Efficient Control Flow Quantification

Christoph Bockisch* Sebastian Kanthak* Michael Haupt*,† Matthew Arnold‡ Mira Mezini*

*Software Technology Group
Darmstadt University of Technology, Germany
†Software Architecture Research Group
Hasso Plattner Institute for Software Systems Engineering, Potsdam, Germany
‡IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
{bockisch,mezini}@informatik.tu-darmstadt.de
kanthak@st.informatik.tu-darmstadt.de
michael.haupt@hpi.uni-potsdam.de, marnold@us.ibm.com

Abstract

Aspect-oriented programming (AOP) is increasingly gaining in popularity. However, the focus of aspect-oriented language research has been mostly on language design issues; efficient implementation techniques have been less popular. As a result, the performance of certain AOP constructs is still poor. This is in particular true for constructs that rely on dynamic properties of the execution (e. g., the `cflow` construct).

In this paper, we present efficient implementation techniques for `cflow` that exploit direct access to internal structures of the virtual machine running an application, such as the call stack, as well as the integration of these techniques into the just-in-time compiler code generation process.

Our results show that AOP has the potential to make programs that need to define control flow-dependent behavior not only more modular but also more efficient. By making means for control flow-dependent behavior part of the language, AOP opens the possibility of applying sophisticated compiler optimizations that are out of reach for application programmers.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—run-time environments

General Terms Languages, Measurement, Performance

Keywords Aspect-oriented programming, virtual machine support, control flow

1. Introduction

The aspect-oriented programming (AOP) paradigm [23, 14] introduces a new kind of modules called *aspects* that allow for capturing crosscutting concerns in a localized way and with explicit interfaces to the rest of the system. Aspect-oriented programming languages introduce the following notions [22].

Crosscutting behavior encapsulated in aspects is seen 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 are queries over the execution of a program. Whenever a pointcut matches, an *advice* associated with the matching pointcut is executed. Advice are method-like constructs.

The technique used to implement such a model is called *weaving*, denoting that an application's and its crosscutting concerns' control flows are interwoven. Several approaches to implement weaving have been devised [12]. The most common approach today is *bytecode weaving*, where advice invocations are inserted into application bytecode at locations called *join point shadows* [27]. Join point shadows are code structures (expressions, statements or blocks) that possibly yield join points during execution, e. g., the shadow of a method call is a call instruction. Join point shadows are determined by statically evaluating pointcuts.

Pointcuts that quantify over dynamic properties of join points cannot definitely be mapped to code locations. For example, in AspectJ[1] [22, 5], pointcuts exist that select join points depending on the current control flow. The control flow can be quantified over by means of the `cflow` dynamic pointcut. The efficiency of implementations supporting this pointcut is in the focus of this paper.

Researchers and programmers have already identified several useful usage scenarios for `cflow` pointcuts. It can be used to only execute advice the first time some code section is entered, but not when it is entered recursively, e. g., to implement authentication. Another use case is to restrict advice execution so that it does not (or does only) take effect if some sub-system of the application is currently acting.

For example, the Law of Demeter (LoD) aspect [25] advises, among others, all base application method executions. In the advice, it stores objects captured from the application into a hash map, which causes their `hashcode()` method to be executed. Normally, this would recursively trigger another advice execution. To avoid an endless recursion, the LoD aspect does not advise all method executions, but only all executions for which no advice from the LoD aspect is currently executing on the call stack. Note that the `hashcode()` method is called indirectly through Java's collection library. Thus, it would not be sufficient to skip advice execution if the direct caller of a method is the LoD aspect. Instead, if a method

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

[1] An aspect-oriented extension to Java.

```
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`.

execution is in the control flow of some LoD method directly or indirectly, advice must not be executed.

Laddad [24] gives an extensive discussion of the `cflow` pointcut. He presents transaction management, policy enforcement and controlling tracing as examples. `cflow` is also used in industrial projects that are implemented in AspectJ. The Glassbox Corporation's *Glassbox Inspector* [15] (a trouble-shooting tool for enterprise applications), employs `cflow` to, among others, restrict advice to the top-level call of methods, excluding recursive calls.

For illustration of the possible implementations of the `cflow` pointcut, consider the classes in Lst. 1. All three methods in `A` invoke methods on an instance of `B`, and both these 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, but *only* if `k()` is called in the course of an execution of `A.m()`. The corresponding pointcut, matching all calls to `B.k()` that occur in the control flow of the execution of `A.m()`, looks as follows:

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

It cannot be determined statically whether a particular call to `k()` is inside a certain control flow. Hence, the weaver generates pieces of conditional logic called *residues* that are woven into application code at join point shadows for calls to `B.k()`. A residue queries meta data to ensure that the advice is only invoked when the dynamic condition is satisfied.

Different AOP implementations have different ways of implementing these residues (details will be given in Sec. 2). The majority of existing AOP systems implement them as calls to the particular system's AOP infrastructure, and these calls are woven into the application at join point shadows. This has the effect that the residues are 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 data structures connected with residues [13].

There are several ways to reduce the overhead introduced by these residues. The `abc` compiler [28, 8, 1] of the AspectJ language employs static intra- and interprocedural analyses to reduce the number of required residues. The `abc` compiler also optimizes code that ensures thread-safety for `cflow` pointcuts. The code produced by `abc` is considerably faster [8] than that of `ajc`, the standard AspectJ compiler. The `abc` compiler performs an interprocedural analysis only at its highest optimization level as it is very time- and memory-intensive. It depends on a *whole-program analysis* that needs to know all possible entry points and the class files for all classes reachable from there. This places Java applications under a closed-world assumption that contradicts Java's dynamic class loading capabilities.

The closed world assumption is particularly hard to bear in the area of middleware containers, for which AOP has been recognized as a great tool to reduce the complexity of transparent service injection; such containers heavily rely on dynamic deployment of business applications.

Our work on improving the efficiency of aspect-oriented programs has followed another path: integrating support for aspect-oriented mechanisms directly at the execution layer, i. e., in the virtual machine. The underlying rationale is as follows:

- The VM maintains some dynamic model of the execution as it executes the code, over which we can potentially directly quantify.
- When required information is not directly accessible and we need to construct and maintain it, the supporting infrastructure can be implemented more efficiently within the VM.

For the validation of our hypothesis, we have been working on *Steamloom* [17, 9], a Java virtual machine with dedicated support for AOP mechanisms, which is based on IBM's Jikes Research Virtual Machine (RVM) [3, 2, 21].

With Steamloom, we have already shown that supporting dynamic aspect weaving within the virtual machine is beneficial [17]. However, our previous work was limited with regard to its depth of integrating aspect-oriented concepts into the virtual machine. This is especially obvious regarding the treatment of residues for quantifying over dynamic properties of join points. So far, Steamloom's treatment of residues is by weaving them as bytecodes into application bytecodes; hence, it does not significantly differ from other aspect-oriented implementations, except for the fact that the VM weaves directly, rather than relying on a third-party bytecode manipulation tool.

This paper goes a significant step forward in deeper integration of support for aspect-oriented mechanisms within the virtual machine. Specifically, this paper focuses on the efficient implementation of quantification over control flows by means of the `cflow` pointcut.

The contributions of the paper are as follows:

1. We show how existing implementation strategies for `cflow` can be made more efficient by exploiting direct access to internal structures of the VM, such as the call stack.

   Specifically, we consider two existing implementation strategies for `cflow`: (a) the counter-based approach used by the AspectJ compilers [22, 5, 28, 1], and (b) the stack walking approach used by some other aspect-oriented languages and frameworks, e. g., JAsCo [32, 19] and JBoss AOP [20].

2. We present a new approach to implement `cflow`, which integrates support for it directly into the just-in-time (JIT) compilers of the Jikes RVM that dynamically translate Java bytecodes into efficient native machine code. This way, we offer the virtual machine a better opportunity for optimizations.

3. Finally, we would like to draw the attention to another more conceptual contribution of the paper. The efficiency improvements that result from the integration of support for `cflow` into the JIT compilers emphasize an advantage of aspect-oriented quantification mechanisms that has been overlooked so far.

   Hitherto, increased modularity has been the main argument for AOP. While this is certainly the key benefit of AOP, this paper shows that AOP also has the potential to make programs that need to define control flow-dependent behavior more efficient. The idea is that by making means for such control flow-dependent behavior part of the language, AOP opens the possibility of applying sophisticated compiler optimizations that are out of reach for application programmers. Positive effects of AOP adoption on performance have previously also been observed in the setting of parallel applications [16].

The structure of this paper is as follows. In Sec. 2, we will first abstractly outline common implementation strategies for `cflow`. For each strategy, we will briefly present current AOP implementations adopting it, and show how it has been realized in Steamloom. Sec. 3 covers our main contribution, a novel implementation strategy for `cflow`. The performance of all approaches presented in this paper is evaluated in Sec. 4. This section is also where the different approaches are discussed. Sec. 5 presents a discussion of future work and Sec. 6 concludes the paper.

2. Cflow Implementations

We start this discussion by a short introduction to the terminology used in the remainder of the paper. The following subsections each present a particular approach to addressing control flow matching.

Each of the approaches is described in generalized form, followed by a brief description of concrete AOP implementations employing it. We do not claim to provide a complete overview of existing AOP implementations; the systems described are typical representatives. The approaches will also not be described in depth; for more detailed descriptions, we refer to particular literature, or to a survey of AOP languages and their implementations [11]. For each of the particular approaches, we also give a description of its implementation in Steamloom.

2.1 Cflow Terminology

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[2].

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

1. At constituent shadows, 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. This is 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 contexts of constituent join points, and to use such context information in advice attached to dependent join points, e. g., the object that was active when the control flow was constituted. We do not provide an implementation of this feature, but discuss a possible implementation in Sec. 5.1. Although some of the approaches we compare our implementation with provide support for such context passing, they all provide a more efficient infrastructure for `cflow` when no use is made of context passing.

2.2 Counters

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

Control flow counters exist once per control flow. Furthermore, they must exist once per thread for this approach to work; otherwise, different threads entering and leaving the same control flow could easily corrupt control flow counter state.

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

2.2.1 Adoption

AOP implementations employing the counter approach described above are AspectJ [22, 5] and AspectWerkz [10, 7]. Both available AspectJ compilers (`ajc` and `abc`) compile AspectJ programs to Java bytecode, generating infrastructural code that uses `ThreadLocal` instances to encapsulate `cflow` counters. In principle, AspectWerkz also follows the counters approach, but it always uses a stack to monitor control flows. The stack is used by default to allow for accessing state from the constituent join points. Control flow checks are implemented by querying the stack's size.

The `abc` compiler [28, 8, 1] largely adopts the counter-based approach similar to `ajc`. However, it adds several optimizations. Thread-local counters are optimized for the first application thread, so that accessing the counter via a `ThreadLocal` instance is avoided for this thread. This facilitates a very quick retrieval of a counter object for single-threaded applications. Multi-threaded applications still have to use a `ThreadLocal` instance for counter management. Code generated by `ajc` always relies on `ThreadLocal` instances.

Moreover, the `abc` compiler provides intra- and interprocedural optimizations to improve the performance of code conjoined with `cflow` pointcuts. Both optimization types are achieved using static analysis [31]. The `abc` compiler performs the time- and memory-intensive interprocedural analysis only at its highest optimization level.

Of the *intra*procedural optimizations `abc` applies, only one is of further interest with regard to this paper. Others either deal with binding parameters from constituent pointcuts, which is out of the focus of this paper, or have been described above. In fact, the counter approach used in recent versions of `ajc` was first introduced in `abc`.

The remaining intraprocedural optimization employed in `abc` is the reuse of counters in methods [8]. From the observation that retrieving a counter from thread-local storage can be expensive, the implementors of `abc` have derived the following optimization. Whenever a control flow counter is required several times in a method (e. g., in a loop or at constituent shadows for control flow entry and exit) the counter is shared in a local variable and has to be retrieved only once. Since local variables are implicitly thread-local, this optimization is obviously correct.

As a result of *inter*procedural analysis, `abc` can completely avoid weaving `cflow` infrastructure at some join point shadows: interprocedural analysis [8] exploits a call graph of the entire application, which is why *all* classes reachable from the application's entry points must be known at compile-time. In particular, if the virtual machine dynamically loads classes that are not known at compile-time, new execution paths may be possible due to late binding of method calls in Java. If this happens, interprocedural analysis becomes unsound. In the following, we will give a brief overview of how `abc`'s interprocedural analysis works.

For each pointcut expression containing a `cflow` designator, analysis yields three sets of join point shadows that are then further processed by the weaver. For the example `cflow(pc1) && pc2`, the three computed sets 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

[2] In [8], a terminology using the corresponding terms *update* and *query* shadows has been introduced.

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` 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.

2.2.2 Realization in Steamloom

Steamloom's approach differs from other counter-based implementations in how residues are implemented, and how thread-local counters are maintained. Residues woven at both constituent 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 synchronization is needed, enabling a lock-free implementation of counter updating and checking residues.

To reduce the cost of retrieving a counter even more, we have implemented the counter sharing proposed by `abc` [8] in Steamloom as well.

2.3 Stack Walking

The stack walking approach does not require 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.

There is no cost at control flow entries and exits connected with stack walking. However, the cost imposed on dependent shadows directly depends 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 currently *in*active.

2.3.1 Adoption

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. JAsCo [32, 19], an extension to Java, and JBoss AOP [20] follow this approach.

In Smalltalk, the call stack is immediately accessible due to the reflective nature of the language. AspectS [18, 6], implemented in Smalltalk, accesses the call stack by means of the `thisContext` pseudo variable.

2.3.2 Realization in Steamloom

The residues employed by Steamloom for the stack walking approach are, as seen above with the counter approach, direct calls 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 representing 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[3], 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 accesses VM-internal stack frames to extract the signature of the method executed in each frame. The method signatures 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.

As this algorithm operates directly on the call stack maintained by the virtual machine, no additional memory has to be allocated while traversing the stack. This reduces the overhead compared to the standard Java solution, which has to construct its own representation of the call stack first by creating an instance of `Throwable`.

3. Control Flow Guards

This section presents a novel approach to implementing control flow pointcuts in a virtual machine. Developing our solution inside a VM has the advantage that dynamic optimization technology used in today's just-in-time (JIT) compilers can be adapted and applied to dynamic aspect constructs.

Our technique for optimizing `cflow` is based on the concept of *guards* that protect the execution of code via lightweight tests. Our guarding approach is similar to that of *thin guards* [4], which uses lightweight guards to reduce the performance penalty of dynamic class loading in Java.

This section begins with background information on thin guards. Sec. 3.2 presents a high-level overview of our approach, and Sec. 3.3 describes our implementation in the Jikes RVM.

3.1 Background: Thin Guards

Thin guards [4] are a virtual machine optimization techique to enable efficient *speculative optimizations*, i. e., optimizations that rely on certain conditions being true. The primary application of thin guards was to reduce the performance penalty of dynamic class loading in Java. The VM would speculate that dynamic class loading will not occur, and optimize accordingly. All speculative optimizations are guarded by a lightweight check to ensure that correct execution will occur if the assumptions change in the future.

The application of thin guards involved three primary steps.

1. Identify the *optimistic assumptions*. Optimistic assumptions are facts about the currently executing program that, when true, enable improved optimization.
2. Map the optimistic assumptions' *condition bits*[4], which are used to record whether the assumption is currently true.
3. Insert *guards*, lightweight tests that check a condition bit, into the compiled code. Any region of code with speculative optimization applied is prepended with a guard; if the guard is false, an unoptimized (but correct) region of code is executed.

[3] E. g., method `m()` only constitutes a control flow if it is executed in the control flow of `o()`.

[4] Although referred to throughout the paper as a *condition bit*, the notion of a bit is abstract. The implementation of the condition could be any representation that is convenient and efficient.

```
class Fib {
  public int test() {
    return fib(5);
  }
  public int fib(int n) {
    if (n <= 1) return 1;
    return fib(n-1)+fib(n-2);
  }
}
aspect Aspect {
  before() : call(int Fib.fib(int)) &&
    cflow(execution(int Fib.fib(int))) {
    // ...
  }
}
```

Listing 2. Pointcut matching recursive calls to `fib()`.

In the context of dynamic class loading, the optimistic assumption is, "no class loading has occured since the given method has been optimized." If class loading does occur, the condition bit is set to false to ensure that all speculative optimizations are disabled.

The performance advantage of thin guards relies on two key observations. On the one hand, the overhead of a guard (checking a bit) is cheap, and on the other, optimization can be performed within a compiled method to remove redundant guards, thus creating large regions of highly optimized, guard-free code.

3.2 Introduction to Control Flow Guards

Our approach to optimizing conditional control flow is similar to that of thin guards. Advice depending on `cflow` pointcuts must execute only when certain conditions are met, i. e., when executed in the calling context of a control flow constituent.

Condition bits are used to monitor whether a thread is currently executing in the context of a `cflow` constituent. It is optimistically assumed that `cflow`-dependent advice do not need to be executed, and such advice are protected by guards to verify that assumption. When the thread *is* executing in a context such that the advice need to be executed, the guard will ensure they are executed.

In our approach, the VM maintains a guard bit for every relevant control flow, and this bit is updated on entry/exit to that control flow. Just like one distinct counter is used per control flow selected by a pointcut in the counter based approaches (see Sec. 2.2), we use one distinct bit per control flow in the guards approach. As a result, there are as many control flow guard bits per thread as there are different pointcuts using the `cflow` construct in the application. At shadows dependent on control flow i, advice execution is guarded by testing whether bit i is set.

When the control flow is left, the bit cannot simply be reset: this would yield incorrect behavior for recursive control flows. To cope with recursive control flows, each method activation record stores its own copy of the word containing the control flow guard bits: before setting the `cflow` bit at constituent shadows, the current value is stored into local storage of the current method activation. When leaving the control flow, the bit's value is restored from local storage. This ensures correct behavior in the presence of recursive control flows.

For illustration, consider the code in Lst. 2, which recursively computes the n^{th} Fibonacci number, and includes a `cflow` aspect that advises recursive calls to the `fib()` method, save the initial call from inside the `test()` method. Lst. 3 shows the code for the two methods `test()` and `fib()` where the control flow-dependent pointcut has been mapped to a bit that is stored as the 0^{th} bit in a guard word. The execution of the `fib()` method constitutes the control flow in question. Hence, at the beginning of this method,

```
int test() {
  int result;
  if((thread.cflowState & 1) != 0)
    advice();
  result = fib(5);
  return result;
}

int fib(int n) {
  int oldState = thread.cflowState;
  thread.cflowState |= 1;
  int result;
  if (n <= 1) {
    result = 1;
  } else {
    result = 0;
    if ((thread.cflowState & 1) != 0)
      advice();
    result += fib(n-1);
    if ((thread.cflowState & 1) != 0)
      advice();
    result += fib(n-2);
  }
  thread.cflowState = oldState;
  return result;
}
```

Listing 3. Compiled pseudo-code Fibonacci.

the `cflow` state is saved in a local variable and the pointcut's `cflow` bit is set. At dependent shadows both in `test()` and `fib()`, the same bit is tested to see whether advice should be executed. Finally, before returning from the method, the value saved in the local variable is restored into the thread's `cflow` state. As a result, only the outermost execution of `fib()` resets the `cflow` bit and all recursive calls are correctly advised. When `test()` first invokes `fib()`, the `cflow` bit has not yet been set, hence this call does not lead to an execution of the advice.

Please note that the code shown is only pseudo-code: `cflow` guard bits have no representation at bytecode level. Rather, our implementation is integrated directly into the compilers of the virtual machine. This allows for additional optimizations, as we will show in Sec. 3.3.3.

The mapping of control flow-dependent pointcuts to bit positions is done as soon as the pointcut is deployed. Thus, whenever Java bytecode is translated into machine code, the bit position needed at a particular join point shadow is constant and the required bit masks can be computed at compile-time.

3.3 Implementation

We have implemented the approach in Steamloom [17, 9], an extension of the Jikes RVM that contains VM support for efficient aspect execution.

The Jikes RVM has an adaptive optimization system that continuously monitors execution characteristics of the running application. Initially, code is compiled by the *baseline* compiler. Since Jikes does not contain a bytecode interpreter, it is important that the baseline compiler is executed quickly to avoid delays when a method is executed for the first time. To ensure efficiency, the baseline compiler does not perform any optimizations, and the native code it emits very closely emulates the stack-machine model used by Java bytecode.

If certain methods are observed to be executed frequently, based on profile data, they will be recompiled by the *optimizing* compiler. The optimizing compiler applies state-of-the-art optimization techniques to produce efficient code, including profile-directed inlining.

Jikes RVM uses a mixture of preemptive and cooperative multi-threading. A small number of operating system-level threads can execute a large number of Java threads. Jikes RVM's compilers generate machine code containing *yield points*, which transfer control to another Java thread if the current thread's time slot has elapsed.

Every operating system-level thread is represented by an instance of the `VM_Processor` class[5], which holds a reference to the object representing the thread that is currently executed on "this" processor. A reference to the current processor object is always held in a special register, the *processor register*.

The remainder of this section describes our storage of the `cflow` condition bits, as well as our changes to Jikes RVM's baseline and optimizing compilers.

3.3.1 Condition Bits

One word of storage is allocated per thread to accommodate `cflow` guard bits. This allows for monitoring 32 different control flow pointcuts, which should be enough for most applications. The system can fall back to a more conventional counter-based strategy if more than 32 different control flow pointcuts are used[6].

To store `cflow` state information thread-locally, the virtual machine's multi-threading design is exploited. One word is added to every thread object, and this word is used to store the thread's `cflow` state. To improve the performance of accessing this field, it is added to the processor object as well. Upon every thread switch, the value in the processor object is synchronized with the value in the thread object.

Since the address of the processor object is always held in a register and the position of the `cflow` state field in the processor object is a constant offset known at compile-time, the `cflow` state field can be accessed with as little overhead as a single memory load or write operation.

3.3.2 Guards in the Baseline Compiler

For the baseline compiler, the operations described in 3.2 are implemented in a straightforward way. For accessing the `cflow` state word, a memory load or write operation is used. To test or modify a single bit, standard bit operations (like bitwise *and* or *or*) are used.

At constituent shadows, an additional word is used in the method's stack frame. Before setting the `cflow` bit, the old value is copied into this guard word. When the method is left, the guard word is copied back into the processor object's field to restore the `cflow` state.

Machine code (for the x86 architecture) generated by the baseline compiler for all control flow state related operations is shown in Lst. 4. The `esi` register holds a reference to the processor object, which contains the `cflow` state field at the offset given by `field_offset`. A reference to the current method's stack frame is stored in the `esp` register by the baseline compiler. Jikes' stack frame layout for baseline-compiled methods can be seen in Fig. 1. The old control flow state is stored in a special slot of the stack frame, which is located at offset `stack_offset`.

Note that the code is fairly inefficient, particularly the part that tests whether control flow 0 is active. First, the index of the control flow to test (0) is pushed on the stack and then popped into a register to produce a bit mask that can be `AND`-ed with the processor field.

[5] Operating system-level threads are internally called *virtual processors* in Jikes.

[6] Guards can be used for the 32 most-used control flows, and the fallback strategy for additional pointcuts. Furthermore, the number of bits used as control flow guards could easily be increased if applications using a large number of control flow pointcuts become common; of course, at the cost of additional space per thread.

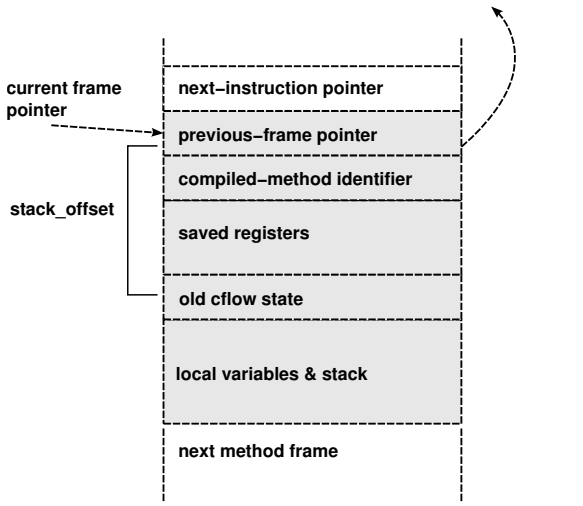

Figure 1. Stack frame layout for baseline-compiled methods.

```
1  ;; save cflow state in stack frame
2  MOV        ecx  [esi]<field_offset>
3  MOV       [esp]<stack_offset> ecx
4  ;; enter control flow 0
5  PUSH         0
6  POP        ecx
7  MOV        eax                1
8  SHL        eax              ecx
9  OR        [esi]<field_offset> eax
10 ;; test whether control flow 0 is active
11 PUSH         0
12 POP        ecx
13 MOV        eax                1
14 SHL        eax              ecx
15 MOV        ecx  <field_offset>[esi]
16 AND        ecx              eax
17 PUSH       ecx
18 ;; restore cflow state from stack frame
19 MOV        ecx  [esp]<stack_offset>
20 MOV       [esi]<stack_offset> ecx
```

Listing 4. Machine code generated by the baseline compiler.

As the control flow index is already known at compile-time, these instructions could be replaced by a single `AND` instruction. The code is generated like this because of the way the weaver passes the control flow index to the compiler (by pushing it on the Java stack) and the stack-machine based model used by the baseline compiler. We did not bother to optimize this further, as the baseline compiler already produces fairly inefficient code.

3.3.3 Guards in the Optimizing Compiler

The implementation of the approach with the optimizing compiler is different. Internally, Jikes' optimizing compiler proceeds in several phases that operate on decreasing levels of abstraction. In the first phase, bytecode is translated into a high-level intermediate representation; the last phase produces optimized machine code for the target architecture. Support for `cflow` guards is implemented using the high-level intermediate representation generated in the first phase.

At this stage, an unlimited number of virtual registers is available. In later phases, these registers are mapped to the (limited) set of physical registers provided by the target architecture. If the number of physical registers is not sufficient to hold all virtual registers

used in the previous phase, they will automatically be stored to and loaded from the method's stack frame. During method calls, registers are stored in the stack frame as well, so that they can be used in the called method's native code.

We have modified the optimizing compiler to load control flow state information into a virtual register, called the *cflow register*, at the beginning of a method. When a control flow is entered at a constituent shadow, three steps are performed: (a) the current value of the `cflow` state is stored in a separate virtual register called *backup register*, (b) the control flow state in the `cflow` register is modified, and (c) the processor object field is updated with the new value of the `cflow` register. On leaving the constituent join point, the virtual register and the processor object field are restored from the backup register. When control flow state has to be tested at dependent shadows, it can be accessed directly from the virtual register it has been stored into at method entry.

If the compiler decides to inline a method into another method, the inlined method's high-level intermediate representation is generated independently and then inserted into the outer method. This implies that it uses separate `cflow` and backup registers for storing control flow state. This is indeed necessary for correct behavior, e. g., if the inlined method constitutes the control flow. In this case, the `cflow` and backup registers of the outer and inlined methods hold different values.

At first sight, this compilation strategy does not seem to differ significantly from the one described for the baseline compiler. It might even look less efficient, because control flow state is read at the beginning of every method, although it is probably required only in a small fraction of the methods executed. However, since the approach operates on the high-level representation, the optimizing compiler will apply all its standard optimization techniques in later phases. Some of the applying optimizations are listed below:

- If the virtual register holding control flow state is never read, the compiler will detect this and eliminate the memory load operation that initialized the register. Thus, methods that do not access control flow state do not exhibit any overhead.
- If control flow state information is frequently required (e. g., when a dependent shadow appears in a tight loop), it has to be loaded only once from main memory and can be kept in a physical register. Basically, by using a virtual register for storing control flow state, the decision on whether to keep the value in a physical register or in the stack frame is left to the optimizing compiler's advanced algorithms.
- The same applies to the old control flow state value, which is saved at constituent shadows. Again, the compiler can decide whether to keep it in a physical register or to store it in the method's stack frame, based on how many registers are needed by the method.

The intermediate representation generated by the optimizing compiler is shown in Lst. 5. Here, `PR` denotes the processor register. The old `cflow` state is loaded from the processor object and stored into the backup register named `t2psi`. When entering the control flow, bit 0 is set by an `OR` operation and the new value is stored in register `t28si`, as well as in the processor object. At dependent shadows, the control flow state is accessed directly from a virtual register. Note how the bit mask is included as a constant and does not have to be computed via a shift instruction anymore. Finally, when the control flow is left, the processor object is updated from the value stored in the backup register.

In addition to the optimizations performed by the compiler, our implementation employs two custom optimizations:

- If a method is inlined, the virtual register holding the control flow state is not initialized by loading it from the processor object (which would result in a memory load). Instead, the value is copied from the outer method's virtual register. In addition to saving a memory load operation, this optimization allows the compiler to eliminate some virtual registers if one of the methods does not modify control flow state. In this case, subsequent phases of the compiler can infer that both virtual registers hold the same value and thus map them to the same physical register.
- The test checking `cflow` state at dependent shadows is removed if it is guaranteed to always succeed. This is determined by checking whether the control flow is entered unconditionally at the beginning of the method containing the test. If the code is generated to be inlined into another method, the outer method is checked for an unconditional control flow entry, too—as well as *its* outer method if several levels of inlining are performed.

  For the Fibonacci example from Lst. 3, this optimization would remove the guards for advice executions inside the `fib()` method, since they will always succeed.

  This optimization can be seen as a somewhat weaker form of `abc`'s interprocedural analysis. It does not perform a whole-program analysis, which is not feasible in a virtual machine due to time and memory requirements. Instead, it is restricted to the set of methods inlined into the method currently being compiled. This set will always be reasonably small, as the compiler avoids creating large method bodies.

```
1  ;; save cflow state in virtual register
2  int_load  t2psi(I) = PR, <field_offset>
3  ;; enter control flow 0
4  int_or    t28si(I) = t2psi(I), 1
5  int_store t28si(I), PR, <field_offset>
6  ;; test whether control flow 0 is active
7  int_and   t7si(I) = t28si(I), 1
8  int_ifcmp t7si(I), 0, ==, SKIP_ADVICE
9  ;; restore cflow state
10 int_store t2psi(I), PR, <field_offset>
```

Listing 5. Intermediate Representation generated by the optimizing compiler.

Both optimizations are particularly effective in the presence of inlining. This complements nicely with the fact that Jikes performs profile-directed inlining: inlining is focused on the most frequently used methods (and trivial methods, like setters and getters). Thus, the above optimizations apply to "hot" parts of the application, where they matter most. Less frequently executed methods will be compiled only by the baseline compiler, which quickly generates less efficient code.

4. Evaluation

This section evaluates the `cflow` implementations presented in Secs. 2 and 3. Three kinds of measurements were performed:

- A micro-measurement benchmark measures and compares the overhead of two parts of the `cflow` implementations: (a) the overhead for constituting a control flow, and (b) the overhead for checking whether a dependent join point actually occurs within a control flow.
- A modified version of the SPECjvm98 benchmarks measures and compares the impact of the `cflow` infrastructures on different approaches in real programs.
- Benchmarks collected by the `abc` group to measure the performance of aspect-oriented programs.

Results of these benchmarks will be presented in Secs. 4.2, 4.3, and 4.4, after a short introduction to the overall setting for the evaluation in Sec. 4.1.

4.1 Evaluation Setting

When a VM begins executing an application, there is a number of sources of overhead, such as class loading, verification, and dynamic compilation. Once these activities have subsided, the VM is often referred to be executing in *steady state*.

To ensure that different start-up behaviors of the presented environments do not influence the measurements, the presented results are steady-state results. For each benchmark, 30 iterations were executed and the median of the last 10 runs' results constituted the benchmark's performance. In all cases, the last 10 runs were clearly executed at steady state.

All measurements were made on a Dual Xeon workstation (2x3 GHz) with 2 GB RAM running Linux 2.4.23.

HotSpot 1.5.0 (Sun's standard JVM) was used to run the benchmarks compiled with `ajc` 1.5.0 and `abc` 1.1.0, as well as to execute the benchmarks in the JAsCo 0.8.7 environment. For JAsCo, the recommended *HotSwap 2* implementation was used, with the `inlinecompiler` switch enabled for improved performance. For AspectWerkz 2.0, the benchmarks were executed on JRockit 1.4.2_08; Steamloom is an extension to Jikes RVM 2.3.1.

The programs compiled with `ajc` or `abc` compilers were not run on the Jikes RVM, which would have given a direct comparison to our implementations in Steamloom. This is because both AspectJ compilers produce code which exploits special optimizations of production Java virtual machines; such code is bound to execute untypically slow on other VMs like Jikes RVM. AspectWerkz and JAsCo even rely on features which are not provided by Jikes RVM.

Nevertheless, to produce comparable results, our measurements were conducted relative to reference performances: as such, the performances of running the benchmarks without deployed aspects on a Java virtual machine without support for AOP were considered. For `ajc`, `abc`, and JAsCo, the reference system was HotSpot, for AspectWerkz it was JRockit, and for Steamloom it was the Jikes RVM, each in the previously mentioned versions.

Benchmark results will be presented as overheads of the benchmarks with deployed aspects as compared to the performances of the corresponding reference JVMs. Additionally, the absolute performance numbers for each benchmark and each approach are provided. However, when comparing the absolute numbers, one should keep in mind that underlying virtual machines exhibit significant differences in their respective performance characteristics.

4.2 Simple Micro-Measurements

The `cflow` infrastructure consists of two parts: on the one hand, bookkeeping which control flows are currently active and, on the other, checking whether a join point occurs in a specific control flow. The performances of these two parts are measured independently of each other. As we will explain, the benchmark was constructed such that only the `cflow` infrastructure is executed, but no advice.

4.2.1 Benchmark Setup

The micro-benchmarks, a small program presented in AspectJ syntax in Lst. 6 was used. For the other environments, the example was implemented in their respective syntax, or by using appropriate API calls. The example makes use of method execution pointcut designators because they are supported by all AOP implementations in focus. The actual implementation of the benchmark harness is not shown in the listing and would have to be inserted at line 16.

For a single run of the benchmark, the total time used to perform 100,000 iterations of each of the `constituent()` and `dependent()` operations was measured:

- To measure the overhead of the control flow constitution infrastructure the method `constituent()` (line 9 of Lst. 6) was called. This method constitutes the control flow and immediately returns. Hence, by executing it, only the bookkeeping infrastructure is executed.
- To measure the overhead of the infrastructure for checking whether a join point occurs in a specific control flow, the method `dependent()` (line 5) was called. A pointcut (lines 24 to 25) conditionally binds an advice to the execution of this method; the pointcut only matches when `dependent()` is executed in the control flow of `constituent()`. This is not the case in our benchmark harness, which is implemented completely in the method `main()`; hence, the advice will never be executed during our measurement.

Under certain circumstances, `abc` can determine that, e. g., a certain control flow will always or never be active when a given join point shadow is executed; in such cases, the infrastructure for constituting or checking a control flow can be omitted. The goal of the micro-benchmark is to measure the impact of this checking mechansim, thus the example program was written such as to prevent the optimizations mentioned above; their effect on efficiency will be measured by the macro-benchmarks.

If it was not possible to call `dependent()` from within the control flow of `constituent()`, the `abc` compiler would realize that the pointcut has no join points and would not weave any infrastructure. Similarly, if `dependent()` was not called from outside the control flow of `constituent()`, the `abc` compiler would realize that the check would never fail at run-time and weave an unconditional call to the advice in `dependent()`. To prevent these optimization, a call to `dependent()` was inserted outside the control flow in line 15.

The executing virtual machine can also apply optimizations. For example, when it can determine that a called method is empty, it can optimize the call away. To prevent this from happening, the method `constituent()` and the advice increment a counter (lines 7 and 27).

For measuring the overhead of each implementation relative to the version of the benchmark without the aspect, the class `Base` was compiled with the standard `javac` compiler and executed on the reference system as specified in this section's introduction.

4.2.2 Results

Results for the micro-measurements are shown in Tab. 1. The columns with the title **constitution** show the performance for the `cflow` infrastructure inserted at constituent join point shadows; the columns with the title **check** show the performance for the `cflow` infrastructure at dependent shadows. In each case, the first sub-column shows the absolute number of milliseconds needed for 100,000 runs. The second sub-column shows the factor by which execution time was increased in the presence of the aspect as compared to the reference system, i.e., lower numbers indicate better performance.

The results for each approach are presented in one row. For `abc`, the benchmark was executed with different settings. First, the optimizations of levels one and three were used, denoted by `O1` and `O3` in the table. Second, a single- and a multi-threaded environment were simulated, denoted by `st` and `mt`, respectively. As explained in Sec. 2.2, `abc` applies a special optimization for single-threaded applications. Single- and multi-threaded environments were simulated by measuring the performance in the first and in the second

```
class Base {
  static int counter;
  static boolean callDependent = false;

  void dependent()
  {
    counter++;
  }
  void constituent()
  {
    if (callDependent)  dependent();
  }
  public static void main(String[] args)
  {
    new Base().dependent();
    /*execute benchmark*/
  }
}

aspect Aspect {
  static int counter;

  before() :
    execution(void Base.dependent()) &&
    cflow(execution(void Base.constituent()))
  {
    counter++;
  }
}
```

Listing 6. Code for the first micro-measurement.

	constitution		check	
	ms	overhead	ms	overhead
HotSpot (ref)	262	1.000	197	1.000
AspectJ	10321	39.393	2,915	14.797
abc-O1-st	1,586	6.053	1,550	7.868
abc-O3-st	1,566	5.977	1,583	8.036
abc-O1-mt	5,142	19.557	4,350	22.081
abc-O3-mt	5,196	19.832	4,436	22.518
JAsCo	613	2.340	$1.1 \cdot 10^7$	56,751.269
JRockit (ref)	198	1.000	204	1.000
AspectWerkz	14783	74.700	8859	43.400
Jikes (ref)	197	1.000	197	1.000
SL-Stackw.	328	**1.665**	67967	345.010
SL-Counter	1049	5.325	721	3.660
SL-Guards	524	2.660	328	**1.665**

Table 1. Results of the micromeasurements

thread. It was ensured that only one thread was actually executing and the other one was sleeping during the measurement.

The results of the Steamloom implementations are shown in the lines **SL-Stackw.** for the stack walking-based implementation, **SL-Counter** for the counter-based implementation, and **SL-Guards** for the guards-based implementation. Rows that contain the performance of the reference system are denoted by the suffix **(ref)**.

The following observations follow from interpreting the numbers in the table:

- The stack walking approach, used by JAsCo and Steamloom Stackwalking, does not produce overhead at points constituting a control flow. The small overhead observed is due to the support of these approaches for run-time deployment of aspects. This support also contributes to the observed overhead in JAsCo, AspectWerkz, and all Steamloom implementations. Conversely, the check at dependent join points is extremely expensive. Integrating stackwalking directly into the virtual machine is several orders of magnitude faster than JAsCo's approach of accessing the call stack. This was to be expected, since Steamloom's VM integration allows for a direct access to the virtual machine's call stack. However, even Steamloom's stackwalking implementation is significantly slower than the counter-based approaches at dependent shadows.
- The `abc` compiler produces less overhead than `ajc` in the single-threaded case. In the multi-threaded case, the code produced by both compilers performs badly. Since we implemented the benchmark in a way that `abc` cannot apply optimizations, the results for both optimization levels are about the same.
- The numbers clearly show that Steamloom's counter (SL-Counter) and guards (SL-guards) implementations both outperform the `abc` compiler even in the single-threaded case, while already ensuring thread-safety.
- Last but not least, the numbers show that constituting a control flow *and* performing the corresponding check at dependent join points with the guards-based approach is even absolutely faster than respective operations with both the `ajc` and `abc` compilers, although the latter execute on a production JVM.

4.3 Macro-Measurements

The benchmark presented in the previous section measures only the overhead introduced by infrastructure needed by `cflow` implementations. Although the results revealed big differences between the implementations, the overhead may be less significant in large-scale applications where it is possible to optimize residues away. To compare our approach against `abc` in a more realistic environment that enables `abc`'s interprocedural analysis as well as Steamloom's optimizations, we present a more complex benchmark in this section.

4.3.1 Benchmark Setup

The SPECjvm98 benchmarks were modified by adding 15 pointcut-and-advice pairs to each benchmark. The pointcuts all have the following form:

```
execution(pc1) && cflow(execution(pc2))
```

They have been picked to cover a wide range of different characteristics. Properties of the pointcuts vary with respect to

- the rate of control flow constitutions,
- the ratio of dependent join point shadow executions inside to outside of the control flow,
- and the ratio of dependent join point shadow occurrences directly in vs. outside the constituent method.

The advice attached to each pointcut only increments a counter, so that the overhead introduced by additional functionality is minimal.

The benchmarks were compiled including the corresponding aspect with `ajc` and with `abc` at optimization levels `O1` and `O3`. The resulting code was executed on HotSpot. `abc`'s optimization level `O3` includes interprocedural optimizations and increases `abc`'s compilation time to well over ten minutes. Compilation with `ajc` or `abc` at optimization level `O1` only took a few seconds at most.

For AspectWerkz, an aspect definition file was provided and passed to the execution environment when starting the benchmark. For Steamloom, the benchmark harness was extended to deploy the aspect before starting the iterations.

It was verified that the advice are executed correctly in the various environments by counting the advice executions and comparing them to each other.

We do not present results for JAsCo in this section, since its weak performance in the presence of `cflow` pointcuts prohibits the execution of the macro-benchmark. The "mpegaudio" benchmark is not included because it was only available as obfuscated class files that could not be processed by most AOP implementations. `abc` could not successfully compile the "javac" benchmark; hence, this benchmark is also omitted[7].

4.3.2 Results

The results of running this benchmark are presented in Tab. 2. Each row shows the result of one implementation. For each benchmark, two numbers are shown: the absolute running time in milliseconds (ms) and the overhead compared to the respective reference implementation (ovhd), i. e., the reference virtual machine without support for aspect-oriented features as specified in section 4.1 executing the benchmark without any aspects deployed.

Although the primary goal is to measure the impact of the `cflow` infrastructure, the presented numbers also include some additional overhead. For each approach, not only the additional time for executing the infrastructure, but also the time needed to execute the advice added by the aspect is included in the presented measurement times. For AspectWerkz and Steamloom, the overhead of the dynamic deployment facility is also included in the presented measurement times.

From the numbers presented in Tab. 2, we draw the following conclusions.

- Our novel implementation based on control flow guards exhibits the least overhead for all benchmarks, even if `abc`'s interprocedural analysis is used. Our counter-based implementation is usually at the same level with `abc`-O1. The only exception to both rules is the "jack" benchmark.
- For the `mtrt` benchmark, our Steamloom-Guards implementation is considerably faster than `abc` and all other approaches. We would also like to draw the reader's attention to the fact that, with the exception of "mtrt", all benchmarks are single-threaded; hence, `abc` can benefit from its optimization for this case. But, our implementation would exhibit the same performance in multi-threaded environments, i. e., in contrast to numbers for `abc` compiled single-threaded benchmarks, the numbers for our approach characterize the most general, thread-safe implementation.
- `ajc` and AspectWerkz provide `cflow` at a very unsatisfactory performance. While `ajc` is faster than AspectWerkz, it still inhibits a significant overhead (e. g., 14.7 % for `jess` or 31.4 % for `mtrt`).
- An implementation based on stack inspection is not beneficial. This becomes especially visible in the `jess` benchmark. It includes a recursive interpreter for a logic programming language that yields particularly deep call stacks. Accordingly, the overhead for Steamloom's stack walking implementation is very high for this benchmark.
- `abc` exhibits a considerably less overhead than `ajc` already at the lower optimization level. The impact of the interprocedural optimization introduced in `O3` is large for the `compress` benchmark, which `abc` can analyze very effectively due to its small size. For all other benchmarks, the higher optimization level performs only slightly faster than `O1`.

[7] We are in contact with `abc`'s implementors; but they could not fix the bug prior to submission deadline of this paper.

```
1  pointcut move():
2    call(void FigureElement+.moveBy(...)) ||
3    call(void Point.setX(int)) ||
4    call(void Point.setY(int)) ||
5    call(void Line.setP1(Point)) ||
6    call(void Line.setP2(Point));
7
8  after() returning:
9      move() && !cflowbelow(move()) {
10   Display.needsRepaint();
11 }
```

Listing 7. Aspect and advice for the figure benchmark.

- The overhead of `abc` is very close to that of `ajc` for the `mtrt` benchmark, while it is significantly below that of `ajc` for all other benchmarks. This benchmark is the only multi-threaded benchmark of the SpecJVM98 suite. Thus, `abc`'s optimization for single-threaded applications, explained in Sec. 2.2, does not apply here.

4.4 Benchmarks from `abc` Group

While the benchmarks presented in Sec. 4.3 used real applications from the SPECjvm98 benchmark suite it was synthetic in the sense that it used `cflow` pointcuts introduced only for the purpose of the benchmark. These pointcuts did not add any useful functionality to the application.

The abc team has gathered various benchmarks by collecting AspectJ programs from public sources on the web [13, 8] some of which also use cflow pointcuts. We were able to run the `figure` and the `quicksort` benchmarks. However, both benchmarks are fairly short, each benchmark consiting of approx. 150 lines of code. Unfortunately, we were not able to run the more complex benchmarks `Law of Demeter`, `Cona`, and `ants` because they are using more advanced constructs not currently supported by Steamloom[8].

4.4.1 Benchmark Setup

The measurements in this section were performed against a reference version without deployed aspects. We only performed measurements for `ajc`, `abc` and Steamloom's control flow guards implementation. The previous benchmarks have already shown that the other implementations yield a considerably worse performance so that an additional measurement would not add any value to the results of this paper.

The `cflow`-dependent pointcuts used in both benchmarks are very similar, having the form `pc && !cflowbelow(pc)` to capture non-recursive entry-points into certain parts of the program. For the `figure` benchmark—which simulates a simple figure editor—the pointcut shown in Lst. 7 is used. It causes a notification to the display object whenever a figure element is changed. However, calling a point's `moveBy` method during the execution of a line's `moveBy` method does not result in a duplicate notification because of the employment of the `!cflowbelow(move())` pointcut.

The `quicksort` benchmark collects various statistics on the sorting algorithm. It uses a similar cflow pointcut to select the top-level call to the recursive `quicksort` method to initialize and display the statistics (Lst. 8).

4.4.2 Results

The results for both benchmarks are presented in Tab. 3. Again, our novel implementation using control flow guards performs very

[8] Please note, that these unsupported constructs are independent from the implementation of the `cflow` pointcut.

	compress		jess		db		javac		mtrt		jack	
	ms	ovhd	ms	ovhd	ms	ovhd	ms	ovhd	ms	ovhd	ms	ovhd
HotSpot (ref)	5266	1.000	2045	1.000	13760	1.000	4131	1.000	1865	1.000	2679	1.000
AspectJ	35548	6.750	2346	1.147	13863	1.007	4289	1.038	2450	1.314	2698	1.007
abc-O1	14047	2.667	2079	1.017	13741	0.999	N/A	N/A	2384	1.278	2691	**1.004**
abc-O3	6852	1.301	2075	1.015	13754	**1.000**	N/A	N/A	2289	1.227	2702	1.009
JRockit (ref)	4345	1.000	1315	1.000	8812	1.000	3084	1.000	1748	1.000	2006	1.000
AspectWerkz	83340	19.181	2153	1.637	8868	1.006	3418	1.108	3007	1.720	2061	1.027
Jikes (ref)	4769	1.000	1790	1.000	10442	1.000	5900	1.000	2944	1.000	2982	1.000
SL-SW	N/A	N/A	28970	16.184	10444	**1.000**	12651	2.144	15520	5.272	3081	1.033
SL-Ctr	10623	2.228	1884	1.053	10451	1.001	5993	1.016	3366	1.143	3077	1.032
SL-Guards	6039	**1.266**	1779	**0.994**	10445	**1.000**	5939	**1.007**	3287	**1.117**	3044	1.021

Table 2. Results from running the spec benchmarks.

```
1  pointcut sort():
2    call(void QuickSort.quicksort(...));
3  pointcut entry():
4    sort() && !cflowbelow(sort());
5  before() : entry() {
6      Stats.before_entry();
7  }
8
9  after() returning: entry() {
10     Stats.after_entry();
11 }
```

Listing 8. Aspect and advice for the quicksort benchmark.

	figure		quicksort	
	ms	ovhd	ms	ovhd
HotSpot (ref)	10	1.000	9062	1.000
AspectJ	672	67.200	10303	1.137
abc-O1	102	10.200	9854	1.087
abc-O3	10	**1.000**	9984	1.102
abc-exec-O3	43	4.300	—	—
Jikes (ref)	25	1.000	7895	1.000
SL-Guards	43	1.720	8291	**1.038**

Table 3. Results from running the figure and quicksort benchmarks.

well. In fact, it exhibits the least overhead except for the figure example when `abc` is run with the highest optimization level. A closer inspection shows that `abc` is able to completely optimize away all control flow-related residues in this benchmark. Since the overhead for notifying the display is negligible, `abc` does not show any overhead compared to the reference measurement in this case.

We argue that this was possible mainly due to the small size of the benchmark and the particular pointcut. As we have shown in Sec. 4.3, this does not happen as frequently, when applying a larger variety of pointcuts on bigger programs.

Although the figure benchmark's essentially being a micro-benchmark, an interesting difference between the optimizations performed by `abc` and Steamloom can be seen by modifying the pointcut only slightly. When all `call(...)` expressions are changed to `execution(...)` expressions, the semantics of the aspect does not change: the display is still notified everytime a figure element is modified and duplicate notifications are avoided.

However, this slight modification already prevents some of `abc`'s intra-procedural optimizations, as can be seen in the row `abc`-exec-O3, displaying a highly increased overhead. The reason for this behaviour is that when using `call` pointcuts, the whole pointcut has more shadows in the program (at every call) compared to the `execution`-based pointcut. With more shadows, `abc` can reason about each shadow separately and—in the case of this benchmark—determine statically whether it occurs inside or outside the control flow. If `execution` is used in the pointcut, however, there are less shadows in the program (only the method bodies) and these shadows may be executed both inside *and* outside the control flow, so that a dynamic check is necessary.

Steamloom's control flow guard optimizations are not vulnerable to such modifications. Jikes' inlining optimization will generate a separate (inlined) version of a method body at every hot call site, so that our interprocedural optimizations can reason about these join points in the same differentiated way as if `call` pointcuts had been used.

This shows an advantage of delaying program optimizations until run-time. They can exploit the VM's dynamic optimizations, e. g., inlining, to obtain additional information, for example about the calling context, that enable more efficient optimizations. On the other hand, offline optimizations allow more time-consuming and memory-consuming optimizations which—in some cases—can completely optimize away the overhead induced by `cflow` pointcuts.

5. Future Work

There are three areas of future work. First, as further discussed in the next subsection, we will provide support for context extraction from the control flow. Next, we will investigate the usefulness of our implementation to facilitate other program language features, like context-oriented programming (see section 5.2). Finally, we will include further optimization techniques to be applied by our `cflow` implementation. Our current implementation does not perform sophisticated redundant guard elimination. A next step in reducing guard overhead would be to perform more advanced path splitting and redundant guard removal analysis, similar to that performed in previous work [4].

5.1 Context Extraction

In Sec. 2.1, we have mentioned that our approach does not support the extraction of context information from constituent join points. Implementing such support is a subject of future work, and we expect that context extraction can be implemented at excellent performance when VM structures are exploited, as with control flow guards.

As mentioned in Sec. 2.2, approaches adopting counters to implement `cflow` matching usually use a stack when context information from constituent join points needs to be made available to advice. Our planned implementation of context extraction does not require a stack, but will instead rely on extracting the required values from the corresponding stack frames directly. We are confident

that this is feasible, given that values from frames further up in the call stack are always stored in a way that allows for efficient access to them.

The required values are, in all cases, local state of the method containing the constituent shadow. Depending on whether the method that has to access them is inlined in the former method or not, they can be accessed as follows:

- In the case of inlining, local state of the calling method is available in the form of operands at high-level intermediate representation, which makes their access obviously feasible.
- In all other cases, local state of the method containing the constituent shadow is stored in its stack frame. Baseline-compiled methods keep their state in stack frame slots anyway, and for optimised methods, values stored in registers during execution are saved to the stack frame in case another method is called. From these locations, the respective values can be retrieved.

This approach requires that, for elements extracted from constituent join point context, a set of references to their storage locations is maintained. In case of recursive control flow entries, the set needs to be updated accordingly; otherwise, it can be passed on in the same way as our approach does for control flow guards.

Adopting this solution will come at some cost, since maintaining context references is an additional task that compiled application code will have to fulfil. Yet, the cost of maintaining a complex data structure – e. g., a stack, as used by current implementations – is most likely much higher.

5.2 Context-Oriented Programming

Another subject of future work is to extend the guards-based `cflow` implementation to programming approaches that allow for control flow-dependent behaviour but do not apply explicit quantification over join points in terms of pointcuts. An example of such an approach is *context-oriented programming* (COP), existing in the form of the Lisp-based ContextL programming language [29]. COP allows for dynamically (de)activating *layers* that augment the running application with additional behaviour. An important difference to the *pointcut-and-advice* flavour of AOP [26] is that in COP, points where crosscutting behaviour applies are not defined in terms of pointcuts, but implicitly in terms of the interfaces of application classes. Hence, COP's join point model is more coarse-grained than that of, e. g., AspectJ, but no less powerful in the respect of possible interaction with the base application.

Given that layers can be dynamically (de)activated, the possibility of applying control flow-dependent behaviour is given in COP: in the control flow of a method that activates a layer, behaviour applies that would not outside the so-defined context. The current implementation of ContextL [30] relies on the dynamic definition of classes representing layer combinations and on multiple dispatch to realise context-dependent behaviour. The implementation utilises caching mechanisms to improve performance. The authors plan to port COP to the Java platform [30]. We believe that the still-remaining overhead due to caching costs could be eliminated in such an implementation by exploiting guards-based control flow matching.

6. Conclusion

The most important *result* of the work presented in this paper is that our novel implementation based on control flow guards displays the best performance among all approaches. It performs even better than `abc` at its highest optimization level, which provides one of the most efficient implementations available today. However, a whole-program analysis as performed by `abc` comes at the cost of (a) 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. Our approach has neither of these disadvantages. If `abc` is used without the whole-program analysis, benchmark results are even more favorable for our approach. The same is true for multi-threaded applications, as our strategy is thread-safe by default and does not need any special optimizations for the single-threaded case.

If there are more control flow-based pointcuts deployed in a system than the number of control flow guards we have reserved[9], we have to use the counter strategy for the additional pointcuts. Even though this strategy is slower than our counter-based implementation, it is still at the same level with `abc`. Thus, we expect all applications that are making use of `cflow` pointcuts to benefit from our approach, even if they are using a large number of pointcuts.

By implementing several approaches to the best of our knowledge within the same environment, we have been able to directly compare their relative performance. Stack walking, although having no overhead at constituent shadows, does not seem to be an alternative since its complexity depends on the stack depth met at dependent shadows. Counter-based strategies, exhibiting constant cost at control flow entries and exits as well as at dependent shadows, perform significantly better in all real-word benchmarks. This performance is topped only by our novel guards-based strategy.

From the results presented in Sec. 4.3, it is obvious that interprocedural optimizations can lead to considerable performance gains. However, due to time and memory constraints, a whole-program analysis is infeasible inside a virtual machine. We believe that, by applying these optimizations on a smaller scale, e. g., to the code that is produced from a method and all methods that are inlined into it, they could become feasible. As described in Sec. 3.3.3, we have already successfully applied some of `abc`'s interprocedural optimizations to remove redundant control flow checks. Since this analysis works across method boundaries only in the presence of inlining, it is guaranteed to execute quickly and is always focused on the hot parts of the application. It seems very promising to include more static analyses for predicting the control flow into Jikes RVM's optimizing compiler.

Acknowledgements

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

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] Matthew Arnold and Barbara G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 498–524, London, UK, 2002. Springer-Verlag.

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

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

[9] Although the number of guard bits cannot be increased at runtime, an arbitratry number of bits can be reserved.

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

[8] 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.

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

[10] 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.

[11] J. Brichau, M. Haupt, N. Leidenfrost, A. Rashid, L. Bergmans, T. Staijen, A. Charfi, C. Bockisch, I. Aracic, V. Gasiunas, K. Ostermann, L. Seinturier, R. Pawlak, M. Südholt, J. Noyé, D. Suvée, M. D'Hondt, P. Ebraert, W. Vanderperren, M. Pinto, L. Fuentes, E. Truyen, A. Moors, M. Bynens, W. Joosen, S. Katz, A. Coyler, H. Hawkins, A. Clement, and O. Spinczyk. Report describing survey of aspect languages and models. Technical Report AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01, Vrije Universiteit Brussel, 17 May 2005 2005.

[12] T. Dinkelaker, M. Haupt, R. Pawlak, L. D. Benavides Navarro, and V. Gasiunas. Inventory of aspect-oriented execution models. Technical Report AOSD-Europe Deliverable D40, AOSD-Europe-TUD-4, Darmstadt University of Technology, 28 February 2006 2006.

[13] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the Dynamic Behaviour of AspectJ Programs. In *Proc. OOPSLA 2004*, 2004.

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

[15] Glassbox-Inspector Home Page. `https://glassbox-inspector.dev.java.net/`.

[16] B. Harbulot and J. R. Gurd. Using aspectj to separate concerns in parallel scientific java code. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 122–131. ACM Press, 2004.

[17] 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.

[18] 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.

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

[20] JBoss AOP Home Page. `http://www.jboss.com/products/aop`.

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

[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] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[25] Karl Lieberherr, David H. Lorenz, and Pengcheng Wu. A case for statically executable advice: checking the law of demeter with aspectj. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 40–49, New York, NY, USA, 2003. ACM Press.

[26] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proc. ECOOP 2003*, 2003.

[27] 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.

[28] P. Avgustinov and others. abc: an Extensible AspectJ Compiler. In *Proc. AOSD'05*, pages 87–98. ACM Press, 2005.

[29] P. Costanza and R. Hirschfeld. Language Constructs for Context-Oriented Programming: an Overview of ContextL. In *Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*. ACM Press, 2005.

[30] P. Costanza and R. Hirschfeld and W. de Meuter. Efficient Layer Activation for Switching Context-Dependent Behavior. In *Joint Modular Languages Conference 2006 (JMLC2006)*. Springer, 2006.

[31] D. Sereni and O. de Moor. Static analysis of aspects. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 30–39. ACM Press, 2003.

[32] 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.