and dependent shadows (Figs. 5 and 6).

Discussion of Results: Counter-Based Implementations. Counter-based approaches generally yield the best performance. Differences lie in the particular implementations of counter storage and management. The effects of different implementation approaches become especially visible when the two scenarios with different frequencies of control flows and dependent shadows (Figs. 3 and 4) are regarded.

It is a characteristic of the $(5, 100)$ scenario that the operation that is most frequently executed in it is control flow counter update. Conversely, the $(100, 5)$ scenario's characteristic is that the most frequent operation is the counter check. Thus, the performance of systems in the first scenario is an indicator for the efficiency of counter management for updating, and the performance in the second scenario indicates the efficiency of counter checks.

Of the counter approaches, `abc` and Steamloom provide the most efficient implementations for both counter updates and checks, which Figs. 3 and 4 indicate. Steamloom benefits from maintaining counters directly in the VM's internal representation of threads, and `abc` profits from avoiding unnecessary updates and checks due to static optimisation.

AspectWerkz reveals a weakness in counter check efficiency (Fig. 4); its performance is almost the same as that of Steamloom's continuous weaving. This due to its implementation approach; it does not merely encapsulate counters, but maintains stack objects, and a control flow check needs to determine whether the size of the stack is larger than zero. This operation is more expensive than a simple arithmetic comparison on numbers.

Scalability with the Number of Threads. An interesting aspect with regard to the efficiency of **cflow**-related operations is their scaling behaviour when multiple threads are involved. The systems based on Jikes, independently of whether AspectJ, `abc`, or Steamloom with counters are regarded, generally are the most moderate ones in this re-

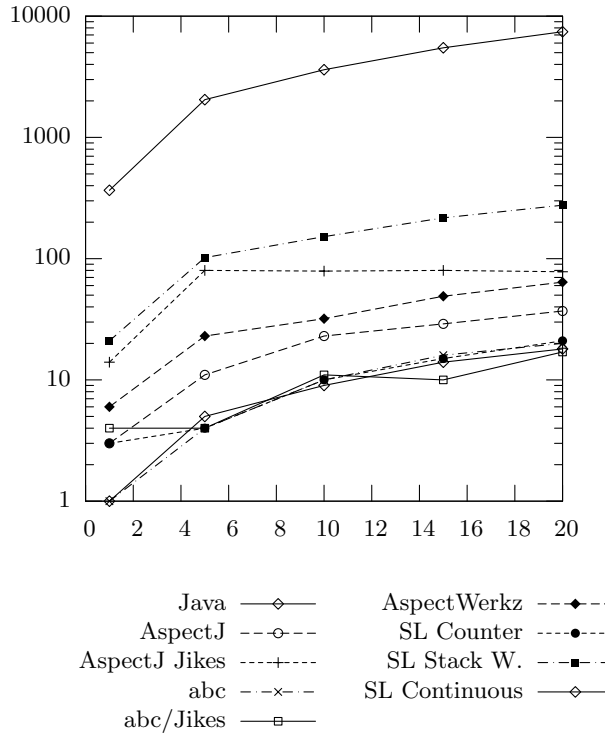

Figure 3: Variability results (5,100): frequent control flows, infrequent dependent join points.

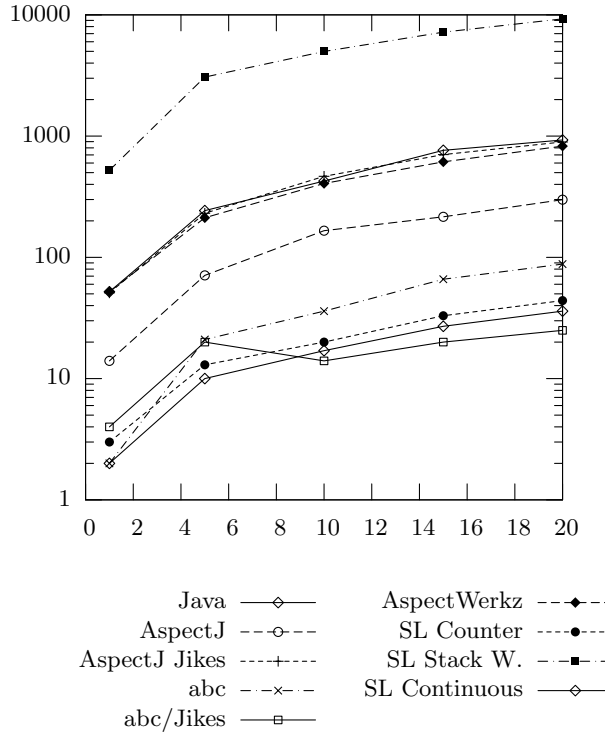

Figure 4: Variability results (100,5): infrequent control flows, frequent dependent join points.

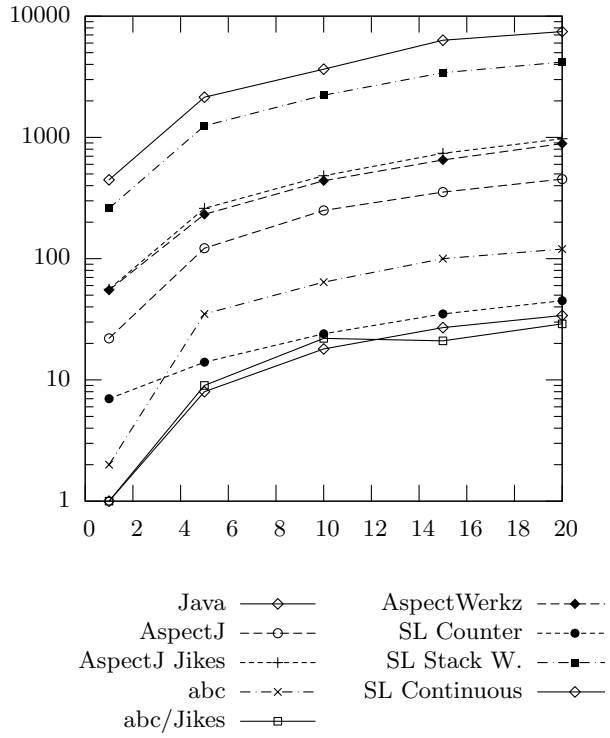

Figure 5: Variability results (5,5): frequent control flows and dependent join points.

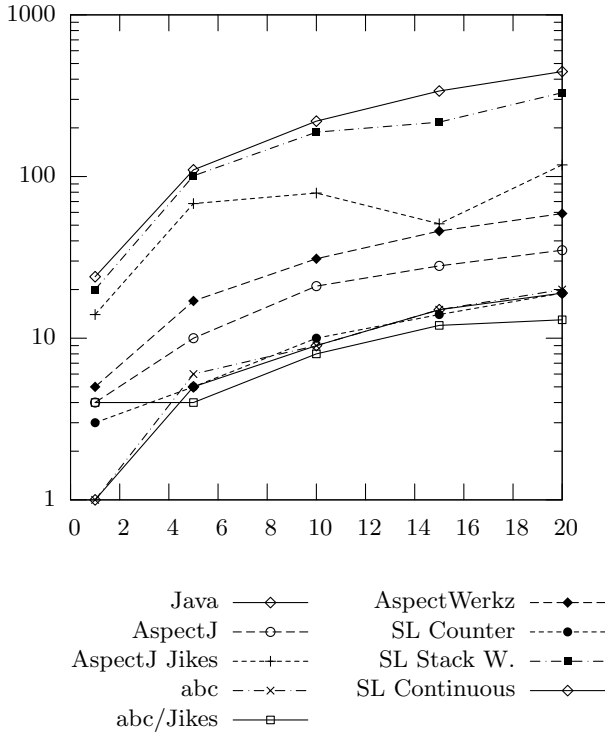

Figure 6: Variability results (100,100): infrequent control flows and dependent join points.

```
1  public class CflowBenchmark {
2    int entries, deps;
3    public void test(
4      int outer, int freq_cflow, int inner,
5      int freq_dep
6    ) {
7      while (outer-- != 0) {
8        int real_inner =
9          (outer % freq_dep == 0) ? inner : 0;
10       if (outer % freq_cflow == 0) {
11         m(real_inner);
12       } else {
13         m0(real_inner);
14       }
15       Thread.yield();
16     }
17   }
18   public void m(int runs) {
19     entries++;
20     while (runs-- != 0) {
21       foo();
22       x();
23     }
24   }
25   public void m0(int runs) {
26     while (runs-- != 0) {
27       foo();
28       x();
29     }
30   }
31   public void foo() {
32     x();
33   }
34   public void x() {
35     ++deps;
36   }
37 }
38
39 public aspect CflowAspect {
40   int ctr;
41   before():
42     cflow(execution(* CflowBenchmark.m(int))) &&
43     call(* CflowBenchmark.x()) {
44       ctr++;
45   }
46 }
```

Listing 2: Source code of the cflow benchmark.

spect. Their moderate scaling behaviour hence is not due to their specific implementations, but to the scheduler of Jikes, and to the way the access to thread-local state (even for ThreadLocal instances) is implemented in its class library, which is GNU Classpath [13]. Steamloom's integrated approach thus does not affect the scaling behaviour as much as it affects the performance of residue executions.

4.3 Nested Control Flow Benchmark

The purpose of this benchmark is to measure the performance of **cflow** implementations when nested **cflow** statements are used. In the benchmark, ten methods f0()–f9() invoke each other *recursively* in all possible permutations. The last method in the row always invokes a method foo().

The benchmark applies an aspect with a pointcut/advice combination like the one in Lst. 3 to the application. It nests ten **cflow** pointcuts in some order and attaches an advice to the execution of the foo() method.

The semantics of this aspect is that the advice will be executed *only* if the ten f ...() methods are on the call stack in *exactly* the order determined by the nested **cflow** designators. That is, the advice will, during the benchmark, be executed exactly once, when the correct permutation of

```
1  before():
2    cflow(execution(void X.f0()) &&
3      cflow(execution(void X.f1()) &&
4        cflow(execution(void X.f2()) &&
5          cflow(execution(void X.f3()) &&
6            cflow(execution(void X.f4()) &&
7              cflow(execution(void X.f5()) &&
8                cflow(execution(void X.f6()) &&
9                  cflow(execution(void X.f7()) &&
10                   cflow(execution(void X.f8()) &&
11                     cflow(execution(void X.f9())
12   ))))))))))) && execution(void X.foo()) {
13     X.count++;
14   }
```

Listing 3: An aspect with nested cflow pointcuts.

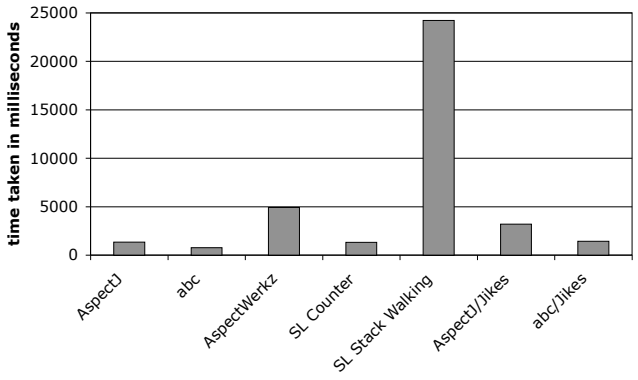

Figure 7: Results from the nested control flow benchmark.

f ...() methods is on the stack.

This benchmark measures the time the application takes to run. It yields information on the cost of entering and leaving control flows, and also on the cost of checking **cflow** matches when nested control flows are on hand.

Results for the nested control flow benchmark are shown in Fig. 7. This measurement was not applied to the continuous weaving approach of Steamloom, because the extremely high frequency of entering and leaving control flows, which has already led to bad results in the previous benchmark, suggests that its performance is extremely weak in such circumstances[2].

The results show once more that the stack walking approach which has already proven to be suboptimal in the simple control flow benchmarks is indeed not optimal, even if support for it is integrated in the virtual machine. The cost of walking the stack is too high, and non-matching stacks impose a high cost on the matcher.

All other approaches that use counters perform better. AspectWerkz suffers from its expensive counter management strategy using a stack by default. Steamloom once more benefits from the integration of counter storage with VM-internal data structures. It even performs better than AspectJ on the Sun standard VM. Put in relation to AspectJ running on Jikes, it becomes even more obvious that Steamloom's approach is beneficial.

The code generated by abc clearly benefits from the op-

[2]In fact, the benchmark was, as an experiment, run on the continuous weaving implementation; the process was terminated after several minutes.

timisation at compile-time. Both on the Java 5 VM and on Jikes, the `abc` code is about twice as fast as the AspectJ code. It is, however, interesting to note that Steamloom's counter approach performs *slightly* better than `abc` on Jikes. The benefits gained by static analysis and optimisation are obvious, but those gained by integrating support for control flow counter management in the VM itself still have an, if slightly, more significant impact.

4.4 Summary

The results of the analysis can be summarised as follows with regard to comparing the three strategies. Stack walking has not proven to be a reasonable solution since its complexity depends on the stack depth met at join point shadows. Continuous weaving does not yield satisfactory performance unless control flows are *very* infrequently entered and left. Counters, exhibiting constant cost at control flow entries and exits as well as dependent shadows, appear to be the best solution for matching control flows. When they are given dedicated support from the run-time environment, they even gain some more efficiency.

The above results suggest that VM integration of support for dynamic pointcuts is promising. The results measured for `abc` and Steamloom suggest that a combination of VM integration with interprocedural analysis will yield even better results. By such a combination, we do not mean a combination of static analysis and VM integration, though. There are two reasons for which this does not seem optimal.

First, the closed-world assumption imposed on the application makes a large number of interesting applications infeasible. A very prominent field of AOP adoption is middleware, where AOP is used to transparently decorate objects with services. The Spring framework [21, 26] is a good example of a lightweight approach to middleware that does so. In the context of middleware, dynamic deployment of entire applications is an important feature. When the entire application must be known at compile-time to perform a whole-program analysis, dynamic deployment is no longer an option.

Second, the whole-program analysis performed by `abc` to optimise run-time performance significantly slows down the compilation process. For example, the application for the variability benchmark takes the AspectJ 1.2.1 compiler 1.6 seconds to compile. For `abc` without optimisations, compilation takes 3.6 seconds. With interprocedural analysis, `abc` needs three minutes.

So, instead of merely combining optimised compilers and VM integration, we opt for exploiting even more of the VM's internal structures to achieve *dynamic optimisation*. The VM maintains a call graph internally, which can also be used to perform interprocedural control-flow analysis. The facilities for performing such analyses exist in the VM, where they are normally exploited by the optimising compiler.

5. CONCLUSION

There are two main conclusions that we draw from the results of the paper.

Firstly, the most important result is that VM integration of support for dynamic pointcuts is a research path worth pursuing. We showed that both the counter-based as well as the stack walking implementation strategies exhibit increased performance when support for them is integrated into the VM. An important point to make in this context is that dedicated support from the run-time environment makes the implementation of **cflow** at least as efficient as the `abc` implementation. The latter provides one of the most efficient implementations available today. However, this is achieved at the cost of (a) significant increase of the compilation time and (b) placing Java applications under a closed-world assumption that contradicts Java's dynamic class loading capabilities. The VM-integrated approach does not suffer from any of these problems. By exploiting analyses of the dynamic control call graph in the VM, we hope to be able to further improve the performance in the future.

Secondly, by implementing all three approaches within the same environment, we have shown that counters, exhibiting constant cost at control flow entries and exits as well as dependent shadows, appear to be the best solution for matching active control flows. Stack walking does not seem to be an alternative since its complexity depends on the stack depth met at join point shadows. Continuous weaving does not yield satisfactory performance unless control flows are *very* infrequently entered and left.

Acknowledgements

This work was supported by the AOSD-Europe Network of Excellence, European Union grant no. FP6-2003-IST-2-004349.

6. REFERENCES

[1] abc (AspectBench Compiler) Home Page. `http://aspectbench.org/`.

[2] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*. ACM Press, 1999.

[3] B. Alpern et al. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[4] AspectJ Home Page. `http://www.eclipse.org/aspectj/`.

[5] AspectS Home Page. `http://www-ia.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/`.

[6] AspectWerkz Home Page. `http://aspectwerkz.codehaus.org/`.

[7] P. Avgustinov et al. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 117–128. ACM Press, 2005.

[8] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proc. AOSD 2004*. ACM Press, 2004.

[9] J. Bonér. What Are the Key Issues for Commercial AOP Use: how Does AspectWerkz Address Them? In *Proc. AOSD 2004*, pages 5–6. ACM Press, 2004.

[10] J. Brichau and M. Haupt (editors). Survey of Aspect-Oriented Languages and Execution Models. `http://aosd-europe.net/documents/aspLang.pdf`, AOSD-Europe Network of Excellence, 2005.

[11] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[12] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[13] GNU Classpath Home Page. `http://www.gnu.org/software/classpath/classpath.html`.

[14] S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing Aspects: Incompletely Woven Aspects and Continuous Weaving. In *Proc. AOSD 2004*. ACM Press, 2004.

[15] M. Haupt and M. Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. In M. Weske and P. Liggesmeyer, editors, *Proc. Net.ObjectDays 2004*, volume 3263 of *LNCS*. Springer, 2004.

[16] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An Execution Layer for Aspect-Oriented Programming Languages. In *Proc. VEE 2005*. ACM Press, June 2005.

[17] R. Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *LNCS*, pages 216–232. Springer, 2003.

[18] JAsCo Home Page. `http://ssel.vub.ac.be/jasco/`.

[19] JavaGrande Benchmarks Home Page. `http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/`.

[20] The Jikes Research Virtual Machine. `http://jikesrvm.sourceforge.net/`.

[21] R. Johnson and J. Hoeller. *Expert One-on-One J2EE Development without EJB*. Wiley, 2004.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[24] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In G. Hedin, editor, *Proc. CC 2003*, volume 2622 of *LNCS*, pages 46–60. Springer, 2003.

[25] P. Avgustinov and others. abc: an Extensible AspectJ Compiler. pages 87–98. In [28].

[26] Spring Home Page. `http://www.springframework.org/`.

[27] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented Approach Tailored for Component Based Software Development. In *Proc. AOSD 2003*, pages 21–29, 2003.

[28] P. Tarr, editor. *Aspect-Oriented Software Development. Proceedings of the 4th International Conference on Aspect-Oriented Software Development*. ACM Press, 2005.