Redundancy-free Residual Dispatch

Using Ordered Binary Decision Diagrams for Efficient Dispatch

Andreas Sewe Christoph Bockisch Mira Mezini

Technische Universität Darmstadt
Hochschulstr. 10, 64289 Darmstadt, Germany
{sewe, bockisch, mezini}@st.informatik.tu-darmstadt.de

ABSTRACT

State-of-the-art implementations of common aspect-oriented languages weave residual dispatching logic for advice whose applicability cannot be determined at compile-time. But being derived from the residue's formula representation the woven code often implements an evaluation strategy which mandates redundant evaluations of atomic pointcuts. In order to improve upon the average-case run-time cost, this paper presents an alternative representation which enables efficient residual dispatch, namely ordered binary decision diagrams. In particular, this representation facilitates the complete elimination of redundant evaluations across all pointcuts sharing a join point shadow.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*Code Generation, Optimization*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms

Languages, Performance

Keywords

Advice, aspect-oriented programming, dispatch functions, ordered binary decision diagrams, pointcuts, residual dispatch

1. INTRODUCTION

This paper presents an approach for optimizing dispatch in aspect-oriented languages of the pointcut-and-advice (PA) flavor [18]. In this flavor, of which the AspectJ language [17] is the most prominent example, aspects encompass two kinds of constructs: *pointcuts* and *advice*. While advice define the actions to be performed whenever the program is in a certain state,[1] called a *join point*, their associated pointcuts define predicates on join points, thereby associating states with the advice in question.

The *pointcut language*, which is an integral part of any PA flavor language, is typically based on propositional logic or some extension thereof. Its *atomic pointcuts* designate, e.g., a **call** to a specific method and are subsequently composed by propositional operators to form more complex pointcuts. Furthermore, they generally are parameterized; such atomic pointcuts are satisfied if and only if the program's state allows for a satisfying parameter binding [23]. On the whole, these constructs are the foundation more elaborate pointcut languages based on, e.g., temporal logics can rest upon [10].

1.1 Residues and Dispatch Functions

The semantics outlined above are typically implemented in two steps [16]: *join point shadow matching* and *weaving*. First, the matching step identifies all parts of the program, called *join point shadows*, whose execution may result in a join point satisfying a given pointcut. Then, the weaving step inserts code at the join point shadows performing the actions defined by each advice. But as compilers may be unable to statically determine whether an atomic pointcut is satisfied or not at a join point shadow [3], residual dispatching logic has to be woven into the program's code. This *residue* is the result of the pointcut's partial evaluation [19].

In this setting, residual dispatch at a join point shadow can be viewed as the evaluation of a finitary Boolean function $f_\phi : \mathbb{B}^n \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$, the so-called *dispatch function* [9]; whether an advice is applicable depends solely on the prior evaluation of the n atomic pointcuts occurring in ϕ, the residue in question. Residual dispatch thus bears close resemblance to predicate dispatch [12], even though the function's range is restricted to two outcomes: an advice is either applicable or not. But this restriction is not inherent in residual dispatch; in fact, this paper extends the notion of dispatch functions to the simultaneous evaluation of all residues $\phi \in \Phi$ sharing a join point shadow. The dispatch function hence becomes $f_\Phi : \mathbb{B}^{\tilde{n}} \to \mathbb{B}^m$ and characterizes the subset of advice applicable at the join point; which combination of the $m = |\Phi|$ advice is executed depends on the state of the $\tilde{n}$ atomic pointcuts jointly occurring in Φ.

Both cases are illustrated by Figure 1, which depicts the residual dispatching logic woven for the following two advice.

[1] For uniformity of presentation, events in the program's execution, as identified, e.g., by AspectJ's **call** atomic pointcut, are treated as part of the program's state.

Seventh International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2008), April 1, 2008, Brussels, Belgium.

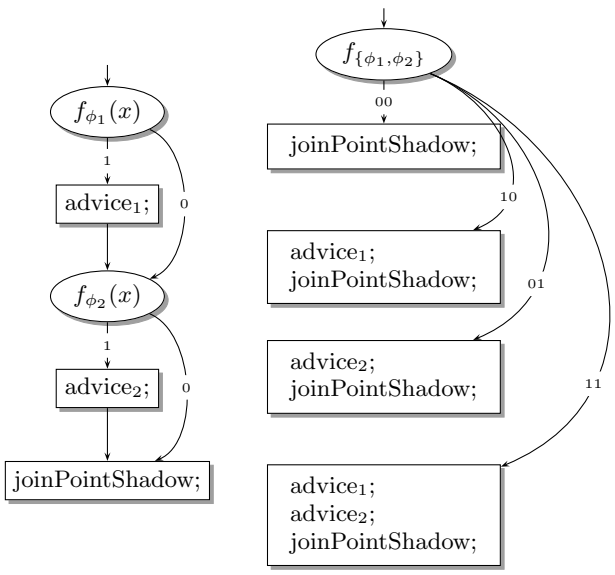

Figure 1: Two implementation schemes employing dispatch functions for residual dispatch at a join point shadow.

Hereby, the two associated pointcuts match the same join point shadow but give rise to two residues, namely ϕ_1 and ϕ_2, which are—in general—different, although there may of be identical atomic pointcuts occurring in both.

before() : joinPointShadow && ϕ_1 { advice$_1$; }
before() : joinPointShadow && ϕ_2 { advice$_2$; }

The first of the above implementation schemes resorts to two distinct dispatch functions, f_{ϕ_1} and f_{ϕ_2}, which is the approach followed by current weavers, whereas the second employs just a single dispatch function: $f_{\{\phi_1,\phi_2\}}$. Here, in a state x and based on the value of $f_{\{\phi_1,\phi_2\}}(x) \in \mathbb{B}^2$, some combination of advice is executed. But although the above suggests that the weaver generates code for each combination of advice, this need not be the case in practice (cf. Sections 4, 5.1).

1.2 Efficient Evaluation Strategies

In either case dispatch functions open up the possibility for *redundancy-free* residual dispatch. In particular, each of the n arguments of a dispatch function f_ϕ need not be evaluated more than once. By extension, any of the $\tilde{n}$ arguments of f_Φ, each of which may be shared by several residues, has to be evaluated at most once. This possibility hinges on a few assumptions on residual dispatch, though. But these assumptions typically hold for PA flavor languages in general and AspectJ in particular (cf. Section 2).

As each atomic pointcut's evaluation incurs a certain run-time cost, which varies depending on the kind of pointcut [3], it is of interest to find an *evaluation strategy* minimizing the overall cost of evaluation. Hereby, the average-case cost of evaluating $f_\Phi(x)$ for all states $x \in \mathbb{B}^{\tilde{n}}$ is most relevant to efficient dispatch, as it determines the run-time cost incurred in the long run. In contrast, the worst-case cost with respect to all states merely determines an upper bound.

Yet, regardless of the precise notion of optimality employed, the problem of finding an optimal strategy is NP-hard, as the Boolean satisfiability problem can be reduced to it.[2] Consequently, this optimization problem is usually approached heuristically. It is, e.g., often advantageous to evaluate those atomic pointcuts first whose evaluation incurs the least run-time cost. In addition to such heuristics there is a fundamental method which ensures improvement with respect to run-time costs: the elimination of redundant evaluations—ideally across residues.

1.3 Contributions and Structure of this Paper

This paper presents an approach which performs complete redundancy elimination across all pointcuts sharing a join point shadow. It furthermore makes use of the aforementioned heuristic by incorporating an ordering based on the cost of the atomic pointcuts' evaluation. To enable these optimizations, the approach employs an alternative representation of Boolean functions, namely ordered binary decision diagrams [8].

The remainder of this paper is organized as follows. First, Section 2 states the assumptions made on advice dispatch to enable the simultaneous evaluation of multiple residues. Then, Section 3 discusses advantages and disadvantages of the functions' traditional formula representation. Section 4 presents ordered binary decision diagrams as an alternative representation. Thereafter, Section 5 assesses both representations based on implementation experience and experimental results. Finally, Section 6 discusses related work, while Section 7 concludes this paper and suggests areas for future work.

2. ASSUMPTIONS ON ADVICE DISPATCH

The simultaneous evaluation of multiple residues is made possible by the three assumptions on advice dispatch stated below.

- The evaluation of an atomic pointcut is side-effect free.
- The binding of parameters is no side-effect of an atomic pointcut's evaluation.
- The execution of an advice does not affect the evaluation of a pointcut associated with another advice.

The first and second assumption make it possible to evaluate the atomic pointcuts occurring in a single residue in any order. The third assumption extends this possibility to multiple residues. Together, these assumptions allow for a clean separation between the residues' evaluation on the one hand and the advice's execution on the other hand.

The above assumptions impose only moderate restrictions on PA programs. In particular, they hold for most—if not all—programs written in AspectJ. Violations of the first assumption, although prohibited by neither *ajc* [16, 17] nor *abc* [4], the two principal compilers for the AspectJ language, are strongly advised against in the language's *Programming Guide* [2], since the order of evaluation of atomic pointcuts is implementation-specific. The second assumption is even actively enforced by *ajc* as the AspectJ language disallows ambiguous parameter bindings [23]. It should be noted, however, that *abc* resolves these ambiguities consistently rather

[2]If a non-tautological formula ϕ were satisfiable, i.e., if $\exists x \in \mathbb{B}^n . f_\phi(x) = 1$, then at least one argument of f_ϕ would have to be evaluated by an optimal strategy.

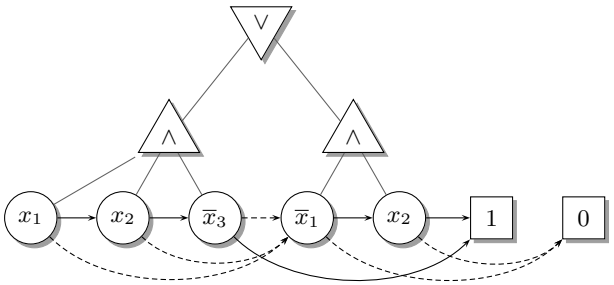

Figure 2: A formula (in DNF) and an evaluation strategy for $(x_1 \wedge x_2 \wedge \overline{x}_3) \vee (\overline{x}_1 \wedge x_2)$.

than disallowing them outright [4]; here, parameter binding becomes a side-effect of evaluation.

In contrast to the first two assumptions, which carry over from predicate dispatch [12], the third assumption is specific to advice dispatch and may indeed be violated by legal AspectJ programs: a pointcut associated with one advice can observe, by means of an **if** atomic pointcut, some state affected by another advice. The latter thus becomes what is known as narrowing advice [21]; whether the former is executed depends on the execution of the latter. But this advice-on-advice interaction is subtle, error-prone, and presumably not often used. Still, if it is used, the weaver has to resort to multiple dispatch functions instead of a joint one (cf. Section 7). Although this precludes redundancy elimination across residues, each dispatch function's evaluation strategy may still be freed from redundancies.

3. PROPOSITIONAL FORMULAS

As pointcuts are typically specified using a language akin to propositional logic, propositional formulas are a representation of considerable import; at the very least they will serve as the dispatch functions' intermediate representation. But formulas do not, by themselves, give rise to an evaluation strategy. In most programming languages their evaluation is therefore governed by a set of rules. The Java Language Specification [15], e.g., mandates left-to-right short-circuit evaluation. In contrast, AspectJ, despite its evocative use of Java's short-circuiting && and || operators, does not prescribe any particular order of evaluation. But since all atomic pointcuts are required to be side-effect free (cf. Section 2), this is not a problem but an asset; it allows for evaluation strategies optimized not only by reordering the atomic pointcuts' evaluations but also by removing redundant atomic pointcuts outright.

Still, the evaluation strategies chosen by both *ajc 1.5.4* and *abc 1.2.1* are ultimately based on the left-to-right short-circuit evaluation of a formula. Figure 2 exemplifies this. The resulting strategy is hereby depicted as an if-then-else straight-line program. By convention, the then- or 1-edges are drawn solid, whereas the else- or 0-edges are drawn dashed. If, e.g., the three atomic pointcuts are in state $x = (1,1,1)^\intercal \in \mathbb{B}^3$, then computation proceeds along the path $\langle x_1 \rightarrow x_2 \rightarrow \overline{x}_3 \dashrightarrow \overline{x}_1 \dashrightarrow 0\rangle$; hereby, the second evaluation of the atomic pointcut x_1, in the literal[3] $\overline{x}_1$, is redundant when it comes to computing the value $f(x) = 0$.

[3]A literal is either an atomic pointcut or its negation.

3.1 Redundancy Elimination

Both compilers do not directly derive evaluation strategies from a residue; instead, the strategies chosen revolve around a refined formula representation thereof: the original formula ϕ is brought into disjunctive normal form (DNF). Given this representation, either compiler generates code that performs short-circuit evaluation of the normalized formula. In addition, *ajc* makes use of the DNF representation to eliminate some redundancies by applying two laws of Boolean algebra: idempotence and boundedness. The compiler furthermore performs a minor optimization by reordering the literals in each conjunct in order of increasing runtime cost. Similarly, the conjuncts themselves are ordered. But, as Figure 2 illustrates, being limited to a two-level formula representation like DNF makes it frequently impossible to eliminate redundant evaluations of atomic pointcuts.

Some of these can, however, be eliminated after the dispatching logic has been generated by the weaver for the chosen evaluation strategy. Compilers which perform data-flow analyses like common subexpression elimination [1] can, e.g., eliminate the second evaluation of x_1 in Figure 2, which is redundant, as on all paths leading to the corresponding vertex the value of this common subexpression has already been computed. In contrast, the second evaluation of x_2 cannot avoided, as there is a path $\langle x_1 \dashrightarrow \overline{x}_1 \rightarrow x_2\rangle$ on which the value of x_2 has not previously been computed.

3.2 Extended Propositional Operations

As propositional formulas have traditionally been used to represent functions $f_{\phi_i} : \mathbb{B}^n \rightarrow \mathbb{B}$ only, AspectJ compilers generate dispatching logic that evaluates, for $i = 1, \ldots, m$, one residue ϕ_i after the other. But Boolean logic allows for a straight-forward extension to formulas which can cover the class of functions $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ for arbitrary m. Using this extension the m formulas $\phi_1, \ldots, \phi_m$ can be encoded into one. Hereby, the truth value of variables $x_1, \ldots, x_n$ is the same across all component formulas, i.e., each variable $x_1, \ldots, x_n$ still evaluates to either $\bot = (0, \ldots, 0)^\intercal \in \mathbb{B}^m$ or $\top = (1, \ldots, 1)^\intercal \in \mathbb{B}^m$. The propositional operators, however, are extended to operate component-wise on $\mathbb{B}^m$. To facilitate projections onto a single component the extension is also enriched by m constants $e_1, \ldots, e_m$ evaluating to truth values other than $\bot$ and $\top$, namely to the Boolean atoms $(1, 0, \ldots, 0)^\intercal, \ldots, (0, \ldots, 0, 1)^\intercal \in \mathbb{B}^m$. Disjunctions thereof thus cover the entire range of $\mathbb{B}^m$.

Considering the above extension, let there be m residues, represented by formulas $\phi_1, \ldots, \phi_m$ whose signatures jointly encompass the variables $x_1, \ldots, x_n$ satisfiable at a single join point shadow. Then the joint dispatch function $f_\Phi : \mathbb{B}^n \rightarrow \mathbb{B}^m$, or rather a formula representation thereof, is given by $\bigvee_{i=1}^m e_i \wedge \phi_i$. Conceptually, each residue ϕ_i is first evaluated separately. The result is then projected onto the i-th component, before all intermediate results are combined by means of disjunction. This is exemplified by Figure 3, which depicts such an extended formula with its constants distributed downwards to the level of literals. For a state $x = (0,0,0,0)^\intercal \in \mathbb{B}^4$, e.g., the subformula on the left evaluates to $(0,0)^\intercal$ whereas the subformula on the right evaluates to $(0,1)^\intercal$, which consequently is the value of $f_\Phi(x)$.

Unfortunately, this extended representation does not allow for short-circuit evaluation; while for a range of $\mathbb{B}$ the use of if-then-else instructions was sufficient for implementing evaluation strategies, code generated for the extended range

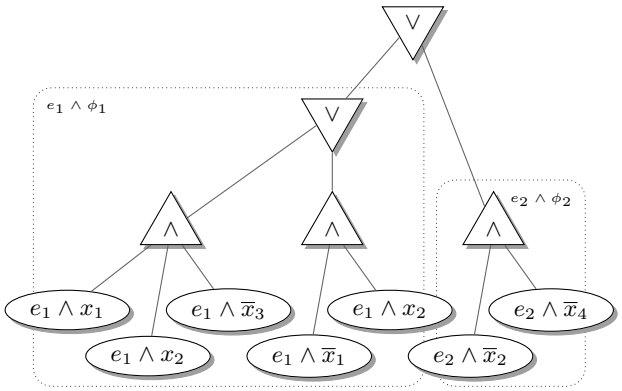

Figure 3: An extended formula, together with its two component formulas $\phi_1 = (x_1 \wedge x_2 \wedge \overline{x}_3) \vee (\overline{x}_1 \wedge x_2)$ and $\phi_2 = \overline{x}_2 \wedge \overline{x}_4$.

of $\mathbb{B}^m$ needs to evaluate the actual conjunctions and disjunctions. Nevertheless, propositional formulas are a useful representation. Not only are they the representation pointcuts are typically specified in, but they also allow for an elegant description of a pointcut's addition or removal: given an extended formula representation, the addition of another advice and its associated pointcut ϕ_{m+1} can be described simply in terms of disjunction. Similarly, their removal can be described in terms of conjunction and negation.

4. BINARY DECISION DIAGRAMS

Like formulas, binary decision diagrams (BDDs) are a natural representation of Boolean functions with a long history. But unlike the former, the latter make evaluation strategies explicit. They are thus of particular interest when such strategies are sought, as is the case when optimizing residual dispatch. Informally, a BDD is a rooted, directed, acyclic graph (DAG) built up from two kinds of vertices: *splits* and *sinks*. Hereby, all splits are labeled with variables $x_1, \ldots, x_n$ and have two outbound edges, labeled 0 and 1, respectively. Similarly, the sinks are labeled with values from a set $\mathbb{B}^m$ but do not have outbound edges.

Of the several equivalent definitions of a BDD's semantics [24], one closely reflects the intuition of a control-flow from source to sink: given a state $x \in \mathbb{B}^n$, computation begins at the root or *source* of the BDD G. At a vertex labeled x_i it then proceeds along the *low* or 0-edge if $x_i = 0$ and along the *high* or 1-edge otherwise. It finally stops, when a sink is reached; $f_G(x) \in \mathbb{B}^m$ is the value this sink is labeled with. This top-down approach to evaluation immediately gives rise to an evaluation strategy. Code generation is also straight-forward. Figure 4 exemplifies this by depicting such a strategy and thus the BDD itself as an if-then-else straight-line program. Note in particular that this BDD is *decision equivalent* to the extended formula of Figure 3; it represents the same function $f : \mathbb{B}^4 \to \mathbb{B}^2$, i.e., for every state $x \in \mathbb{B}^4$ evaluation of either representation results in the same value $f(x)$. But, as exposed by the BDD representation, ϕ_1 and ϕ_2 cannot be satisfied simultaneously, i.e., $f(x) \neq (1,1)^\intercal$; if code is generated for each combination of advice (cf. Section 1.2), then this fact may be exploited to avoid code generation for all combinations of advice.

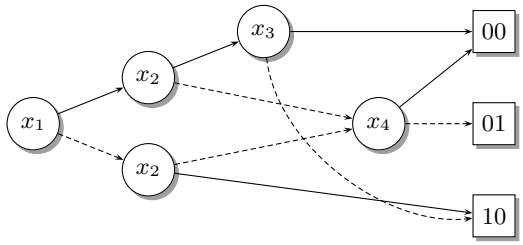

Figure 4: A BDD representation decision equivalent to the extended formula of Figure 3.

4.1 Redundancy Elimination

Since a BDD representation corresponds to an evaluation strategy, redundant evaluations of atomic pointcuts can be characterized by a simple syntactic property: the existence of paths from source to sink on which more than one split is labeled with the same variable or, equivalently, the existence of paths which are not taken for any state x [24]; BDDs without such inconsistent paths are called *free* or *read-once*.

While the read-once property is generally desirable, the conversion from unconstrained BDDs to free ones can cause an exponential blow-up in terms of size [24]. This blow-up is, however, in general no worse than that which a formula's conversion to either CNF or DNF can cause. Still, there are functions which allow for a polynomial-size normal form but which require a free BDD (FBDD) whose size is exponential in the number of variables [6]. But this is also true vice versa. Thus, the representational power of FBDDs, DNFs, and CNFs is only comparable on a case by case basis; neither representation is smaller for all functions.

Yet, there is one assumption one can reasonably make about the functions encountered during residual dispatch: they stem from a small and structured formula representation. This is because these formulas correspond to the pointcuts as written. It is hence reasonable to restrict the discussion to those formulas which are likely to form a pointcut in real-world programs. But this set of small, structured formulas is difficult to characterize. Nevertheless, such a characterization will be attempted in Section 5.2.

4.2 Propositional Operations

Given two (free) BDDs G and H, the problem of computing their conjunction or disjunction, a task known as *synthesis*, is NP-hard. (Negation, however, can trivially be applied by negating the value of all sinks.) Thus, a variety of syntactic constraints has been imposed on BDDs in order to facilitate efficient synthesis [22, 14]. The most prominent constraint [8] gives rise to a subclass of BDDs known as ordered binary decision diagrams (OBDDs): on each path from source to sink the variables are required to occur in the same order π. The OBDD shown in Figure 4, e.g., is a π-ordered representation of the extended formula of Figure 3, where $\pi = \langle x_1, x_2, x_3, x_4 \rangle$.

Obviously, every OBDD is free and hence gives rise to a redundancy-free evaluation strategy. The converse, however, is false; thus, the representational power of OBDDs is strictly smaller than that of FBDDs—although not smaller than that of either CNF or DNF (cf. Section 4.1). Nevertheless, OBDDs are of particular interest when dispatch functions are to be represented. In this setting, being con-

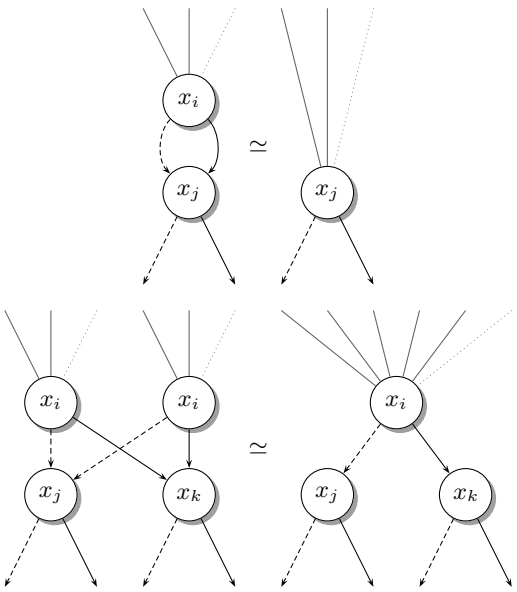

Figure 5: The *deletion rule* and the *merging rule*.

strained to a fixed variable ordering π is only a moderate impediment. In fact, ordering the atomic pointcuts simply in the order of increasing run-time cost is a heuristic which performs well in experiments (cf. Section 5.2).

Provided that both operands are π-ordered, there exists an efficient algorithm for computing the disjunction or conjunction of two OBDDs G and H [8]; employing memoization, synthesis is performed in $O(|G||H|)$. While originally devised for OBDD representations of functions $f : \mathbb{B}^n \to \mathbb{B}$, the algorithm can easily be adapted to sinks labeled with values from the larger set $\mathbb{B}^m$. It can furthermore be modified such that the result of each step is reduced, i.e., the number of vertices is minimized [7]. This is done by repeatedly applying the two *reduction rules* shown in Figure 5. Applying these rules during each synthesis step is desirable not only because it keeps intermediate results small, but also because it minimizes the size of the final OBDD.

5. ASSESSMENT

When assessing the utility of OBDD-based dispatch functions, two questions have to be answered: whether dispatch functions can be easily integrated with a compiler or an aspect-aware execution environment and whether an OBDD representation thereof can improve upon the average-case run-time cost of residual dispatch.

5.1 Implementation Experience

OBDD-based dispatch functions were implemented and integrated with an experimental execution environment for PA flavor languages developed as part of the Aspect Language Implementation Architecture project [5];[4] they supplanted the previous, DNF-based residual dispatching logic. Furthermore, dispatch functions were incorporated into the framework the environment builds on, with the abstraction completely hiding the chosen representation; whether the functions are represented by means of formulas, OBDDs, or even a combination thereof is immaterial to the framework itself. Only when the residual dispatching logic needs to be woven by an instantiation of the framework, e.g., a compiler or an execution environment, the functions' concrete representations have to be considered. The weaving approach which the default instantiation hereby follows avoids generating code for each combination of advice. Instead, it computes the joint dispatch function's value and, based on this, dispatches the applicable advice one after the other.

Overall, the changes required by the integration to both the framework and its instantiation were minimal. This fact can serve as indication that the notion of dispatch functions is a natural one. Deployment and undeployment of aspects in particular were found to be easily implementable in terms of the extended propositional operations (cf. Section 3.2).

5.2 Experimental Results

Traditionally, the complexity theory of Boolean functions has considered the class of functions $f : \mathbb{B}^n \to \mathbb{B}$ only. Consequently, any representation's expressiveness has been assessed primarily with respect to this class. When a representation suitable for residual dispatch has to be chosen, however, this assessment is of limited usefulness as all dispatch functions ultimately stem from pointcuts and their respective residues. Thus, an attempt was made to characterize the formulas likely to form residues in real-world programs: these *non-trivial* but *simple* formulas are those which are not decision equivalent to either $\bot$ or $\top$ and cannot be simplified by applying the laws of idempotence and boundedness.

Although there are, e.g., $2^{2^5} = 4,294,967,296$ Boolean functions in five arguments, there are only about 118 million non-trivial, simple formulas of signature $\langle x_1, \ldots, x_5 \rangle$ with up to six propositional operators; this set, which contains, e.g., the formula of Figure 2, is at the same time large enough to encompass most residues encountered in practice but also small enough to experiment with. It should be noted, however, that it contains various decision equivalent formulas. During the course of the experiments these were treated as distinct, for they may give rise to distinct evaluation strategies. The cost incurred by evaluating each of the five atomic pointcuts was assumed to be 1.0, 1.5, 2.0, 2.5, and 3.0, respectively, which reflects the range of relative costs observed for common atomic pointcuts like **this** and **cflow**.

Figure 6 charts the average-case costs of evaluation strategies derived from the original formulas and their DNF counterparts, respectively. Each data point hereby corresponds to numerous pairs of formula and DNF; its shade indicates how many representations give rise to a particular combination of average-case costs. While there are formulas whose DNF representation is considerably larger and thus incurs higher cost of evaluation, it is noteworthy that there are numerous formulas which benefit from conversion to this representation. The figure depicts these cases as data points above and below the bisector, respectively.

The aforementioned result is due to two causes: first, conversion allows for simplification to be applied twice; the laws of idempotence and boundedness were employed once before and once after conversion to DNF. Second, literals and conjuncts were reordered according to the run-time costs incurred by their evaluation. These two optimizations are

[4]The implementation is available to the public: `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/ALIA/alia.html`.

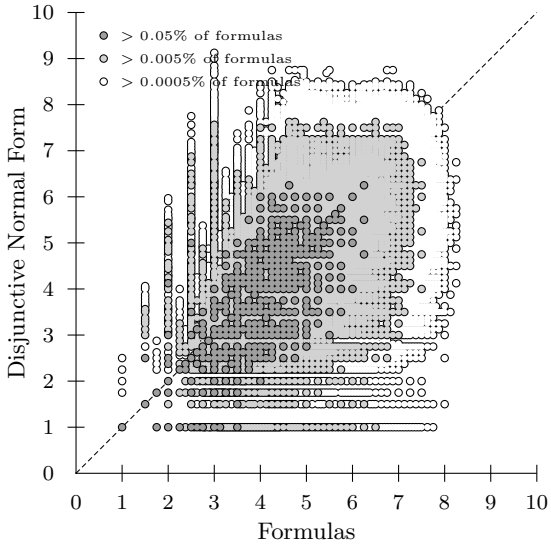

Figure 6: The average-case costs of evaluation strategies derived from formulas and their DNF.

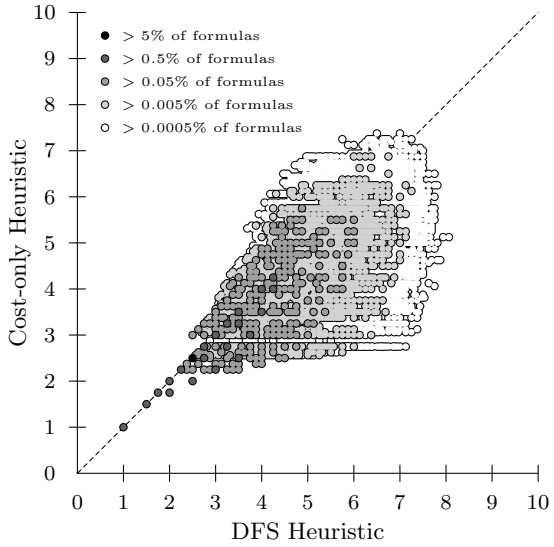

Figure 7: The average-case costs of evaluation strategies derived from OBDDs ordered with respect to two straight-forward heuristics.

precisely those used by *ajc* (cf. Section 3.1) and here have been proven effective; the evaluation cost averaged over all simple formulas considered is 4.398 for DNF representations, whereas their left-to-right short-circuit evaluation results in an average cost of 4.603.

In contrast to this marginal difference OBDD representations offer significant improvements over either formula-based representation. These improvements are only partly owed to the redundancy-free nature of OBDDs; the variable ordering chosen is also important, as illustrated by Figure 7. One of the variable ordering heuristics applied here, the so-called depth-first search (DFS) heuristic [13], derives an ordering from a depth-first traversal of the original formula. The second heuristic applied, the cost-only heuristic, derives an ordering from the costs incurred by the variables' evaluation. While the former heuristic ignores the atomic pointcuts' cost, the latter ignores the residue's structure. Both are straight-forward, but when the average-case cost of evaluation strategies is the primary concern, the cost-only heuristic was found to be superior to the DFS heuristic.

For 47.6% of the formulas considered, the cost-only heuristic gives rise to a strategy with average-case cost superior to that derived using the DFS heuristic. The latter heuristic is superior in only 18.2% of the cases. But with 3.438 and 3.721, respectively, the evaluation costs averaged over all formulas in either case are lower than that of the two formula-based representations.

6. RELATED WORK

Dispatch functions of the form $h : \mathbb{B}^n \to \{1, \ldots, m\}$ play an important role in predicate dispatch [12] and bear a close resemblance to the functions employed during advice dispatch [20]; in this setting, the n atomic predicates determine which of the m methods is ultimately executed. When the dispatch function is viewed as a composition $h = g \circ f$, with $f : \mathbb{B}^n \to \mathbb{B}^m$ and $g : \mathbb{B}^m \to \{1, \ldots, m\}$, this resemblance is most prominent. Hereby f characterizes applicability, whereas g determines the overriding relationship. This decomposition is particularly advantageous if, as suggested by the present paper, an OBDD representation is used for the former function. First, a representation of f is synthesized by applying the extended propositional operations (cf. Sections 3.2, 4.2). Then the sinks are relabeled according to g. Finally, subsequent applications of the reduction rules (cf. Figure 5) yield the reduced OBDD representing h.

Decision diagrams have been employed, under the name of *lookup DAGs*, for the efficient implementation of both multiple and predicate dispatch [9]. But these decision diagrams are not necessarily binary; this complicates synthesis. For residual dispatch, however, this complication is unnecessary as atomic pointcuts are propositional in nature. Construction of lookup DAGs is further complicated by the fact that the synthesis algorithm requires a canonical formula representation, namely DNF, to be used. In some cases, this can cause an exponential blow-up of intermediate results even though the size of the final result is moderate. Also, as the range of f is specified with respect to the DNFs' conjuncts instead of the m predicates themselves, this view prevents the straight-forward but elegant description of the addition or removal of predicates in terms of propositional operations (cf. Section 3.2). This paper therefore links the work on lookup DAGs with the theory of Boolean functions in general and that of BDDs in particular [24].

7. CONCLUSIONS AND FUTURE WORK

This paper has shown that, under three assumptions which typically hold for PA flavor languages, the average-case runtime cost incurred by residual dispatch can be improved upon by applying two optimizations. The first and foremost of these, namely the complete redundancy elimination across multiple residues, is facilitated by an alternative representation of the dispatch function in question: OBDDs. This representation does also allow for a straight-forward implementation of the second optimization, namely the reordering of atomic pointcuts in order of increasing cost. Fur-

thermore, as OBDDs are, like formulas, a representation of dispatch functions, they, too, allow for an elegant description of aspect deployment and undeployment.

It should be noted, however, that, strictly speaking, only point-in-time semantics [11] allow for the one-to-one correspondence between dispatch functions and join point shadows this paper so far has alluded to. Yet, dispatch functions can be used to good effect with AspectJ's region-in-time semantics [17] as well. They merely necessitate a separate treatment of **after returning** and **after throwing** advice. This is necessary, since in either case a parameter may be bound which is unavailable at the beginning of the join point's region-in-time. A straight-forward solution would require two dispatch functions, which handle the beginning of the join point's region-in-time and the end of the join point's region-in-time, respectively. Using several dispatch functions may also be advantageous in the presence of **around** advice which do not **proceed**. While such an advice may preclude the execution of other advice and is therefore narrowing, it leaves their pointcuts' satisfiability unaffected and hence does not violate the assumptions of Section 2. The precise workings of this scheme are, however, an area for future work.

Other areas for future work include methods which exploit static information, e.g., the fact that **args**(Integer) implies **args**(Number) or which perform profile-guided optimizations whose notion of optimality is based not on the average- but on the expected-case cost. In these areas, however, experiments require a detailed model of both the interdependencies of atomic pointcuts and the probability distribution underlying the set of states; such a model does not yet exist.

8. ACKNOWLEDGEMENTS

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

9. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, MA, USA, 2nd edition, 2006.

[2] The AspectJ Project. *The AspectJ Programming Guide*. `http://www.eclipse.org/aspectj/doc/released/progguide/`.

[3] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. *ACM SIGPLAN Notices*, 40(6), 2005.

[4] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development I*. Springer, Berlin, Germany, 2006.

[5] C. Bockisch, M. Mezini, W. Havinga, L. Bergmans, and K. Gybels. Reference model implementation. Technical Report AOSD-Europe Deliverable D96, Technische Universität Darmstadt, 2007.

[6] B. Bollig and I. Wegener. A very simple function that requires exponential size read-once branching programs. *Information Processing Letters*, 66(2), 1998.

[7] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, 1990.

[8] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.

[9] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. *ACM SIGPLAN Notices*, 34(10), 1999.

[10] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Proceedings of the 22nd OOPSLA Conference*, 2007.

[11] Y. Endoh, H. Masuhara, and A. Yonezawa. Continuation join points. In *Proceedings of the 5th FOAL Workshop*, 2006.

[12] M. D. Ernst, C. S. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the 12th ECOOP Conference*, 1998.

[13] M. Fujita, H. Fujisawa, and Y. Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems*, 12(1), 1993.

[14] J. Gergov and C. Meinel. Efficient Boolean manipulation with OBDDs can be extended to FBDDs. *IEEE Transactions on Computers*, 43(10), 1994.

[15] J. Gosling, W. N. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Reading, MA, USA, 3rd edition, 2005.

[16] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd AOSD Conference*, 2004.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th ECOOP Conference*, 2001.

[18] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of the 17th ECOOP Conference*, 2003.

[19] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of the 12th Conference on Compiler Construction*, 2003.

[20] D. Orleans. Incremental programming with extensible decisions. In *Proceedings of the 1st AOSD Conference*, 2002.

[21] M. C. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM SIGSOFT*, 2004.

[22] D. Sieling and I. Wegener. Graph driven BDDs: A new data structure for Boolean functions. *Theoretical Computer Science*, 141(1 & 2), 1995.

[23] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5), 2004.

[24] I. Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.