Turbo DiSL: Partial Evaluation for High-level Bytecode Instrumentation

Yudi Zheng[1], Danilo Ansaloni[2], Lukas Marek[3], Andreas Sewe[4], Walter Binder[2], Alex Villazón[5], Petr Tuma[3], Zhengwei Qi[1], and Mira Mezini[4]

[1] Shanghai Scalable Computing Lab, Shanghai Jiao Tong University, China
{zheng.yudi, qizhwei}@sjtu.edu.cn

[2] University of Lugano, Switzerland
{danilo.ansaloni, walter.binder}@usi.ch

[3] Faculty of Mathematics and Physics, Charles University, Czech Republic
{lukas.marek, petr.tuma}@d3s.mff.cuni.cz

[4] Technische Universität Darmstadt, Germany
andreas.sewe@cased.de, mezini@informatik.tu-darmstadt.de

[5] Universidad Privada Boliviana, Bolivia
avillazon@upb.edu

Abstract. Bytecode instrumentation is a key technique for the implementation of dynamic program analysis tools such as profilers and debuggers. Traditionally, bytecode instrumentation has been supported by low-level bytecode engineering libraries that are difficult to use. Recently, the domain-specific aspect language DiSL has been proposed to provide high-level abstractions for the rapid development of efficient bytecode instrumentations. While DiSL supports user-defined expressions that are evaluated at weave-time, the DiSL programming model requires these expressions to be implemented in separate classes, thus increasing code size and impairing code readability and maintenance. In addition, the DiSL weaver may produce a significant amount of dead code, which may impair some optimizations performed by the runtime. In this paper we introduce Turbo, a novel partial evaluator for DiSL, which processes the generated instrumentation code, performs constant propagation, conditional reduction, and pattern-based code simplification, and executes pure methods at weave-time. With Turbo, it is often unnecessary to wrap expressions for evaluation at weave-time in separate classes, thus simplifying the programming model. We present Turbo's partial evaluation algorithm and illustrate its benefits with several case studies. We evaluate the impact of Turbo on weave-time performance and on runtime performance of the instrumented application.

Keywords: Bytecode instrumentation, aspect-oriented programming, domain-specific languages, partial evaluation, Java Virtual Machine

1 Introduction

Dynamic program analysis tools support numerous software engineering tasks, including profiling, debugging, and reverse engineering. Prevailing techniques for

building dynamic analysis tools are based on low-level abstractions that make tool development tedious, error-prone, and expensive. In the context of managed languages, bytecode instrumentation is a widely used implementation technique for dynamic analysis tools. For example, the Java Virtual Machine (JVM) supports bytecode instrumentation through a native code interface and there are many low-level libraries and frameworks for manipulating bytecode (e.g., ASM, BCEL, gnu.bytecode, Javassist, Serp, ShrikeBT, Soot).

The domain-specific language DiSL [8] offers high-level abstractions to enable rapid development of efficient dynamic analysis tools for the JVM. DiSL succeeds in reconciling a high level of abstraction for tool development, expressiveness, and efficiency of the resulting tools. DiSL is an aspect language based on a pointcut/advice mechanism [7]. The benefit of using aspects for dynamic program analysis stems from the convenient model offered by join points (representing specific points in the execution of a program), pointcuts (denoting a set of join points of interest), and advice (code to be executed whenever a join point of interest is reached) [13]. A dynamic analysis aspect is concise and is easier to define, tune and extend, compared to an equivalent implementation based on low-level bytecode instrumentation techniques [8].

DiSL supports the weave-time evaluation of custom conditionals to decide whether join points are woven, as long as these conditionals only depend on static context information of the join point in question. This feature is key to avoiding the repeated evaluation of such conditionals within advice at runtime. Alas, the user needs to factor out the code that is to be evaluated at weave-time from the advice. This complicates the DiSL programming model, as extra classes and methods need to be introduced, and in some cases advice needs to be split up into several pieces for which the weaving order has to be explicitly specified.

From the user's point of view, it would be much more convenient to simply write conditionals within advice code and rely on the DiSL weaver to optimize the code and to move as much computation as possible from runtime to weave-time. However, the DiSL weaver as presented previously [8] does not perform any optimization of the woven advice. It may even generate a significant amount of dead code that may hinder certain optimizations of the runtime.[6]

In this paper we introduce Turbo, a new partial evaluator that is plugged into the DiSL weaver to optimize woven advice. Turbo performs constant propagation and executes pure methods (i.e., methods that are free of side effects and compute the same result when invoked multiple times with the same arguments) at weave-time; it also removes dead code. Thanks to Turbo, the DiSL programmer usually does not need to take care of factoring out expressions to be evaluated at weave-time. Instead, such expressions are simply embedded in advice code. Turbo will detect them and evaluate them at weave-time. In particular, Turbo guarantees that conditionals that only depend on static context information will always be evaluated at weave-time. Furthermore, the DiSL user may annotate methods

[6] For example, the just-in-time compilers of some recent JVMs base inlining decisions on the size of methods; if the weaver inserts a lot of dead code, the woven method may not be eligible for inlining (a very effective compiler optimization) anymore.

that are pure such that Turbo may execute them at weave-time. Turbo is also aware of many pure methods in the standard Java class library (e.g., methods in `java.lang.Integer` or `java.lang.String`).

The original, scientific contributions of this paper are twofold:

1. We introduce Turbo, a partial evaluator for DiSL [8]. While partial evaluation has been explored by others, Turbo is unique in simplifying the programming model for the development of instrumentation-based dynamic analyses.
2. We present four case studies to illustrate the benefits of Turbo. With one of these case studies, we evaluate the performance impact of Turbo both on weave-time and on runtime of the woven application.

Section 2 gives an overview of DiSL. Section 3 introduces Turbo, our new partial evaluator for DiSL. Section 4 illustrates how Turbo simplifies the DiSL programming model in four case studies, before Sect. 5 explores the performance impact of Turbo. Section 6 discusses related work and Sect. 7 concludes.

2 Background: DiSL Overview

Below we give an overview of some language constructs supported by DiSL, limiting the discussion to the features used in this paper. We will show the DiSL language constructs with concrete examples in Sect. 4. We refer the reader to our comprehensive description of the DiSL language [8] for further information.

Join point model. DiSL has an *open join point model* in which any region of bytecodes can be used as a join point. Pointcuts are expressed with *markers* that select bytecode regions. DiSL provides an extensible library of such markers including ones for selecting whole method bodies, basic blocks, single bytecodes, and exception handlers. DiSL relies on *guards* to further restrict the join points selected by a marker. Guards are predicate methods free of side-effects that are executed at weave-time which have access to static context information.

Advice. Advice in DiSL are expressed in the form of code *snippets*. Snippets are void methods that are instantiated by the weaver and that take annotations indicating whether they are to be woven before or after a join point. In contrast to mainstream AOP languages such as AspectJ, DiSL does not support around advice (synthetic local variables [8] mitigate this limitation).

Context information. Snippets and guards have access to complete static context information (i.e., static reflective join point information). To this end, snippets and guards can take an arbitrary number of static context references as arguments. Methods in static context classes return constants: primitive values, strings, or class literals. DiSL provides an extensible library of static context classes. Snippets have also access to complete dynamic context information, including local variables and the operand stack. Dynamic context information is provided

through an interface type; snippets may take an argument of that type to access dynamic context information.

When a snippet is selected to be woven at a join point, it is first instantiated with respect to the context of the join point. The DiSL weaver first replaces invocations of static context methods with the corresponding constants. That is, static context method invocations in the snippet are pseudo-method calls that are substituted with concrete constants. This step in the weaving process introduces constants into the snippet; thus, the opportunity to optimize the code with partial evaluation arises. Dynamic context method invocations in the snippet are also pseudo-method calls that are replaced with bytecode sequences to access local variables respectively to copy operands from the stack. The partial evaluator Turbo described below is invoked after the removal of static context method calls but before the removal of the dynamic context method calls.

3 Turbo: Partial Evaluator for DiSL

Turbo is a new on-the-fly partial evaluator integrated with the most recent version of the DiSL weaver. If enabled, Turbo performs code optimizations during the process of snippet instantiation. As discussed in Sect. 2, the DiSL weaver instantiates snippets by replacing invocations of static context methods with bytecodes that load the corresponding constants. Turbo can perform any computation in snippets that depends only on constants and does not produce any side effects at weave-time, thus avoiding repetitive runtime computations. To reduce the weaving overhead, Turbo is designed to be simple but efficient, aiming at simplifying the DiSL programming model.

Turbo partial evaluation is divided into three major steps that can be iterated until no further optimization is possible: (1) constant propagation, (2) conditional reduction, and (3) pattern-based code simplification. Turbo guarantees that any intermediate result in the process of partial evaluation represents valid bytecode without any change in the semantics with respect to the initial bytecode. Consequently, it is easily possible to adjust the trade-off between the quality of the partially evaluated bytecode and the time spent in optimization. For example, the DiSL programmer may specify an upper limit on the number of iterations performed by Turbo on each snippet, so as to limit the performance impact of Turbo on weave-time.

If Turbo is iterated until no further optimizations are possible, it guarantees that any bytecode within conditionals that are statically known to evaluate to `false` (i.e., that only depend on computation with constants and on invocations of pure methods) will be discarded. Such conditionals may even enclose snippet code that results in bytecode which would fail the JVM's bytecode verification when instantiated; as the unreachable snippet code is discarded by Turbo, this otherwise unverifiable bytecode can neither cause a weave-time error nor a verification error when the instrumented class is linked. In Sect. 4.3 we will show an example where this property of Turbo is useful, as it allows the programmer

to access potentially illegal positions on the operand stack when enclosed by an appropriate static context information check.

Below we explain each step in the partial evaluation algorithm.

Constant propagation. The constant interpreter performs constant propagation on the input snippet. It symbolically executes the bytecodes in the control-flow graph by transforming an input frame (which represents the local variables and the operand stack) into an output frame according to the bytecodes' semantics. For each bytecode, Turbo stores a frame containing the constant status of each local variable and stack operand before executing it. If a bytecode is reachable through multiple execution paths, Turbo merges the input frames. This operation will replace a constant with a dedicated value indicating that the local variable respectively stack operand is not constant or not the same constant for all merged input frames. Our implementation of the constant interpreter is based on the symbolic analyzer provided by ASM, a Java bytecode manipulation framework. This is a sensible implementation choice, since DiSL's weaver is itself ASM-based.

Besides symbolically executing bytecodes, Turbo also executes pure methods at weave-time, thus enabling constant propagation across pure method calls. To this end, the DiSL programmer must annotate such methods with `@Pure` and ensure that the annotated methods indeed have no side effects and that their output does not change for subsequent invocations with the same arguments. Moreover, the methods have to be static with parameters of primitive (resp. wrapper) types or strings. Out-of-the-box, Turbo also supports the removal of calls to pure methods in the Java class library (e.g., string operations).

Figure 1a presents the algorithm of constant propagation implemented by Turbo. It uses the auxiliary operations defined in Fig. 1b.

Conditional reduction. After constant propagation, some branch instructions can be resolved to either `if(true)` or `if(false)`. Turbo discards the branch that is not taken and replaces the branch bytecode with a number of **`pop`** bytecodes corresponding to the number of operands that would be consumed by the branch bytecode. This code transformation ensures that the snippet code remains valid. After all branch bytecodes have been processed, Turbo removes inaccessible basic blocks from the control-flow graph.

Pattern-based code simplification. After each iteration, Turbo eliminates superfluous code matching one of several different patterns, such as jumping to the next instruction. Another code pattern optimized by Turbo is the sequence of **`pop`** bytecodes introduced by conditional reduction. For each **`pop`** bytecode, Turbo finds out the source bytecodes that push the operand. If all those bytecodes are free of side effects, Turbo removes both the **`pop`** bytecode and its source bytecodes; for each bytecode thus removed, Turbo inserts **`pop`** bytecodes corresponding to the number of stack operands that would be consumed by the removed source bytecode.

```
Input     : An instruction list Φ
Output    : ∪_{instr∈Φ} instr.frame
Initially :
  Q := new Queue();
  Q.enqueue(⟨first instruction of Φ, new Frame()⟩);
Iteration:
  while Q ≠ ∅ do
      ⟨instr, input⟩ := Q.dequeue();
      changed := false;
      if instr.frame = null then
          instr.frame := input.clone();
          changed := true;
      else
          for i := 0 to input.size − 1 do
              if input.get(i) ≠ instr.frame.get(i) then
                  instr.frame.set(i, ĉ);
                  changed := true;
          end
      if changed then
          output := input.clone();
          switch instruction pattern of instr do
              case Load_constant: c → dst, c ∈ C
                  output.set(dst, c);
              case Data_transfer: src → dst
                  v := input.get(src);
                  output.set(dst, v);
              case Data_processing: (op)srcs → dst
                  if ∀src ∈ srcs : input.get(src) ∈ C then
                      v := instr.process(∪_{src∈srcs} input.get(src));
                      output.set(dst, v);
                  else
                      output.set(dst, ĉ);
              case Invocation: call f(srcs)→ dst
                  if ∀src ∈ srcs : input.get(src) ∈ C and f is pure then
                      v := instr.invoke(f, ∪_{src∈srcs} input.get(src));
                      output.set(dst, v);
                  else
                      output.set(dst, ĉ);
              otherwise if instr rewrites dst then output.set(dst, ĉ);
          endsw
          ∀next ∈ instr.next : Q.enqueue(⟨next, output⟩);
  end
```

Fig. 1a. Turbo's algorithm for constant propagation. Auxiliary procedures used are shown in Fig. 1b (Notation: C denotes the set of constants (e.g., `0`, `1.0`, `null`), $\hat{c} \notin C$ denotes a dedicated non-constant value.)

class Queue
  `dequeue`(): Return and remove the first tuple from the queue;
  `enqueue`($\langle instruction, frame \rangle$):
    Insert the tuple $\langle instruction, frame \rangle$ at the end of the queue;

class Frame
  `Frame`(): Initially all elements are assigned the non-constant value $\hat{c}$;
  `clone`(): Return a copy of this frame;
  `get`($position$): Return the element at the specified *position* in the frame;
  `set`($position, value$):
    Replace the element at the specified *position* in the frame with *value*;

class Instruction
  `frame`: constant status of each local variable or stack operand before evaluation;
  `next`: union of possible next instructions;
  `invoke`($method, set\langle argument \rangle$): Execute *method*;
  `process`($set\langle operand \rangle$):
    Symbolically execute the instruction according to its semantics;

Fig. 1b. Auxiliary procedures used by Turbo's algorithm for constant propagation

4 Case Studies

Below we discuss four case studies comparing real-world dynamic analyses using plain DiSL with equivalent versions using Turbo DiSL. They illustrate how Turbo simplifies the programming of efficient dynamic analysis tools.

4.1 Case Study 1: Configurable Instrumentation

Dynamic analyses often require some external configuration to bypass part of their behaviors. When analyzing method calls, e.g., one might be interested in the arguments passed or the execution time of the call. But since not all information is always needed, it is desirable to configure the analysis accordingly to avoid unnecessary overhead.

A straightforward implementation of such a configurable analysis is shown in Fig. 2a. The snippet is woven in at the beginning of each method body; its code will be executed upon each method entry. All configurable cases are coded as conditionals within this single snippet. Alas, the configuration is evaluated at runtime each time the snippet is invoked.[7]

Now, if the boolean methods `profileArgs()` and `profileTime()` always return the same constant value, one would prefer to evaluate the conditionals within the snippet once at weave-time instead of evaluating them upon each method call. In plain DiSL (i.e., without Turbo), the programmer may resort to guards [8] to factor out the code to be evaluated at weave-time, as illustrated in Fig. 2b. However, the resulting code is more complicated and verbose, as the

[7] The JVM's just-in-time compiler may be able to remove some of this overhead.

```
public class MethodAnalysis {
  @Before(marker = BodyMarker.class)
  static void onMethodEntry() {
    if (Configuration.profileArgs()) { ... /* profile method arguments */ }
    if (Configuration.profileTime()) { ... /* profile current wall time */ }
  }
}
```

Fig. 2a. Skeleton implementation of a configurable analysis

```
public class MethodAnalysis {
  @Before(marker = BodyMarker.class, order = 1, guard = ArgsGuard.class)
  static void onMethodEntryArgs() { ... /* profile method arguments */ }

  @Before(marker = BodyMarker.class, order = 0, guard = TimeGuard.class)
  static void onMethodEntryTime() { ... /* profile current wall time */ }
}

public class ArgsGuard {
  @GuardMethod
  static boolean evalGuard() { return Configuration.profileArgs(); }
}

public class TimeGuard {
  @GuardMethod
  static boolean evalGuard() { return Configuration.profileTime(); }
}
```

Fig. 2b. Configurable analysis implemented with guards

programmer has to implement two snippets, two guards, and to supply additional information to the snippet annotation to fix the weaving order.

With Turbo, it is not necessary to use guards to reduce runtime overhead; the code can be implemented exactly as shown in Fig. 2a above. If the methods `profileArgs()` and `profileTime()` in class `Configuration` are annotated with `@Pure`, Turbo will evaluate these methods at weave-time and remove any dead code when weaving the snippet. Consequently, the snippet code can stay simple with all the benefits of weave-time evaluation. As another benefit, the reduction in code size achieved by Turbo helps avoid overlong methods that would violate constraints of the JVM.

4.2 Case Study 2: Tracking Monitor Ownership

In the JVM, each object has an associated monitor. As contention for monitor ownership limits application's scalability, a dynamic analysis to track ownership can assist in finding performance bottlenecks in multi-threaded Java programs.

A thread gains ownership of a monitor either explicitly by entering a **`synchronized`** block (i.e., by executing **`monitorenter`** at the bytecode level), or implicitly by entering a **`synchronized`** method. In the latter case, it is the monitor of the receiver object (**`this`**) that is acquired for instance methods and the monitor of the corresponding instance of `java.lang.Class` for class (static) methods. Which object and hence which monitor is meant is statically known.

```
public class MonitorOwnershipAnalysis {
  @Before(marker = BodyMarker.class, guard = SynchronizedClassMethodGuard.class)
  static void acquireMonitorForClass(MethodStaticContext msc, ClassContext cc) {
    Object assocObj = cc.asClass(msc.thisClassName());
    ... /* track owmership */
  }

  @Before(marker = BodyMarker.class, guard = SynchronizedInstanceMethodGuard.class)
  static void acquireMonitorForInstance(DynamicContext dc) {
    Object assocObj = dc.getThis();
    ... /* track ownership */
  }
}

public class SynchronizedClassMethodGuard {
  @GuardMethod
  static boolean isApplicable(MethodStaticContext msc) {
    return msc.isMethodSynchronized() && msc.isMethodStatic();
  }
}

public class SynchronizedInstanceMethodGuard {
  @GuardMethod
  static boolean isApplicable(MethodStaticContext msc) {
    return msc.isMethodSynchronized() && !msc.isMethodStatic();
  }
}
```

Fig. 3a. Analysis to track monitor ownership using guards

```
public class MonitorOwnershipAnalysis {
  @Before(marker = BodyMarker.class)
  static void acquireMonitor(DynamicContext dc, MethodStaticContext msc,
                             ClassContext cc) {
    if (msc.isMethodSynchronized()) {
      Object assocObj =
        msc.isMethodStatic() ? cc.asClass(msc.thisClassName()) : dc.getThis();
      ... /* track ownership */
    }
  }
}
```

Fig. 3b. Analysis to track monitor ownership relying on partial evaluation

With plain DiSL, however, this distinction needs to be expressed through guards, leading to the both duplicated and hard-to-read code shown in Fig. 3a. Said code is not only verbose but also makes it hard to see that the two cases (`acquireMonitorForClass`/`ForInstance`) complement each other. With Turbo's partial evaluation, the above code can be written in a single snippet using straightforward conditionals as shown in Fig. 3b.

4.3 Case Study 3: Field Access Analysis

Figure 4a shows an instrumentation to profile any access to instance fields. The first snippet is woven before each read access (**`getfield`**) while the second snippet

```
public class FieldAccessAnalysis {
  @Before(marker = BytecodeMarker.class, args = "getfield")
  static void onFieldRead(FieldAccStaticContext fasc, MethodStaticContext msc,
                          DynamicContext dc) {
    String methodID = msc.thisMethodFullName();
    String fieldID = fasc.thisFieldID();
    Object ownerObj = dc.getStackValue(0, Object.class);
    ... /* profile field read */
  }

  @Before(marker = BytecodeMarker.class, args = "putfield")
  static void onFieldWrite(FieldAccStaticContext fasc, MethodStaticContext msc,
                           DynamicContext dc) {
    String methodID = msc.thisMethodFullName();
    String fieldID = fasc.thisFieldID();
    Object ownerObj = dc.getStackValue(1, Object.class);
    ... /* profile field write */
  }
}
```

Fig. 4a. Field access analysis with code duplication

```
public class FieldAccessAnalysis {
  @Before(marker = BytecodeMarker.class, args = "getfield,putfield")
  static void onFieldAcc(FieldAccStaticContext fasc, MethodStaticContext msc,
                         DynamicContext dc) {
    String methodID = msc.thisMethodFullName();
    String fieldID = fasc.thisFieldID();
    int stackDistance = (fasc.getOpcode() == Opcodes.GETFIELD) ? 0 : 1;
    Object ownerObj = dc.getStackValue(stackDistance, Object.class);
    ... /* profile field access (read or write) */
  }
}
```

Fig. 4b. Field access analysis relying on partial evaluation

is woven before each write access (`putfield`). In the former case, the owner object resides on top of the operand stack, whereas in the latter case it is the second topmost stack operand.

Without partial evaluation, it is impossible to combine the two snippets by choosing the stack location to access based on the opcode. As the first argument for the pseudo-method `getStackValue` must be a constant, the snippet would include two branches to access the stack position zero respectively one; for each woven join point, one of the branches would constitute dead code. Moreover, that dead branch would possibly access an illegal position on the operand stack, resulting in bytecode that would fail verification[8]. With partial evaluation, such code duplication is unnecessary; a single snippet with a conditional suffices as shown in Fig. 4b. Turbo guarantees that this conditional is evaluated at weave-time (since it only depends on constant data) and that only the proper constant is propagated to the pseudo-method `getStackValue`.

[8] The DiSL weaver may generate warnings if bytecode is generated that would fail load-time verification.

```
public class ExecutionTraceProfiler {
  @Before(marker = BasicBlockMarker.class, order = 1, guard = ClassInitGuard.class)
  static void onClassInit(MethodStaticContext msc) {
    ... /* profile class initialization */
  }

  @Before(marker = BasicBlockMarker.class, order = 0)
  static void onBB(CustomBasicBlockStaticContext cbbsc) {
    String bbID = cbbsc.thisBBID();
    ... /* profile basic block entry */
  }
}

public class CustomBasicBlockStaticContext extends BasicBlockStaticContext {
  public String thisBBID() {
    String methodFullName = staticContextData.getClassNode().name
      + "." + staticContextData.getMethodNode().name;
    return methodFullName + ":" + String.valueOf(getBBindex());
  }
}

public class ClassInitGuard {
  @GuardMethod
  static boolean evalGuard(BasicBlockStaticContext bbsc, MethodStaticContext msc) {
    return (bbsc.getBBindex() == 0) && msc.thisMethodName().equals("<clinit>");
  }
}
```

Fig. 5a. Execution trace profiler using a custom static context class and a guard

```
public class ExecutionTraceProfiler {
  @Before(marker = BasicBlockMarker.class)
  static void onBB(BasicBlockStaticContext bbsc, MethodStaticContext msc) {
    if (bbsc.getBBindex() == 0 && msc.thisMethodName().equals("<clinit>")) {
      ... /* profile class initialization */
    }
    String bbID = msc.thisMethodFullName() + ":" + String.valueOf(bbsc.getBBindex());
    ... /* profile basic block entry */
  }
}
```

Fig. 5b. Execution trace profiler relying on partial evaluation

4.4 Case Study 4: Execution Trace Profiling

Figure 5a shows a profiler that traces each executed basic block of code, identified by a unique string comprising the fully qualified method name (package, class, method, signature) and a basic block ID (an integer value that is unique within the scope of a method body). In addition, the execution of the first basic block in each class initializer (method `<clinit>` at the bytecode level) is specially tracked by the profiler. The DiSL code in Fig. 5a is complicated; it comprises two snippets and requires both a custom static context and a guard. The static context ensures that the special basic block identifiers are built at weave-time, while the guard identifies the first basic block of class initializers. The snippet

order guarantees that the special profiling of the first basic block in a static initializer happens before the normal basic block profiling.

Figure 5b shows a naïve single-snippet implementation with a conditional; the basic block ID is built within the snippet code. While this implementation is sound, it incurs excessive runtime overhead, since the conditional is evaluated and the identifier is built at runtime for each woven join point, i.e., for each basic block in the base program. However, with partial evaluation, the woven bytecode for both versions of the profiler will be the same, as the conditional depends on static information only and the string operations are pure. Hence, Turbo evaluates these parts of the snippet code at weave-time. In the next section, we will explore weave-time and runtime performance of both versions of the profiler.

5 Performance Evaluation

We use the execution trace profiler of the fourth case study for our performance evaluation, because it intercepts the highest number of join points, both statically at weave-time and dynamically at runtime. That is, the impact of partial evaluation on weave-time performance and the impact of code quality on runtime performance is most pronounced in this case study.

The base programs are benchmarks from the DaCapo suite (release 9.12).[9] We exclude tradebeans and tradesoap because of a well-known issue with a hardcoded timeout[10], which prevents their use together with expensive instrumentation. All measurements were conducted on a 3.0 GHz Intel Core 2 Quad Q9650 with 8 GB RAM running Ubuntu GNU/Linux 10.04 64-bit with kernel 2.6.35. We use Oracle's JDK 1.6.0_30 Hotspot Server VM (64-bit) with a 7 GB heap and DiSL pre-release version 0.9 with complete bytecode coverage, i.e., with a completely woven Java class library [8].

We evaluate three versions of the execution trace profiler: (1) the naïve implementation shown in Fig. 5b *without Turbo*, which serves as a baseline for the comparison; (2) the manually optimized implementation shown in Fig. 5a (without Turbo), henceforth called "DiSL optimized"; and (3) the naïve implementation of Fig. 5b *with Turbo*, called "Turbo DiSL". Moreover, we consider three performance metrics: (a) the weave-time, i.e., the time to weave all classes loaded during a single benchmark iteration; (b) the startup time, i.e., the process time from creation to the termination of the first benchmark iteration; and (c) the steady-state execution time, i.e., the median of the execution times of 15 benchmark iterations within the same JVM process.

For each metric, Fig. 6 illustrates the speedup of "DiSL optimized" and "Turbo DiSL" relative to the baseline. The gray marks refer to the individual benchmarks, while the black marks refer to the geometric mean of all speedup factors. When the speedup factor is below 1x, it indicates a slowdown. The diagonal line indicates data points for which the performance of "DiSL optimized" and "Turbo DiSL" is the same.

[9] See `http://www.dacapobench.org/`.

[10] See `http://sourceforge.net/tracker/?group_id=172498` (artifact ID 2955469).

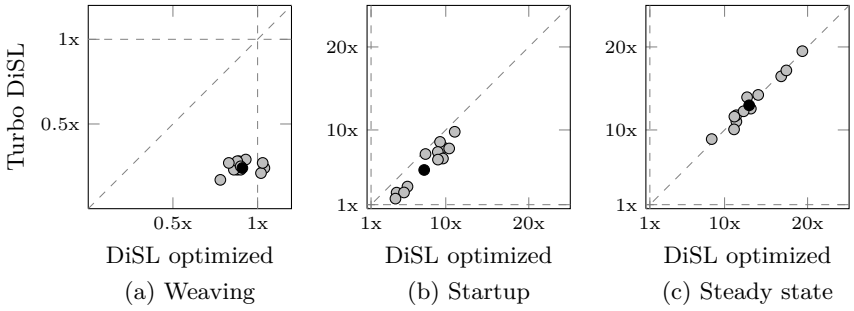

Fig. 6. Speedup factor relative to the naïve implementation without Turbo for the considered DaCapo benchmarks (values below 1x indicate slowdowns)

Regarding weave-time, Fig. 6(a), the baseline is generally faster than both the "DiSL optimized" and "Turbo DiSL" versions, because for the baseline, Turbo is deactivated and there is no guard to be evaluated at weave-time. However, in a few benchmarks "DiSL optimized" outperforms the baseline because the reduced complexity of the inlined snippet code outweighs the cost of guard evaluation. On average, "DiSL optimized" is only 10% slower than the baseline, while the use of Turbo increases weave-time by a factor of 4.2. This result clearly shows the drawback of partial evaluation, a considerable increase in weave-time.

Regarding startup performance, Fig. 6(b), "DiSL optimized" outperforms the baseline by a factor of 7.4, and "Turbo DiSL" outperforms the baseline by a factor of 5.18. Interestingly, a single benchmark iteration (which includes weave-time) is sufficient to achieve a significant speedup by partial evaluation. The manually tuned version is faster still, as it does not significantly increase weave-time.

Regarding steady-state performance, Fig. 6(c), "DiSL optimized" and "Turbo DiSL" reach the same high speedup of about 13x. This result highlights the strengths of Turbo; high steady-state performance is achieved without having to write complicated, manually tuned code. The fact that "Turbo DiSL" significantly outperforms the baseline clearly shows that the just-in-time compiler of the JVM is not able to perform the same kind of optimizations as Turbo.

6 Related Work

Partial evaluation (also called program specialization) enables aggressive interprocedural constant propagation, constant folding, and control-flow simplifications [6]. An online partial evaluator makes decisions about what to specialize during the specialization process, while an offline partial evaluator makes all the decisions before specialization. Hybrid Partial Evaluation (HPE) [12] combines both approaches by letting the programmer guide the specialization process through annotations, e.g., to indicate which objects are to be instantiated at compile time. This is similar to Turbo's annotations used to guide the partial

evaluation, without which not all optimization decisions can be made in an offline-fashion. Thus, Turbo can be considered to follow a hybrid approach, too.

Some approaches to partial evaluation are based on translating the source program into another programming language that provides more powerful specialization mechanisms. For example, Albert et al. use partial evaluation to automatically generate specialized programs by transforming Java bytecode into Prolog to apply powerful constraint logic programming [1]. The Prolog code is then interpreted by the CiaoPP abstract interpreter [5]. While this approach allows for powerful interpretative partial evaluation, it only handles a subset of Java that lacks exception handling, multi-threading, and reflection. In contrast, Turbo's partial evaluator is less powerful, but does not have such limitations.

AspectJ [7] is a language often used for the kind of bytecode instrumentation tasks DiSL is designed for. The standard AspectJ compiler (ajc) already performs partial evaluation of the aspects' pointcuts, which are akin to DiSL's markers, scopes, and guards. It does not, however, partially evaluate the aspects' advice, which are akin to DiSL's snippets. Masuhara et al. describe this approach in terms of a semantics-based compilation model [9]. This model follows an interpretative approach to compilation, based on partially evaluating the AOP interpreter itself (written in Scheme) to remove unnecessary pointcut tests. In contrast to Turbo, advice code is not partially evaluated, but rather the partial evaluator verifies if the advice should be inserted in compiled code or not.

Pesto [2] is a declarative language to describe specialization of object-oriented programs. Pesto generates all context and configuration information needed to use the JSpec offline Java partial evaluator [11], which then generates residual code in AspectJ. Like Turbo, Pesto uses guards to select specialized code when invariants are satisfied. The approach is based on the observation that partial evaluation of an object-oriented program creates new code with dependencies that cross-cut the class hierarchy. Thus, the methods generated by a given specialization can be encapsulated into a separate aspect. Whereas Turbo uses partial evaluation to optimize the execution of advice code, Pesto performs specialization of the base program using AspectJ aspects, which unfortunately do not benefit from optimizations at the advice level.

Spoon [10] is a framework for program transformation and static analysis in Java, which reifies the program with respect to a meta-model. This allows for direct access and modification of its structure at compile-time and enables template-based AOP; similar to DiSL, users can insert code, e.g., before or after a method body. Spoon, however, uses source code-level transformations. This limits its applicability for dynamic analysis, as neither basic blocks analysis nor efficient access to context information are possible. For constant propagation, dead-code elimination, and access to static context for template instantiation, Spoon provides a meta-model partial evaluation facility. Whereas Turbo performs partial evaluation of advice code, Spoon's partial evaluator specializes the meta-model; partial evaluation returns specialized models rather than code.

While Spoon reifies the entire program with respect to a meta-model, the ALIA4J approach [3] to language implementation stipulates a common meta-

model only for so-called advanced dispatching, during which one or more actions are selected depending on the current runtime context, and subsequently executed. Many, but not all bytecode instrumentation tasks possible with DiSL also fit this model. During language implementation, ALIA4J's concepts like actions, predicates, and contexts can be refined to realize the desired language semantics, e.g., by implementing the semantics based on interpretation or code generation [4]. In the latter case, the code generator is exposed to additional static information which often allows for partial evaluation of a refined concept. Unlike in Turbo, this requires manual analysis by the language implementer.

7 Conclusion

We presented Turbo, a new partial evaluator for the domain-specific aspect language DiSL [8] that targets the development of dynamic program analysis tools based on bytecode instrumentation. Turbo is designed as an optional component that is activated by the DiSL weaver during the instantiation of snippets, after static context information has been resolved. Turbo propagates constants, reduces conditionals, evaluates pure methods with constant input data, simplifies certain code patterns, and performs dead-code elimination.

The most significant benefit of Turbo is that it simplifies the DiSL programming model, as we illustrated with four case studies. The DiSL programmer does not need to factor out code to be evaluated at weave-time, but can rely on Turbo to automatically detect and optimize such code. While it is always possible to program efficient instrumentations using DiSL constructs such as guards and custom static context classes, the equivalent code relying on Turbo is generally more concise and easier to write, understand, and maintain.

Our performance evaluation confirms that a simple DiSL instrumentation optimized by Turbo can reach the same steady-state performance as a complicated, manually tuned instrumentation, at the expense of an increase in weave-time, i.e., lower startup performance. Turbo ideally supports rapid prototyping of dynamic analyses in DiSL; the programmer need not care about factoring out parts that can be evaluated at weave-time. If the analysis is applied only a few times (e.g., during workload characterization) or is applied to long-running base programs, the increase in weave-time is usually not an issue. If fast weaving is essential, for example in the case of frequently used profilers, the DiSL programmer may prefer to refactor the code using guards and custom static context classes; still, Turbo is valuable during development to explore the possible steady-state performance of an optimized analysis before implementing it by hand.

Acknowledgments

The research presented here was conducted while L. Marek was with the University of Lugano. It was supported by the Scientific Exchange Programme NMS–CH (project code 10.165), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Institutional Partnership (project no. IP04–092010), by the Swiss

National Science Foundation (project CRSII2_136225), by the National Natural Science Foundation of China (project no. 61073151), by the Science and Technology Commission of Shanghai Municipality (project no. 11530700500), by the Czech Science Foundation (project GACR P202/10/J042), as well as by CASED (www.cased.de).

References

1. Albert, E., Gómez-Zamalloa, M., Hubert, L., Puebla, G.: Verification of Java bytecode using analysis and transformation of logic programs. In: Practical Aspects of Declarative Languages, LNCS, vol. 4354, pp. 124–139. Springer Berlin/Heidelberg (2007)
2. Andersen, H.M., Schultz, U.P.: Declarative specialization for object-oriented-program specialization. In: Proceedings of the Symposium on Partial Evaluation and Program Manipulation. pp. 27–38 (2004)
3. Bockisch, C., Sewe, A., Mezini, M., Akşit, M.: An overview of ALIA4J: An execution model for advanced-dispatching languages. In: Proceedings of the 49th Conference on Objects, Models, Components, Patterns, LNCS, vol. 6705, pp. 131–146. Springer Berlin/Heidelberg (2011)
4. Bockisch, C., Sewe, A., Zandberg, M.: ALIA4J's [(just-in-time) compile-time] MOP for advanced dispatching. In: Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages. pp. 309–316 (2011)
5. Hermenegildo, M.V., Puebla, G., Bueno, F., López-García, P.: Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). Science of Computer Programming 58(1-2), 115–140 (2005)
6. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proceedings of the 15th European Conference on Object-Oriented Programming. pp. 327–353 (2001)
8. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: Proceedings of the 11th International Conference on Aspect-Oriented Software Development (2012)
9. Masuhara, H., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: Proceedings of the 12th International Conference on Compiler Construction, LNCS, vol. 2622, pp. 46–60. Springer Berlin/Heidelberg (2003)
10. Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program Analysis and Transformation in Java. Rapport, INRIA (2007), `http://hal.inria.fr/inria-00071366/en/`
11. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic Program Specialization for Java. Transactions on Programming Languages and Systems 25(4), 452–499 (2003)
12. Shali, A., Cook, W.R.: Hybrid partial evaluation. In: Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 375–390 (2011)
13. Tanter, E., Moret, P., Binder, W., Ansaloni, D.: Composition of dynamic analysis aspects. In: Proceedings of the 9th International Conference on Generative Programming and Component Engineering. pp. 113–122 (2010)