Intraprocedural Dataflow Analysis for Software Product Lines

Claus Brabrand[1,2], Márcio Ribeiro[2,3],
Társis Tolêdo[2], Johnni Winther[4], and Paulo Borba[2]

[1] IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen, Denmark
[2] Federal University of Pernambuco, Av. Prof. Luis Freire, 50.740-540 Recife, Brazil
[3] Federal University of Alagoas, Av. Lourival de M. Mota, 57.072-970 Maceió, Brazil
[4] Aarhus University, Nordre Ringgade 1, 8000 Aarhus, Denmark

brabrand@itu.dk,{mmr3,twt,phmb}@cin.ufpe.br,jw@cs.au.dk

Abstract. Software product lines (SPLs) developed using annotative approaches such as conditional compilation come with an inherent risk of constructing erroneous products. For this reason, it is essential to be able to analyze such SPLs. However, as dataflow analysis techniques are not able to deal with SPLs, developers must generate and analyze all valid products individually, which is expensive for non-trivial SPLs.
In this paper, we demonstrate how to take *any* standard intraprocedural dataflow analysis and *automatically* turn it into a *feature-sensitive* dataflow analysis in five different ways where the last is a combination of the other four. All analyses are capable of analyzing *all* valid products of an SPL without having to generate all of them explicitly.
We have implemented all analyses using `SOOT`'s intraprocedural dataflow analysis framework and experimentally evaluated four of them according to their performance and memory characteristics on five qualitatively different SPLs. On our benchmarks, the combined analysis strategy is up to almost eight times faster than the brute-force approach.

Keywords: Dataflow Analysis, Software Product Lines

1 Introduction

A software product line (SPL) is a set of software products that share common functionality and are generated from reusable assets. These assets specify common and variant behavior targeted at a specific set of products, usually bringing productivity and time-to-market improvements [1, 2]. Developers often implement variant behavior and associated features with conditional compilation constructs like `#ifdef` [3, 4], mixing common, optional, and even alternative and conflicting behavior in the same code asset. In these cases, assets are not valid programs or program elements in the underlying language. We can, however, use assets to generate valid programs by evaluating the conditional compilation constructs using preprocessing tools. That is, a conventional program corresponding

to a particular product may be derived by the SPL by selecting the particular configuration (set of enabled features) corresponding to that product.

Since code assets might not be valid programs or program elements, existing standard dataflow analyses, which are for instance essential for supporting optimization [5] and maintenance [6] tasks, cannot be directly used to analyze code assets. To analyze an SPL using intraprocedural analysis, developers then have to generate all possible methods and separately analyze each one with conventional single-program dataflow analyses. In this case, generating and analyzing each method can be expensive for non-trivial SPLs. Consequently, interactive tools for single-program development might not be usable for SPL development because they rely on fast dataflow analyses and have to be able to quickly respond when the programmer performs tasks such as code refactoring [7]. Also, this is bad for maintenance tools [6] that help developers understand and manage dependencies between features.

To solve this problem and enable more efficient dataflow analysis of SPLs, we propose four approaches for taking any standard dataflow analysis and automatically lifting it into a corresponding *feature-sensitive* analysis that we can use to directly analyze code assets. We also show how to combine these four approaches into a fifth combined analysis strategy.

Our feature-sensitive approaches are capable of analyzing all configurations of an SPL without having to generate all of them explicitly. For this reason, we expect our solutions to be significantly faster than the naive brute force strategy of explicitly generating all products and analyzing them. Although we focus on SPLs developed with conditional compilation constructs, our results apply to other similar annotative variability mechanisms [3]. Our results could also apply to SPLs implemented using a compositional approach through the bidirectional transformation proposed by Kaestner et al. [9] which is capable of refactoring physically separated SPL approaches into virtually separated ones based on conditional compilation.

We evaluate our three of our *feature-sensitive* approaches and compare them with a *brute-force* intraprocedural approach that generates and analyzes all possible methods individually. We report on a number of performance and memory consumption experiments using the ubiquitous dataflow analysis, *reaching definitions* [8], and five SPLs from different domains, with qualitatively different numbers of features, products, `#ifdef` statements, and other factors that might impact performance and memory usage results.

From experimental data we derive an analysis strategy which intraproceduralally combines our above analyses to heuristically select a good analysis for a given method. This combined strategy is faster than all individual analyses on all benchmarks. On the benchmarks, our combined feature-sensitive strategy is up to eight times faster than the feature-oblivious brute-force analysis.

1.1 List of Contributions

Our paper presents the following contributions:

- Five qualitatively different ways of automatically deriving *feature-sensitive* analyses capable of analyzing all configurations of a SPL from a conventional *feature-oblivious* dataflow analysis, $\mathcal{A}0$; in particular:
  - $\mathcal{A}1$, a *consecutive* approach that analyzes all configurations, one at a time;
  - $\mathcal{A}2$, a *simultaneous* approach that analyzes all configurations at the same time;
  - $\mathcal{A}3$, a *simultaneous with sharing* approach that in addition to the simultanous approach is capable of sharing information among configurations;
  - $\mathcal{A}4$, a *simultaneous with sharing and merging* approach that in addition to the previous analysis is also capable of merging equivalent configurations during analysis; and
  - $\mathcal{A}*$, a *combined analysis strategy* which combines the feature-sensitive analyses to achieve an even faster analysis strategy.

  The analyses $\mathcal{A}1$ to $\mathcal{A}4$ introduce variability gradually into dataflow analysis.
- A proof of equivalence of our feature-sensitive analyses, $\mathcal{A}1$ to $\mathcal{A}4$.
- *Experimental validation* of our ideas via an implementation using the ubiquitous reaching definitions dataflow analysis on five SPL benchmarks.

In this paper, compared to earlier work on analyzing SPLs [10], we specify one more feature-sensitive analysis method ($\mathcal{A}4$); although implemented, this analysis is not used in the comparative evaluation of the analyses. We provide proof of equivalance of all feature-sensitive analyses proposed. Also, we show how to combine the feature-sensitive analyses so as to achieve an even faster analysis. This paper also provides one more SPL benchmark in the evaluation, more motivating examples, more on the `#ifdefs` normalization process, and more relation to earlier work on analyzing SPLs.

1.2 Organization of the Paper

The paper is organized as follows. Using concrete examples, we discuss and motivate the need for dataflow analysis of SPLs (Section 2). Then, we introduce conditional compilation and feature models (Section 3). After that, we briefly recall basic dataflow analysis concepts (Section 4) and present the main contributions of this paper: feature-sensitive dataflow analyses for SPLs (Section 5). Then, we analyse important properties of our feature-sensitive analyses (Section 6) and evaluate them (Section 7). Finally, we consider related work (Section 8), and conclude (Section 9).

2 Motivating Examples

To better illustrate the issues we are addressing in this paper, we now present a motivating example based on the `Lampiro` SPL[1]. `Lampiro` is an instant-messaging client developed in Java ME and its features are implemented using `#ifdef`s.

[1] `http://lampiro.bluendo.com/`

```
Image logo;
...
#ifndef GLIDER
... logo = Image.create("icon.png"); ...
#endif
...
UILabel uimg = new UILabel(logo);
```

(a) Uninitialized variable analysis example (error when `GLIDER` is enabled).

```
Display display = null;
...
#ifdef VIDEO
... display = new VideoDisplay(); ...
#endif
...
display.init();
```

(b) Null-pointer analysis example (error when `VIDEO` is disabled).

```
Outputter out = new Outputter();
...
out.open();
...
#ifdef HIGH
... out.close(); ...
#endif
...
#ifdef LOW
... out.close(); ...
#endif
...
```

(c) Library usage analysis example (inappropriate use when `HIGH` *and* `LOW` are both enabled).

```
String s, t;
String password = "ToP-sEcReT";
...
#ifdef OOPS
... s = password; ...
#endif
...
#ifdef YIKES or GOSH
... t = "msg: " + s; ...
#endif
...
out.print(t);
```

(d) Information flow analysis example (password disclosed when `OOPS` *and* (`YIKES` *or* `GOSH`) are enabled).

Fig. 1. Four example SPL fragments with qualitatively different kinds of errors.

Figure 1(a) shows the essence of a code snippet extracted from `Lampiro` implemented in Java with the Antenna[2] preprocessor. As can be seen, if the `GLIDER` feature is *not* enabled (see the `#ifndef` statement), the `logo` variable is assigned an image instantiated by the `createImage` method and is thereby *initialized*. However, in products where `GLIDER` feature *is* enabled, this variable is *uninitialized*.

Figure 1(b) illustrates another common error in an SPL. Here, the `display` variable is assigned a new `VideoDisplay` object whenever the feature `VIDEO` is enabled. However, if this feature is disabled, the last line of the example program will produce a *null-pointer exception* when the `init` method is invoked on `display` (which is `null`).

Both examples have been errors inherent to the Java language. Inappropriate usage of libraries can also cause SPL errors. Also, errors are not necessarily as simple as suggested by the previous examples, limited to whether or not a single `#ifdef` block is enabled or disabled. Errors may also depend intricately on combinations of `#ifdef` blocks. The code fragment in Figure 1(c) sketches a simple combination case that appear to contain an erroneous "double closing" of an output resource (the resource is closed in both `#ifdef`s). However, it might be the case that the `HIGH` and `LOW` features are so-called *mutually exclusive features*; i.e., `HIGH` is enabled if and only if `LOW` is disabled. The SPL will then, in fact, not have the double closing error. In order to analyze SPL programs we thus

[2] `http://antenna.sourceforge.net/`

need to know which combinations of features are designated as valid. We return to this point later.

Figure 1(d) shows yet another kind of error (inspired by [11]) where unintended information flow through the program compromizes sensitive information. In this contrived example, the password is disclosed in products where the features `OOPS` *and* (`YIKES` *or* `GOSH`) are enabled.

In this paper, we focus exclusively on errors that can be detected via conventional dataflow analyses and show how to lift these from analyzing single programs to program families developed via SPLs. Figure 1 showed four examples of such analyses: *uninitialized variables*, *null-pointers*, *inappropriate library usage*, and *unintended information flow*. The two examples at the bottom of the figure further illustrate the importance of taking into account the combinations of features (i.e., configurations) in that a potential error discovered is in fact only a real error if the configuration in which it occurs is designated as valid (cf. Section 3.1).

Syntax and type errors are beyond the scope of this paper. We assume the SPLs to be analyzed are free from these classes of errors. (For more on how to handle such errors in SPLs, we refer to [12, 13].)

The above examples illustrate possible errors when developing SPLs using conditional compilation. Even though some researchers argue that `#ifdef`s may pollute the code, may lack separation of concerns, and may make certain maintenance tasks harder [14–17], conditonal compilation continues to be a widespread mechanism for implementing variability in SPLs.

As with conventional programs, when maintaining SPLs, it is important to be able to be able to discover errors as early on in the software development cycle to minimize cost and the consequences of errors. In traditional single programs, we distinguish between *analysis-time*, *compile-time*, and *runtime*. In SPLs, we have an extra first step, namely that of *instantiation time* (aka., *configuration time*) where features are selected and a preprocessor turns the assets into a conventional program (product) which can then be analyzed, compiled, and run. In other words, for SPLs, it is better to detect errors at instantiation time than (product) analysis time, compilation time, or runtime. Lifting dataflow analyses to SPLs, will help transfer error discovery from (product) analysis time to instantiation time.

All errors in Figure 1 can be caught by conventional dataflow analyses analyzing for *uninitialized variables*, *null-pointers*, *object state*, and *information flow*. However, such conventional analyses need to be "lifted" to analyze not a single program, but a *family of programs* expressed as a software product line.

Another analysis is of particular interest to us; namely the classical *reaching definitions* analysis. This analysis is commonly used to produce a *definition-usage* (aka., *def-use*) graph which may in turn be used to compute data dependencies among different parts of a program. This analysis is particularly interesting for SPLs in cases where multiple developers collaborate on programming an SPL with each developer responsible for his own set of features. In such cases, one developer changing the value of a variable belonging to a feature he or she

is maintaining might inadvertently influence or break other features maintained by another programmer. The reaching definitions analysis could then be used to warn the first programmer of the other features his change may affect. We have proposed the idea of providing information about this kind of feature dependency in an SPL [6]. This was our original motivation for adapting dataflow analysis for SPLs. In this paper, we will use the *reaching definition analysis* as our analysis benchmark.

To intercept dependencies and errors like the ones we have seen above, we need dataflow analyses to work on sets of SPL assets, like the ones using conditional compilation. However, programmers must resort to generating all possible methods and separately analyzing each one by using the conventional single-program dataflow analysis. Depending on the size of the SPL, this can be costly, which may be a problem for interactive tools that analyze SPL code, for example. As we shall see in Section 7, we are able to decrease such costs.

3 Conditional Compilation

In this section, we briefly introduce the `#ifdef` construct and *feature models*. We use a simplified `ifdef` construct the syntax of which is:

$$\begin{array}{lll} S & ::= & \texttt{"ifdef" "("}\ \phi\ \texttt{")"}\ S \\ \phi & ::= & f \in \mathbb{F} \ \mid\ \neg\phi \ \mid\ \phi \wedge \phi \end{array}$$

where S is a *Java Statement* and ϕ is a *propositional logic formula* over *feature names* where f is drawn from a finite alphabet of feature names, $\mathbb{F}$. Throughout the paper, however, we will use formulae from the full propositional logic extended with *true*, *false*, *disjunction*, *implication*, and *bi-implication* via syntactic sugar in the usual way.

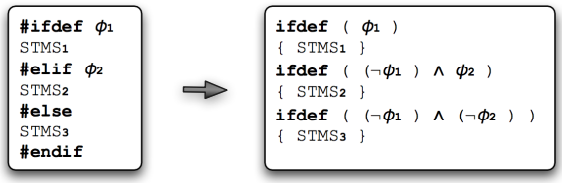


Fig. 2. Example of normalization of `#ifdef` statements.

In general, it is possible to make un-syntactic undisciplined `#ifdef` annotations (e.g., encapsulating *half* of a statement in an `#ifdef`). Dealing with such constructions are beyond the scope of this paper. (Such vulgar constructions could be dealt with by translating them into disciplined `#ifdef`s respecting the

syntactic structure of the underlying programming language [18, 12].) We further refactor these disciplined `#ifdef`s to eliminate any optional `#elif` and `#else` branches they might have by turning them into the normalized syntactic `ifdef` form listed in the BNF above. Figure 2 exemplifies this normalization process.

A *configuration*, $c \subseteq \mathbb{F}$, is a set of *enabled features*. A propositional logic formula, ϕ, gives rise to the *set of configurations*, $[\![\phi]\!] \subseteq 2^{\mathbb{F}}$, for which the formula is satisfied. For instance, given $\mathbb{F} = \{A, B, C\}$, the formula, $\phi = A \wedge (B \vee C)$ corresponds to the following set of configurations:

$$[\![A \wedge (B \vee C)]\!] = \{\{A, B\}, \{A, C\}, \{A, B, C\}\} \subseteq 2^{\mathbb{F}}$$

3.1 Feature Model

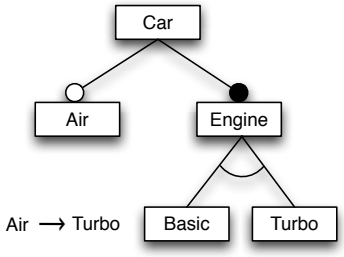


Fig. 3. Diagrammatic representation of a feature model for a car.

To yield only *valid* configurations, sets of configurations are usually further restricted by a so-called *feature model* often represented diagrammatically [19] as in Figure 3. Conceptually, however, when disregarding its structure, focussing only on validity of configurations, a feature model is just a propositional logic formula. Here is the feature model of Figure 3 represented as a propositional formula over the alphabet $\mathbb{F} = \{\texttt{Car}, \texttt{Engine}, \texttt{Air}, \texttt{Basic}, \texttt{Turbo}\}$:

$$\psi_{\text{FM}} = \texttt{Car} \ \wedge \ \texttt{Engine} \ \wedge \ (\texttt{Basic} \leftrightarrow \neg\texttt{Turbo}) \ \wedge \ (\texttt{Air} \rightarrow \texttt{Turbo})$$

corresponding to the following set of *valid* configurations:

$$\begin{aligned}[\![\psi_{\text{FM}}]\!] = \{&\{\texttt{Car}, \texttt{Engine}, \texttt{Basic}\},\\ &\{\texttt{Car}, \texttt{Engine}, \texttt{Turbo}\},\\ &\{\texttt{Car}, \texttt{Engine}, \texttt{Air}, \texttt{Turbo}\}\} \subseteq 2^{\mathbb{F}}\end{aligned}$$

Let $|\psi|$ denote the *number of configurations* in the interpretation of the formula ψ (i.e., $|\psi| =_{\text{def}} |[\![\psi]\!]|$). Notice that the number of *valid* configurations, $|\psi_{\text{FM}}|$, may be significantly smaller than the total number of configurations, $|2^{\mathbb{F}}|$. Indeed, in the above example we have that: $3 = |\psi_{\text{FM}}| < |2^{\mathbb{F}}| = 32$.

4 Dataflow Analysis

A Dataflow Analysis [5] is comprised of three constituents: 1) a *control-flow graph* (on which the analysis is performed); 2) a *lattice* (representing values of interest for the analysis); *and* 3) *transfer functions* (that simulate execution at compile-time). In the following, we briefly recall each of the constituents of the conventional (feature-oblivious) single-program dataflow analysis and how they may be combined to analyze an input program.

Control-Flow Graph: The *control-flow graph* (CFG) is the abstraction of an input program on which a dataflow analysis runs. A CFG is a directed graph where the nodes are the statements of the input program and the edges represent *flow of control* according to the semantics of the programming language. An analysis may be *intraprocedural* or *interprocedural*, depending on how functions are handled in the CFG. Here, we only consider intraprocedural analyses.

Lattice: The information calculated by a dataflow analysis is arranged in a *lattice*, $\mathcal{L} = (D, \sqsubseteq)$ where D is a set of elements and $\sqsubseteq$ is a *partial-order* on the elements [8]. Lattices are usually described diagrammatically using *Hasse*

(a) Lattice for sign analysis.

(b) Effect of transfer function, f_S.

Fig. 4. Lattice and transfer function.

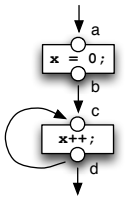
Diagrams which use the convention that $x \sqsubset y$ if and only if x is depicted *below* y in the diagram (according to the lines of the diagram). Figure 4(a) depicts such a diagram of a lattice for analyzing the sign of an integer. Each element of the lattice captures information of interest to the analysis; e.g., "`+`" represents the fact that a value is always *positive*, "`0/+`" that a value is always *zero-or-positive*. A lattice has two special elements; $\bot$ at the bottom of the lattice usually means *"not analyzed yet"* whereas $\top$ at the top of the lattice usually means *"analysis doesn't know"*. The partial order induces a *least upper bound* operator, $\sqcup$, on the lattice elements [8] which is used to *combine* information during the analysis, when control-flows meet. For instance, $\bot \sqcup \texttt{0} = \texttt{0}$, $\texttt{0} \sqcup \texttt{+} = \texttt{0/+}$, and $\texttt{-} \sqcup \texttt{0/+} = \top$.

Transfer Functions: Each statement, S, will have an associated *transfer function*, $f_S : \mathcal{L} \to \mathcal{L}$, which simulates the execution of S at compile-time (with respect to what is being analyzed). Figure 4(b) illustrates the effect of executing transfer function f_S. Lattice element, ℓ, flows into the statement node, the transfer function computes $\ell' = f_S(\ell)$, and the result, ℓ', flows out of the node.

Here are the transfer functions for two assignment statements for analysing the sign of variable x using the sign lattice in Figure 4(a):

$$f_{\texttt{x=0}}(\ell) = \texttt{0} \qquad f_{\texttt{x++}}(\ell) = \begin{cases} \top & \ell \in \{\texttt{-/+}, \texttt{-/0}, \top\} \\ \texttt{+} & \ell \in \{\texttt{0}, \texttt{+}, \texttt{0/+}\} \\ \texttt{-/0} & \ell = \texttt{-} \\ \bot & \ell = \bot \end{cases}$$

The transfer function, $f_{\texttt{x=0}}$, is the constant zero function capturing the fact that x will always have the value *zero* after execution of the statement `x=0` irrespective of the original argument, ℓ, prior to execution. The transfer function, $f_{\texttt{x++}}$, simulates execution of `x++`; e.g., if x was *negative* ($\ell = \texttt{-}$) prior to execution, we know that its value after execution will always be *negative-or-zero* ($\ell' = \texttt{-/0}$). In order for a dataflow analysis to be well-defined, all transfer functions have to obey a *monotonicity* property [8].



(a) CFG

$$T\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} \bot \\ f_{\texttt{x=0}}(a) \\ b \sqcup d \\ f_{\texttt{x++}}(c) \end{pmatrix}$$

(b) Whole-program transfer function, T.

	$T^0(\bot)$	$T^1(\bot)$	$T^2(\bot)$	$T^3(\bot)$	$T^4(\bot)$	$T^5(\bot)$
a	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
b	$\bot$	0	0	0	0	0
c	$\bot$	$\bot$	0	0	0/+	0/+
d	$\bot$	$\bot$	$\bot$	+	+	+

(c) Fixed-point iteration.

Fig. 5. Combining CFG, lattice, and transfer functions to perform dataflow analysis (as a fixed-point iteration).

Analysis: Figure 5 shows how to combine the control-flow graph, lattice, and transfer functions to perform dataflow analysis on a tiny example program. First (cf. Figure 5(a)), a control-flow graph is built from the program and annotated with *program points* (which are the *entry* and *exit* points of the statement nodes). In our example, there are four such program points which we label with the letters a to d. Second (cf. Figure 5(b)), the annotated CFG is turned into a *whole-program transfer function*, $T : \mathcal{L}^4 \to \mathcal{L}^4$, which works on four copies of the lattice, $\mathcal{L}$, since we have four program points (a to d). The entry point, a, is assigned an initialization value which depends on the analysis (here, $a = \bot$). For each program point, we simulate the effect of the program using transfer functions (e.g., $b = f_{\texttt{x=0}}(a)$) and the least-upper bound operator for combining flows (e.g., $c = b \sqcup d$). Third (cf. Figure 5(c)), we use the Fixed-Point Theorem [8] to compute the *least* fixed-point of the function, T, by computing $T^i(\bot)$ for increasing values of i (depicted in the columns of the figure), until nothing changes. (Similarly, we can calculate the *greatest* fixed-point by computing $T^i(\top)$ instead of $T^i(\bot)$ for increasing values of i, but that would, in general, result in less precise analysis

results.) As seen in Figure 5(c), we reach a fixed point in five iterations (since $T^4(\bot) = T^5(\bot)$) and hence, the least fixed-point and result of the analysis, is: $a = \bot, b = \texttt{0}, c = \texttt{0/+}, d = \texttt{+}$ (which is the unique least fixed-point of T). From this we can deduce that the value of the variable x is always *zero* at program point b, it is *zero-or-positive* at point c, and *positive* at point d. (Note that, in practice, the fixed-point computation is often performed using more efficient iteration strategies.)

5 Dataflow Analyses for SPLs

In Section 2 we claimed that analyzing SPLs is important and that the naive brute force approach can be costly. In this section, we show how to take any feature-oblivious intraprocedural dataflow analysis and automatically turn it into a feature-sensitive analysis.

We present four different ways of performing dataflow analysis for software product lines (summarized in Figure 8). The four analyses calculate the same information, but in qualitatively different ways. To illustrate the principles, we use a deliberately simple example analysis; *sign analysis* of one variable, x, and use it to analyze an intentionally simple method, `m` (cf. Figure 6(a)) that increases and decreases a variable, depending on the features enabled.

```
void m() {
  int x = 0;
  ifdef (A) x++;
  ifdef (B) x--;
}
```

(a) Example SPL method

c={A} :	c'={B} :	c''={A, B} :
`int x=0;`	`int x=0;`	`int x=0;`
`x++;`	`x--;`	`x++;`
		`x--;`

(b) and its three distinct method variants (configurations: {A}, {B}, and {A, B}).

Fig. 6. A tiny example of an SPL method along with its three distinct method variants.

The program uses features $\mathbb{F} = \{A, B\}$ and we assume it has a feature model $\psi_{\text{FM}} = A \vee B$ which translates into the following set of valid configurations: $[\![\psi_{\text{FM}}]\!] = \{\{A\}, \{B\}, \{A, B\}\}$.

5.1 $\mathcal{A}$0: Brute Force Analysis (Feature Oblivious)

A software product line may be analyzed intraprocedurally by building *all* possible methods and analyzing them one by one using a conventional dataflow analysis as described in the previous section. A method with n features will give rise to 2^n possible end-product methods (minus those invalidated by the feature model). For our tiny example program that has two features, A and B (and feature model $\psi_{\text{FM}} = A \vee B$), we have to build and analyze the *three* distinct methods as illustrated in Figure 6(b). Figure 7(a) depicts the result of analyzing

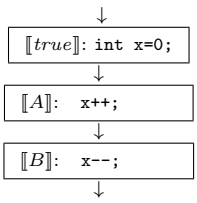
each of these three products using the simple sign-of-x analysis. (Of course, this is a very naive approach; a slightly smarter analysis would avoid re-parsing by building some intermediate representation of the program family.)

5.2 $\mathcal{A}1$: Consecutive Analysis

We can avoid explicitly building all methods individually by making a dataflow analysis *feature-sensitive*. Now, we show how to take any single-program dataflow analysis and automatically turn it into a feature-sensitive analysis, capable of analyzing all possible method variants. Firstly, we consider the consecutive analysis, named this way because we analyze each of the possible configurations one at a time. We render it feature-sensitive by instrumenting the CFG with sufficient information for the transfer functions to know whether a given statement is to be executed or not in each configuration. This analysis will introduce variability into the CFG and overall fixed-point iteration.

Control-Flow Graph: For each node in the CFG, we associate the *set of configurations*, $[\![\phi]\!]$, for which the node's corresponding statement is executed. We refer to this process as *CFG instrumentation*. (In related work, the idea of annotating intermediary representation with configuration information is in known as *configuration lifting* [20] or *variability encoding* [21], although in those cases it is the ASTs that are annotated rather than the CFGs.)

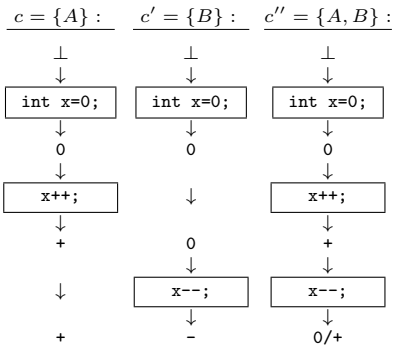
Here is the instrumented CFG for our tiny method of Figure 6(a):



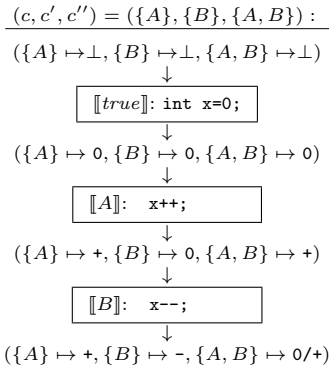
We label each node with "$[\![\phi]\!]$: S" where S is the statement and $[\![\phi]\!]$ is the configuration set associated with the statement. (This is similar to *presence conditions* [22].) Unconditionally executed statements (e.g., `int x=0;`) are associated with the set of all configurations, $[\![true]\!]$. Statements that are nested inside several `ifdef`s will have the intersection of the configuration sets. For instance, statement S in "`ifdef (`ϕ_1`) ifdef (`ϕ_2`)` S" will be associated with the set of configurations $[\![\phi_1]\!] \cap [\![\phi_2]\!] \equiv [\![\phi_1 \wedge \phi_2]\!]$.

Lattice: Analyzing the configurations consecutively does not change the lattice, so the lattice of this feature-sensitive analysis is the same as that of the feature-oblivious analysis.

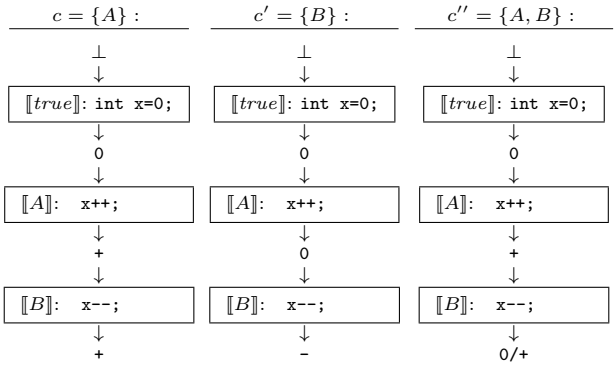
Transfer Functions: All we have to do in the feature-sensitive transfer function is use the associated configuration set, $[\![\phi]\!]$, to figure out whether or not to apply the basic transfer function, f_S, in a given configuration, c; i.e., deciding $c \in [\![\phi]\!]$ (cf. Figure 8). This membership test can be efficiently decided by using Binary Decision Diagrams [23] (BDDs) or *bit vectors* for representing feature formulae.



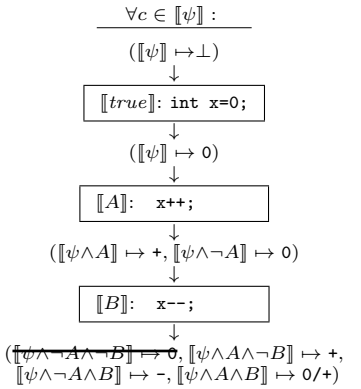
(a) $\mathcal{A}0$: Feature-oblivious analysis.



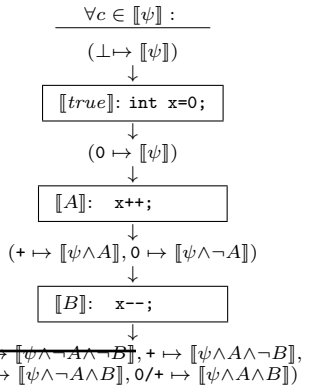
(b) $\mathcal{A}2$: Simultaneous analysis.



(c) $\mathcal{A}1$: Consecutive analysis.



(d) $\mathcal{A}3$: Sim. analysis with sharing.



(e) $\mathcal{A}4$: Sim. with sharing and merging.

Fig. 7. Results of the five analyses on our tiny example method `m` with feature model $\psi = A \vee B$. (Since $\psi \wedge \neg A \wedge \neg B \equiv \mathit{false}$, the two parts in strike out font in sub-figures (d) and (e) will be eliminated by the normalization process (cf. Section 5.4 and 5.5).)

Since the lifting only either applies the basic transfer function or copies the lattice value, the lifted transfer function is also always monotone.

Analysis: In order to analyze a program using $\mathcal{A}1$, all we need to do is to combine the CFG, lattice, and transfer functions as explained in Section 4. Figure 7(c) shows the result of analyzing the increase-decrease method using this consecutive feature-sensitive analysis. As can be seen, the consecutive feature-sensitive analysis needs one fixed-point computation for each configuration. $\mathcal{A}0$ and $\mathcal{A}1$ compute the same information (the same fixed-point solution); the only difference is whether the applicability of statements, $c \in [\![\phi]\!]$, is evaluated before or after compilation.

5.3 $\mathcal{A}2$: Simultaneous Analysis

Another approach is to analyze all configurations *simultaneously* by using a *lifted* lattice that maintains one lattice element per valid configuration. As opposed to the consecutive analysis, the simultaneous analysis needs only *one* fixed-point computation. This analysis will thus further introduce variability into the lattice and the transfer functions in that they will now work on not one configuration, but a sequence of configurations. Again, this analysis will be *feature-sensitive* and it can also be automatically derived from the feature-oblivious analysis.

Control-Flow Graph: The CFG of $\mathcal{A}2$ is the same as that of $\mathcal{A}1$ as it already includes the necessary information for deciding whether or not to simulate execution of a conditional statement.

Lattice: As explained, we lift the basic lattice, $\mathcal{L}$, such that it has one element per valid configuration:

$$\mathcal{L}_2 \quad = \quad [\![\psi_{\text{FM}}]\!] \to \mathcal{L}$$

Note that whenever $\mathcal{L}$ is a lattice, then so is $[\![\psi_{\text{FM}}]\!] \to \mathcal{L}$ (which is isomorphic to $\mathcal{L}^{|[\![\psi_{\text{FM}}]\!]|}$). An example element of this lattice is:

$$L_2 = (\{A\} \mapsto \texttt{+}, \{B\} \mapsto \texttt{-}, \{A, B\} \mapsto \texttt{0/+}) \quad \in \quad [\![\psi_{\text{FM}}]\!] \to \mathcal{L}$$

which corresponds to the information that: for configuration $\{A\}$, we know that the value of x is *positive* ($L_2(\{A\}) = \texttt{+}$); for $\{B\}$, we know x is *negative* ($L_2(\{B\}) = \texttt{-}$); and for $\{A, B\}$, we know it is *zero-or-positive* ($L_2(\{A, B\}) = \texttt{0/+}$).

Transfer Functions: We *lift* the transfer functions correspondingly so they work on elements of the lifted lattice in a point-wise manner. The basic transfer functions are applied only on the configurations for which the statement is executed. As an example, consider the statement "`ifdef (A) x++;`" where the effect of the lifted transfer function on the lattice element $L_2 = (\{A\} \mapsto \texttt{0}, \{B\} \mapsto \texttt{0}, \{A, B\} \mapsto \texttt{0})$ is:

$$L_2 = (\{A\} \mapsto \texttt{0}, \{B\} \mapsto \texttt{0}, \{A, B\} \mapsto \texttt{0})$$
$$\downarrow$$
$$\boxed{[\![A]\!]\text{: } \texttt{x++;}}$$
$$\downarrow$$
$$L_2' = (\{A\} \mapsto \texttt{+}, \{B\} \mapsto \texttt{0}, \{A, B\} \mapsto \texttt{+})$$

The basic transfer function is applied to each of the configurations for which the `ifdef` formula A is satisfied. Since $[\![A]\!] = \{\{A\}, \{A, B\}\}$, this means that the function is applied to the lattice values of the configurations $\{A\}$ and $\{A, B\}$ with resulting value: $f_{\texttt{x++}}(\texttt{0}) = \texttt{+}$. The configuration $\{B\}$, on the other hand, does *not* satisfy the formula ($\{B\} \notin [\![A]\!]$), so its value is left unchanged with value, $L'_2(\{B\}) = \texttt{0}$. Figure 8 depicts and summarizes the effect of the lifted transfer function on the lifted lattice.

Since the transfer function on $[\![\psi_{\text{FM}}]\!] \to \mathcal{L}$ only applies monotone basic transfer functions on $\mathcal{L}$ in a point-wise manner, it is itself monotone. This guarantees the existence of a unique and computable solution.

Analysis: Again, we simply combine the lifted CFG, lifted lattice, and lifted transfer functions to achieve our feature-sensitive simultaneous configuration analysis. Figure 7(b) shows the result of analyzing the increase-decrease method using the simultaneous feature-sensitive analysis. From this we can read off the information about the sign of the variable x at different program points, for each of the valid configurations. For instance, at the end of the program in configuration $\{B\}$, we can see that x is always *negative*. Compared to $\mathcal{A}1$, this analysis only has *one* fixed-point iteration and thus potentially saves the overhead involved. However, it requires the maximum number of fixed-point iterations that are performed in any configuration of $\mathcal{A}1$ in order to reach its fixed-point because of the pointwise lifted lattice. Again, it is fairly obvious that $\mathcal{A}1$ and $\mathcal{A}2$ compute the same information; the only difference being that $\mathcal{A}1$ does one fixed-point iteration per valid configuration whereas $\mathcal{A}2$ computes the same information in one iteration in a point-wise manner.

5.4 $\mathcal{A}3$: Simultaneous Analysis with Sharing

Using the lifted lattice of the simultaneous analysis, it is possible to lazily share lattice values corresponding to configurations that are indistinguishable in the program being analyzed. This analysis will further the notion of variability in the dataflow analyses as it will now be possible to reason about the equivalence of configurations during analysis.

Control-Flow Graph: The CFG of $\mathcal{A}3$ is the same as that of $\mathcal{A}2$.

Lattice: To accomodate the sharing, the lifted lattice of $\mathcal{A}3$ will, instead of mapping configurations to base lattice values, map *sets of configurations* to base lattice values:

$$\mathcal{L}_3 \quad = \quad 2^{[\![\psi_{\text{FM}}]\!]} \hookrightarrow \mathcal{L}$$

This allows $\mathcal{A}3$ lattice values to *share* base lattice values for configurations that have not yet been distinguished by the analysis. For instance, the lifted lattice value of $\mathcal{A}2$, $L_2 = (\{A\} \mapsto \ell, \{B\} \mapsto \ell, \{A, B\} \mapsto \ell) \in \mathcal{L}_2$, can now be represented by $L_3 = ([\![A \vee B]\!] \mapsto \ell) \in \mathcal{L}_3$ where the three configurations, $[\![A \vee B]\!] = \{\{A\}, \{B\}, \{A, B\}\}$, *share* the base lattice value, ℓ.

Transfer Functions: The transfer functions of $\mathcal{A}3$ work by lazily *splitting* sets of configurations, $[\![\psi]\!]$, in two disjoint parts, depending on the feature constraint, ϕ, attached with the statement in question: A set of configurations for which the transfer function *should* be applied, $[\![\psi \wedge \phi]\!]$; and a set of configurations for which the transfer function should *not* be applied, $[\![\psi \wedge \neg\phi]\!]$; i.e.:

$$\begin{array}{c} L_3 = ([\![\psi]\!] \mapsto \ell, ...) \\ \downarrow \\ \boxed{[\![\phi]\!]: \quad S} \\ \downarrow \\ L_3' = ([\![\psi \wedge \phi]\!] \mapsto f_S(\ell), [\![\psi \wedge \neg\phi]\!] \mapsto \ell, ...) \end{array}$$

Note that $[\![\psi \wedge \phi]\!] \cup [\![\psi \wedge \neg\phi]\!] = [\![(\psi \wedge \phi) \vee (\psi \wedge \neg\phi)]\!] = [\![\psi \wedge (\phi \vee \neg\phi)]\!] = [\![\psi \wedge \mathit{true}]\!] = [\![\psi]\!]$ which means that $[\![\psi]\!]$ is split into two parts without losing any configurations. Of course, it might be the case that $[\![\psi]\!]$ is split into "nothing", $\emptyset$, and "everything", $[\![\psi]\!]$ (which happens whenever $\psi \wedge \phi \equiv \mathit{false}$ or $\psi \wedge \neg\phi \equiv \mathit{false}$). We eliminate such *false* constituents in order to ensure a minimal and finite representation. This is taken care of by the function, $\mathit{normalize} : \mathcal{L}_3 \to \mathcal{L}_3$, which is here specified recursively on the structure of $\mathcal{L}_3$ where *nil* is the empty list and the notation "$[\![\phi]\!] \mapsto \ell :: \mathit{tail}$" deconstructs an $\mathcal{L}_3$ value into its first element ($[\![\phi]\!] \mapsto \ell$) and *tail*:

$$\begin{array}{rcl} \mathit{normalize}([\![\psi]\!] \mapsto \ell :: \mathit{tail}) & = & \begin{cases} \mathit{normalize}(\mathit{tail}) & \psi \equiv \mathit{false} \\ [\![\psi]\!] \mapsto \ell :: \mathit{normalize}(\mathit{tail}) & \psi \not\equiv \mathit{false} \end{cases} \\ \mathit{normalize}(\mathit{nil}) & = & \mathit{nil} \end{array}$$

The transfer function of $\mathcal{A}3$ thus has the following behavior on a statement, S, with feature constraint, ϕ:

$$\begin{array}{c} L_3 = ([\![\psi]\!] \mapsto \ell, ...) \\ \downarrow \\ \boxed{[\![\phi]\!]: \quad S} \\ \downarrow \\ L_3' = \mathit{normalize}([\![\psi \wedge \phi]\!] \mapsto f_S(\ell), [\![\psi \wedge \neg\phi]\!] \mapsto \ell, ...) \end{array}$$

Analysis: As always for the analysis, we simply combine the CFG, lattice, and transfer functions to achieve our shared simultaneous analysis. Figure 7(d) shows how this analysis will analyze our tiny program example from earlier. $\mathcal{A}2$ and $\mathcal{A}3$ compute the same information; $\mathcal{A}3$ just represents the same information more compactly using sharing.

5.5 $\mathcal{A}4$: Simultaneous Analysis with Sharing and Merging

In addition to configuration sharing as in $\mathcal{A}3$, it is also possible to *merge* sets of configurations that record the same information (i.e., sets of configurations that map to the same base lattice value) that has earlier been split apart. For

instance, what in $\mathcal{A}3$ might be represented by $L_3 = (\llbracket\phi\rrbracket \mapsto \texttt{+}, \llbracket\psi\rrbracket \mapsto \texttt{+})$ could instead be represented as $L_4 = (\llbracket\phi \vee \psi\rrbracket \mapsto \texttt{+})$, essentially *merging* and thereby also sharing the sets of configurations, $\llbracket\phi\rrbracket$ and $\llbracket\psi\rrbracket$. Since each base lattice value in the co-domain of every $\mathcal{A}4$ lattice value is now unique, it seems more natural to reverse the lattice so as to represent the above information instead as $L_4 = (\texttt{+} \mapsto \llbracket\phi \vee \psi\rrbracket)$ such that base lattice values are partially mapped to sets of configurations instead of the other way around.

Control-Flow Graph: The CFG of $\mathcal{A}4$ is the same as that of the previous feature-sensitive analyses ($\mathcal{A}1$, $\mathcal{A}2$, and $\mathcal{A}3$).

Lattice: As mentioned above, the lattice of $\mathcal{A}4$ now partially maps base lattice values to sets of configurations:

$$\mathcal{L}_4 \quad = \quad \mathcal{L} \hookrightarrow 2^{\llbracket\psi_{\text{FM}}\rrbracket}$$

Transfer Functions: The transfer functions of $\mathcal{A}4$ is easily defined as a transformation of representation, *merge*, from $\mathcal{L}_3$ to $\mathcal{L}_4$, which appropriately merges and disjoins formulae with equal base lattice values; i.e., $\mathit{merge} : \mathcal{L}_3 \rightarrow \mathcal{L}_4$, is defined as $L_4 = \mathit{merge}(L_3)$ where:

$$L_4(\ell) = \begin{cases} \llbracket \bigvee_{L_3(\llbracket\phi\rrbracket)=\ell} \phi \rrbracket & \exists\phi \,.\, L_3(\llbracket\phi\rrbracket) = \ell \\ \text{undefined} & \text{otherwise} \end{cases}$$

A transfer function of $\mathcal{A}4$ for a statement, S, with feature constraint, ϕ, will thus have the following behavior:

$$L_4 = (\ell \mapsto \llbracket\psi\rrbracket, ...)$$
$$\downarrow$$
$$\boxed{\llbracket\phi\rrbracket: \quad S}$$
$$\downarrow$$
$$L_4' = \mathit{merge}(\mathit{normalize}(\llbracket\psi \wedge \phi\rrbracket \mapsto f_S(\ell), \llbracket\psi \wedge \neg\phi\rrbracket \mapsto \ell, ...))$$

Analysis: Again, we simply combine CFG, lattice, and transfer functions to achieve our reversed shared simultaneous analysis. Figure 7(e) illustrates how the analysis analyses our tiny increase-decrease method. Of course, this particular example does not show off the merging capabilities. If we add statement, `x=0`, to the end of our tiny example method, $\mathcal{A}4$ would get the analysis answer: $L_4 = (\texttt{0} \mapsto \llbracket\psi_{\text{FM}}\rrbracket)$ instead of $L_3 = (\{\{A\}\} \mapsto \texttt{0}, \{\{B\}\} \mapsto \texttt{0}, \{\{A, B\}\} \mapsto \texttt{0})$ as in $\mathcal{A}3$. Note that there is a tradeoff between the cost of computing the merging and the benefits of sharing. Whether or not this is worth while is likely to depend on a particular analysis (and SPL) and will have to be tested experimentally (cf. Section 7). In general, it would presumably work better on analyses with small lattices where more information could potentially be shared.

5.6 Other Analysis Approaches

A couple of variations of the feature-sensitive analyses are possible. One could retain the instrumented CFG calculated in $\mathcal{A}1$ and $\mathcal{A}2$, but then *specialize* [24]

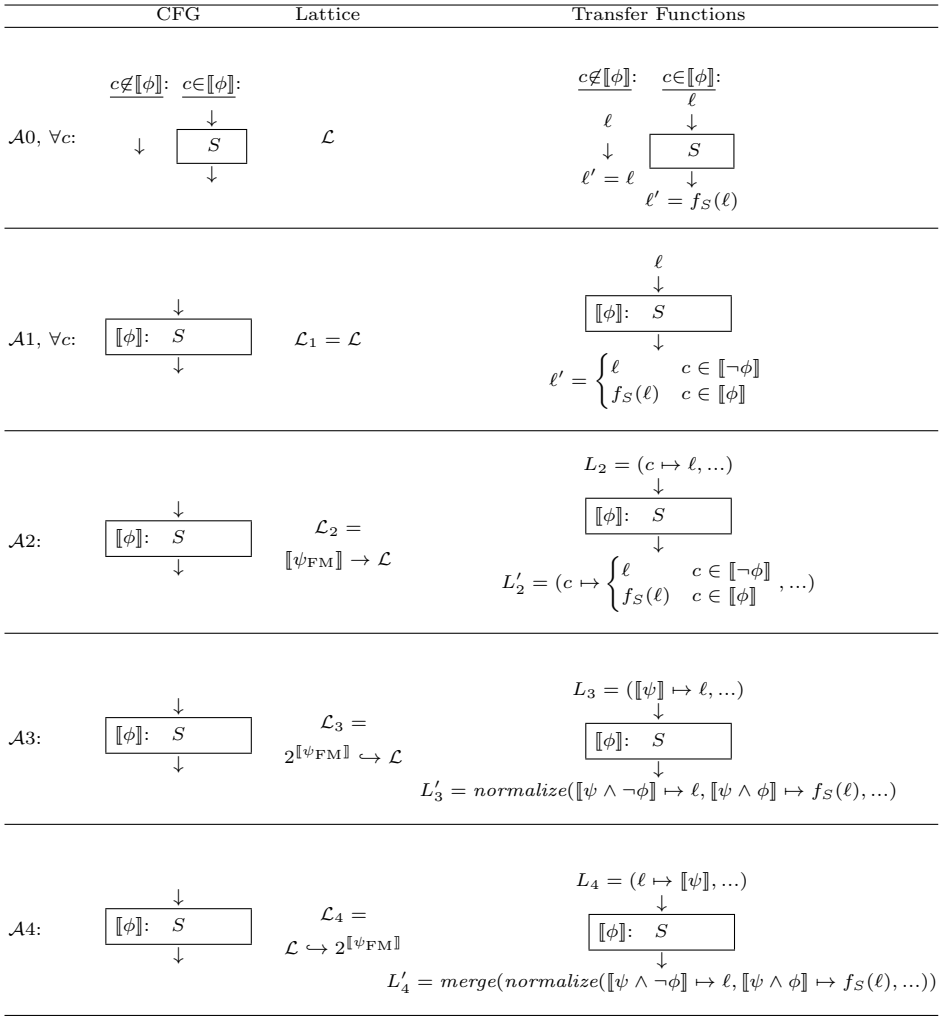


Fig. 8. Summary of dataflow analyses for software product lines.

the CFG prior to analysis for every configuration by resolving all conditional statements relative to the current configuration. This approach would be a variation of $\mathcal{A}1$ with a higher cost due to CFG specialization, but which in turn saves by making membership decisions only once per CFG node. Another approach would be to transform `ifdef`s into normal `if`s and turn feature names into static booleans [20, 21] which could then be resolved by techniques such as *partial evaluation* [24] prior to analysis. We do not explore this idea in the paper, but rather focus on the different ways of automatically transforming a feature-oblivious analysis into a feature-sensitive one, while staying within the framework of dataflow analysis.

6 Analysis of the Analyses

We will now analyze and compare the feature-sensitive analyses. We begin with a proof that all analyses $\mathcal{A}0$ to $\mathcal{A}4$ compute the same information. Then, we will outline the tasks performed by each of the analyses. Finally, we will reason about and compare the asymptotic time and space complexity of the feature-sensitive analyses.

6.1 Equivalence of the Analyses

Lemma 1: The analyses $\mathcal{A}0$, $\mathcal{A}1$, $\mathcal{A}2$, $\mathcal{A}3$, and $\mathcal{A}4$ all compute the exact same information (same analysis result).

Proof: It is sufficient to show that for each configuration and each statement, all transfer functions of $\mathcal{A}0$, $\mathcal{A}1$, $\mathcal{A}2$, $\mathcal{A}3$, and $\mathcal{A}4$ (cf. Figure 8) compute the exact same information (only represented in different ways by the five analyses). Then, the rest follows inductively by transitivity on the entire control-flow graph. Let configuration c and statement S with feature constraint ϕ be given. We assume the input to the transfer function to associate base lattice value ℓ with the configuration c. We will now show that the resulting output of all of the transfer functions for $\mathcal{A}0$ to $\mathcal{A}4$ associate the same (new) output base lattice value, ℓ', for our configuration c. Now, we have two cases, depending on whether or not $c \in [\![\phi]\!]$.

Case $c \in [\![\phi]\!]$:

- $\mathcal{A}0$: Since $c \in [\![\phi]\!]$, we get that: $\ell' = f_S(\ell)$.
- $\mathcal{A}1$: Here, we have that $\ell' = \begin{cases} \ell & c \in [\![\neg\phi]\!] \\ f_S(\ell) & c \in [\![\phi]\!] \end{cases}$ which means that $\ell' = f_S(\ell)$, as required, since $c \in [\![\phi]\!]$.
- $\mathcal{A}2$: By assumption, the input to the transfer function gives us that $L_2(c) = \ell$. Then, the output of the transfer function is $L_2' = (c \mapsto \begin{cases} \ell & c \in [\![\neg\phi]\!] \\ f_S(\ell) & c \in [\![\phi]\!] \end{cases}, ...)$

which means that the information computed for configuration c is $\ell' = L'_2(c) = f_S(\ell)$, as required, since $c \in [\![\phi]\!]$.
- $\mathcal{A}3$: Here, the input to the transfer function is assumed to associate $L_3([\![\psi]\!]) = \ell$, for some ψ where $c \in [\![\psi]\!]$. The output of the transfer function is then $L'_3 = \mathit{normalize}([\![\psi \wedge \phi]\!] \mapsto f_S(\ell), [\![\psi \wedge \neg\phi]\!] \mapsto \ell, ...)$. Since $c \in [\![\psi]\!]$ and $c \in [\![\phi]\!]$, we conclude that $c \in [\![\psi \wedge \phi]\!]$ which in turn means that $\psi \wedge \phi \not\equiv \mathit{false}$ and hence that the information computed is $\ell' = L'_3([\![\psi \wedge \phi]\!]) = f_S(\ell)$. This establishes that $c \in [\![\psi \wedge \phi]\!]$ is mapped to $\ell' = f_S(\ell)$, as required.
- $\mathcal{A}4$: Here, the input to the transfer function is assumed to associate $L_5(\ell) = [\![\psi]\!]$, for some ψ where $c \in [\![\psi]\!]$. In case $\mathcal{A}3$ above, we deduced that $L'_3([\![\psi \wedge \phi]\!]) = f_S(\ell)$ for $L'_3 = \mathit{normalize}([\![\psi \wedge \phi]\!] \mapsto f_S(\ell), [\![\psi \wedge \neg\phi]\!] \mapsto \ell, ...)$ and that $c \in [\![\psi \wedge \phi]\!]$. This then means that for $L'_4 = \mathit{merge}(L'_3)$ we have that $L'_4(f_S(\ell)) \supseteq [\![\psi \wedge \phi]\!] \ni c$. Hence, the information computed for configuration c is $\ell' = f_S(\ell)$, as required.

Case $c \notin [\![\phi]\!]$:

- $\mathcal{A}0$: Since $c \notin [\![\phi]\!]$, we get that: $\ell' = \ell$.
- $\mathcal{A}1$: Since $c \notin [\![\phi]\!]$, we deduce that $c \in [\![\neg\phi]\!]$. Now, by definition, $\ell' = \begin{cases} \ell & c \in [\![\neg\phi]\!] \\ f_S(\ell) & c \in [\![\phi]\!] \end{cases}$, we obtain that $\ell' = \ell$, as required.
- $\mathcal{A}2$: By assumption, the input to the transfer function gives us that $L_2(c) = \ell$. Then, the output of the transfer function is $L'_2 = (c \mapsto \begin{cases} \ell & c \in [\![\neg\phi]\!] \\ f_S(\ell) & c \in [\![\phi]\!] \end{cases}, ...)$ which means that the information computed for configuration c is $\ell' = L'_2(c) = \ell$, as required, since $c \in [\![\neg\phi]\!]$.
- $\mathcal{A}3$: Here, the input to the transfer function is assumed to associate $L_3([\![\psi]\!]) = \ell$, for some ψ where $c \in [\![\psi]\!]$. The output of the transfer function is $L'_3 = \mathit{normalize}([\![\psi \wedge \phi]\!] \mapsto f_S(\ell), [\![\psi \wedge \neg\phi]\!] \mapsto \ell, ...)$. However, since $c \notin [\![\phi]\!]$, we conclude that $c \in [\![\neg\phi]\!]$ and then that $c \in [\![\psi \wedge \neg\phi]\!]$ which in turn means that $[\![\psi \wedge \neg\phi]\!] \not\equiv \mathit{false}$ and hence that $\ell' = L'_3([\![\psi \wedge \neg\phi]\!]) = \ell$. Thus, the information computed for $c \in [\![\psi \wedge \neg\phi]\!]$ is $\ell' = \ell$, as required.
- $\mathcal{A}4$: Here, the input to the transfer function is assumed to associate $L_4(\ell) = [\![\psi]\!]$, for some ψ where $c \in [\![\psi]\!]$. In case $\mathcal{A}3$ above, we deduced that $L'_3([\![\psi \wedge \neg\phi]\!]) = \ell$ for $L'_3 = \mathit{normalize}([\![\psi \wedge \phi]\!] \mapsto f_S(\ell), [\![\psi \wedge \neg\phi]\!] \mapsto \ell, ...)$ and that $c \in [\![\psi \wedge \neg\phi]\!]$. This then means that for $L'_4 = \mathit{merge}(L'_3)$ we have that $L'_4(\ell) \supseteq [\![\psi \wedge \neg\phi]\!] \ni c$ which means the information computed for configuration c is $\ell' = \ell$, as required.

□

6.2 Analysis Tasks

Figure 9 considers the overall tasks performed for each SPL method analyzed in each of the analyses. Analyses $\mathcal{A}0$, $\mathcal{A}1$, and the three simultaneous analyses all

differ substantially in the number of times each of the tasks are performed. Not surprisingly, $\mathcal{A}0$ needs to do a lot of (brute force) compilation and preprocessing. The rest require only one compilation, but pay the price of instrumentation to annotate the CFG with feature constraints. However, this is cheap in practice. $\mathcal{A}1$ performs the analysis (i.e., the fixed-point computation) for every valid configuration whereas $\mathcal{A}2$ to $\mathcal{A}4$ only do this once. We return to these considerations, in practice, when we discuss our experimental results (cf. Section 7).

$$
\begin{aligned}
\text{time}(\mathcal{A}0\,) &= |\psi_{\text{FM}}| \cdot \langle \text{time}(\text{preprocess}) + \text{time}(\text{compile}) + \text{time}(\text{analyze}_{\mathcal{A}0}) \rangle \\
\text{time}(\mathcal{A}1\,) &= \text{time}(\text{compile}) + \text{time}(\text{instrument}) + |\psi_{\text{FM}}| \cdot \langle \text{time}(\text{analyze}_{\mathcal{A}1}) \rangle \\
\text{time}(\mathcal{A}2\,) &= \text{time}(\text{compile}) + \text{time}(\text{instrument}) + \text{time}(\text{analyze}_{\mathcal{A}2}) \\
\text{time}(\mathcal{A}3\,) &= \text{time}(\text{compile}) + \text{time}(\text{instrument}) + \text{time}(\text{analyze}_{\mathcal{A}3}) \\
\text{time}(\mathcal{A}4\,) &= \text{time}(\text{compile}) + \text{time}(\text{instrument}) + \text{time}(\text{analyze}_{\mathcal{A}4})
\end{aligned}
$$

Fig. 9. Overall tasks performed by each of the analyses.

6.3 Asymptotic Time Complexity

The time complexity of the $\mathcal{A}0$ analysis is:

$$\textsc{Time}(\mathcal{A}0\,) = \mathcal{O}(|\psi_{\text{FM}}| \cdot |\mathcal{G}| \cdot T_0 \cdot h(\mathcal{L}))$$

where $|\mathcal{G}|$ is the size of the control-flow graph (which for the intraprocedural analysis is linear in the number of statements in the method analyzed, ignoring exceptions); T_0 is the execution time of a transfer function on the $\mathcal{L}$ lattice; and $h(\mathcal{L})$ is the height of the $\mathcal{L}$ lattice. In total, we need to analyze $|\psi_{\text{FM}}|$ method variants. For each of these, we execute $\mathcal{O}(|\mathcal{G}|)$ different transfer functions, each of which takes execution time, T_0, and can be executed a worst-case maximum of $h(\mathcal{L}_0)$ number of times.

Similarly, the time complexity of the $\mathcal{A}1$ analysis is:

$$\textsc{Time}(\mathcal{A}1\,) = \mathcal{O}(|\psi_{\text{FM}}| \cdot |\mathcal{G}| \cdot T_1 \cdot h(\mathcal{L}_1))$$

Since $\mathcal{L}_1 = \mathcal{L}$, the main difference between the complexity of $\mathcal{A}0$ and $\mathcal{A}1$ is the time it takes to evaluate a transfer function (i.e., T_0 vs. T_1). In $\mathcal{A}1$, the statement applicability condition, $c \in [\![\phi]\!]$, is evaluated for each transfer function. In contrast, in $\mathcal{A}0$, this condition is evaluated at preprocessing time.

Analogously, we can quantify the asymptotic time complexity of $\mathcal{A}2$:

$$\textsc{Time}(\mathcal{A}2\,) = \mathcal{O}(|\mathcal{G}| \cdot T_2 \cdot h(\mathcal{L}_2))$$

which is similar to $\mathcal{A}1$, except that we do not need to analyze $|\psi_{\text{FM}}|$ times and that the numbers are parameterized by the $\mathcal{A}2$ lattice and transfer functions. For the height of the lattice $\mathcal{L}_2$, we have:

$$h(\mathcal{L}_2) \;=\; h([\![\psi_{\text{FM}}]\!] \to \mathcal{L}_1) \;=\; h(\mathcal{L}_1^{|\psi_{\text{FM}}|}) \;=\; \sum_{c \in [\![\psi_{\text{FM}}]\!]} h(\mathcal{L}_1) \;=\; |\psi_{\text{FM}}| \cdot h(\mathcal{L}_1)$$

Note, however, that this is a purely theoretically worst case that does not naturally arise in practice because of the pointwise nature of $\mathcal{A}2$. Since all configurations are independent, the penalty for $\mathcal{A}2$ will not be the *sum*, but rather only the *maximum* number of fixed-point iterations of $\mathcal{A}1$. In practice, we have not observed any significant cost on behalf of $\mathcal{A}2$ from this effect, as we will see in Section 7. The remaining speed factor between $\mathcal{A}1$ and $\mathcal{A}2$ thus boils down to:

$$\mathcal{A}1 : \mathcal{A}2 \quad = \quad |\psi_{\text{FM}}| \cdot T_1 \;:\; T_2$$

In theory, we would not expect any difference in the speed of the two analyses; $\mathcal{A}1$ makes a sequence of n analyses and $\mathcal{A}2$ makes one analysis in which each step costs n. However, as we will see in Section 7, $\mathcal{A}2$ has better cache performance than $\mathcal{A}1$, since statement nodes only have to be retrieved and evaluated once per transfer function in $\mathcal{A}2$, instead of once per configuration as in $\mathcal{A}1$. Apart from data, also the fixed-point iteration code only runs once instead of once per configuration.

The speed of the analyses $\mathcal{A}3$ and $\mathcal{A}4$ depends on the sharing potential which is intimately dependent on a particular SPL. In Section 7, we will see how $\mathcal{A}3$ does on our benchmarks.

6.4 Asymptotic Space Complexity

In terms of memory consumption, the asymptotic space complexity of the analyses is simply proportional to the amount of data occupied by the lattice values. For the analyses $\mathcal{A}1$ and $\mathcal{A}2$ this becomes:

$$\textsc{Space}(\mathcal{A}1\,) = \mathcal{O}(|\mathcal{G}| \cdot log(|\mathcal{L}_1|))$$
$$\textsc{Space}(\mathcal{A}2\,) = \mathcal{O}(|\mathcal{G}| \cdot log(|\mathcal{L}_2|))$$

Comparing the two, we can derive that:

$$log(|\mathcal{L}_2|) = log(|[\![\psi_{\text{FM}}]\!] \to \mathcal{L}_1|) = log(|\mathcal{L}_1^{|\psi_{\text{FM}}|}|) = log(|\mathcal{L}_1|^{|\psi_{\text{FM}}|}) = |\psi_{\text{FM}}| \cdot log(|\mathcal{L}_1|)$$

which thus means that the difference is:

$$\textsc{Space}(\mathcal{A}2\,) = |\psi_{\text{FM}}| \cdot \textsc{Space}(\mathcal{A}1\,)$$

This relationship is also evident when comparing Figures 7(c) and 7(b). Although $\mathcal{A}2$ requires $n = |\psi_{\text{FM}}|$ times more memory to run, it is always possible to cut the $\mathcal{A}2$ lattice into k *slices* of n/k columns (i.e., analyze n/k number of configurations at a time). This provides a way to in some sense combine the time and space characteristics of $\mathcal{A}1$ with $\mathcal{A}2$ to $\mathcal{A}4$.

As for performance, the memory consumption of the analyses $\mathcal{A}3$ and $\mathcal{A}4$ depends on the sharing potential which will vary from SPL to SPL. Again, we will see how $\mathcal{A}3$ does in practice in Section 7.

7 Evaluation

We first present our study settings. Then, we present our results in terms of total analysis time (incl. compilation) and analysis-time only (excl. compilation). We discuss the results and use the knowledge gained from the experiments to derive an interprocedurally combined analysis which is faster than each of our individual analyses, on all benchmarks. Finally, we look briefly at cache implications and memory consumption of our analyses.

7.1 Study settings

To validate the ideas, we have implemented and evaluated the performance and memory characteristics of an intraprocedural version of the ubiquitous *reaching definitions* dataflow analysis which is implemented using `SOOT`'s interprocedural dataflow analysis framework for analyzing Java programs [25].

We have subsequently lifted the analysis into the four feature-sensitive analyses for SPLs ($\mathcal{A}1$ to $\mathcal{A}4$). The sharing in the $\mathcal{A}3$ and $\mathcal{A}4$ analyses is represented by *bit vectors* from the `Colt` high performance open source Java libraries [26]. Note, however, that since the current implementation of our $\mathcal{A}4$ analysis uses a sub-optimal representation which slows it down unfairly, we have excluded it from the experiments below. Since we are using *intra*procedural analyses which analyze one method at a time, we use the *local* set of features of each method, $\mathbb{F}_{local}$, instead of $\mathbb{F}$ which significantly reduces the size of the lattices we work with. (Recall that the number of potential configurations is exponential in the number of features.)

Benchmark	LOC	$\lvert\mathbb{F}\rvert$	$\lvert 2^{\mathbb{F}_{local}}\rvert$	max. $\lvert\psi_{\texttt{FM}}\rvert$	avg. $\lvert\psi_{\texttt{FM}}\rvert$	#methods
`Graph PL`	1,350	18	512 (2^9)	106	3.91	135 (528)
`MM08`	5,700	14	128 (2^7)	24	1.84	285 (523)
`Prevayler`	8,000	5	8 (2^3)	8	1.28	779 (1,001)
`Lampiro`	45,000	11	4 (2^2)	4	1.01	1,944 (1,971)
`BerkeleyDB`	84,000	42	128 (2^7)	40	1.64	3,604 (5,905)

Fig. 10. Size metrics for our five SPL benchmarks.

We have chosen five qualitatively different SPL benchmarks, listed in Figure 10. The table summarizes: `LOC`, as the number of lines of code; $\lvert\mathbb{F}\rvert$, as the number of features in the SPL; $\lvert 2^{\mathbb{F}_{local}}\rvert$, as the maximum number of potential configurations of any one method in the SPL, given $\lvert\mathbb{F}_{local}\rvert$ features; `max.` $\lvert\psi_{\texttt{FM}}\rvert$, as the maximum number of *valid* configurations in a method; `avg.` $\lvert\psi_{\texttt{FM}}\rvert$, as the average number of valid configurations in a method; and `#methods`, as the number of methods (with the total number of distinct methods, including variations of the same method, in parentheses). `Graph PL` (aka., `GPL`) is a small product line for desktop applications [3]; it has a bit more than a thousand lines of code with intensive feature usage. `MobileMedia08` (aka., `MM08`) is a product line for

mobile applications for dealing with multimedia [27]; it is slightly larger than `Graph PL` and has moderate feature usage. `Prevayler` is an in-memory database for Java applications [3] which was not developed as an SPL but has been turned into an SPL as part of a research project; it is a bit larger than `MM08` and has low feature usage. `Lampiro` is a product line for instant-messaging clients [18]; it is more moderately sized and also has very low feature usage. Last but not least, `BerkeleyDB` is a product line for databases [3]; it has 84K lines of code and moderate feature usage.

The histograms in Figure 11 illustrate the distribution of the number of distinct valid methods per number of features in a method for each of the SPLs. `Graph PL` (depicted in Figure 11(a)), for instance, has 27 methods without features each of which need to be analyzed in $2^0 = 1$ configuration (i.e., $27 * 1 = 27$ distinct valid methods to analyze). It has 75 methods with one feature that each need to be analyzed in $2^1 = 2$ valid configurations (i.e., $75 * 2 = 150$ distinct valid methods to analyze). Finally, e.g., it has *one* method with 9 features which amounts to $2^9 = 512$ potential configurations of which 106 are valid (i.e., $1 * 106 = 106$ distinct valid methods to be analyzed which can be seen as the rightmost bar). Note that when plotted this way, the area shown in the histograms is thus directly proportional to the number of distinct methods to be analyzed.

The leftmost white bar represents methods without feature usage and thus trivially only one configuration. On such methods, we expect the simultaneous analyses $\mathcal{A}2$ to $\mathcal{A}4$ to suffer an overhead penalty compared to $\mathcal{A}1$ from having to represent a single base lattice value as either: in $\mathcal{A}2$, as a *list of one value*; or in $\mathcal{A}3$ and $\mathcal{A}4$, as a *set of one value* trivially without prospects of sharing. We have thus plotted those bars in a different color (white) to emphasize the difference while the rest of the method bars are plotted in gray.

As can be seen, the five benchmark SPLs have qualitatively different feature usage profiles. The histogram of `Graph PL` is dominated by one method, `Vertex.display()`, with 106 valid configurations. In `MobileMedia08`, there is one method, `MediaListScreen.initMenu()`, with 24 valid configurations and one, `handleCommand()` in the class `MediaController`, with 20. `Prevayler` has one method, `PrevaylerFactory.publisher()`, with $2^3 = 8$ valid configurations and two methods with $2^2 = 4$ valid configurations. `Lampiro` only has one method, `RegisterScreen.xmppLogin()`, with $2^2 = 4$ valid configurations and 24 methods with $2^1 = 2$ valid configurations. In `BerkeleyDB`, the highest number of valid configurations, 40, is achieved by `DbRunAction.main()`.

Our analyses currently interface with `CIDE` (Colored IDE) [3] for retrieving the conditional compilation statements. `CIDE` is a tool that enables developers to annotate feature code using background colors rather than `ifdef` directives, reducing code pollution and improving comprehensibility. Conceptually, `CIDE` uses a restricted form of `ifdef`s for which only conjunction of features is permitted.

We have executed the analyses on a 64-bit machine with an Intel® Core™ i7 3820 CPU running at 3.6GHz with 32GB of memory (16GB allocated to the

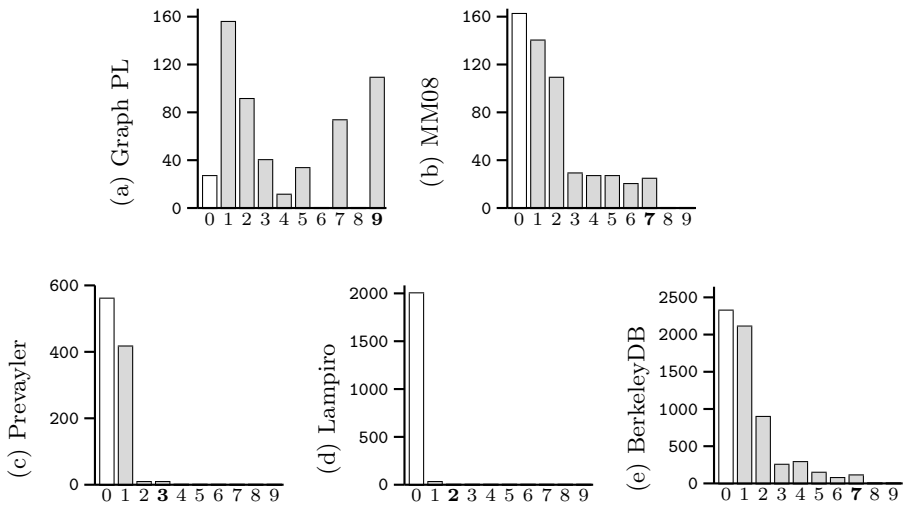


Fig. 11. Histogram showing the distribution of number of distinct valid methods per numbers of features in a method. The x-axis plots the number of features in a method (with the maximum such in bold face). The y-axis plots the number of distinct valid methods; e.g., `Graph PL` has *one* method with 106 valid (out of $2^9 = 512$ potential) configurations which alone account for the entire rightmost bar in Fig. 11(a).

JVM for the experiment) and 10MB of level three cache. The operating system is Linux Mint 13 with kernel 3.2.0-23-generic.

7.2 Results and Discussion of $\mathcal{A}0$ vs. $\mathcal{A}1$ vs. $\mathcal{A}2$ vs. $\mathcal{A}3$

We now present the results[3] obtained from our empirical study. We first present and discuss our results pertaining to the sum of analyzing all configurations for each SPL. First, we look at *total time* (incl. compilation), then at the *analysis-time only* (excl. compilation). Hereafter, we will investigate the analyses more closely and use our insights to combine the analyses so as to obtain an even faster combined analysis. Finally, we look at caching and memory consumption characteristics of our analyses. All times are given as the sum (for all methods) of the median of ten runs.

Total Time: Figure 12 plots the total time (including compilation) of the reaching definitions analysis on each of our five benchmarks where we use the feature-oblivious brute-force analysis, $\mathcal{A}0$, as a baseline. For $\mathcal{A}0$, the total time is shown in black whereas the feature-sensitive analyses, $\mathcal{A}1$ to $\mathcal{A}3$ are plotted in dark gray, light gray, and white, respectively. The percentages below the

[3] Results are available at: `http://twiki.cin.ufpe.br/twiki/bin/view/SPG/EmergentAndDFA`

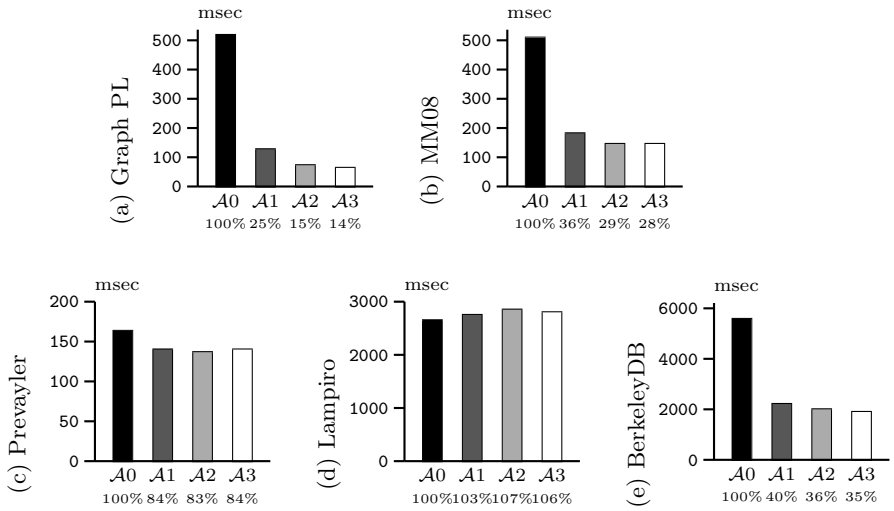


Fig. 12. Total time: $\mathcal{A}0$ (black) vs. $\mathcal{A}1$ (dark gray) vs. $\mathcal{A}2$ (light gray) vs. $\mathcal{A}3$ (white).

histograms are given in relation to the speed of the baseline analysis, $\mathcal{A}0$. All times comprise the tasks outlined in Figure 9 (except preprocessing in the case of $\mathcal{A}0$). For each method, the time computed for $\mathcal{A}0$ is the average of the *slowest* and *fastest* configuration multiplied with the number of valid configurations. We have to do this estimation because several configurations, although valid according to the feature model, generate code that does not compile. Also, since the `CIDE` API does not currently provide an efficient way of getting the color of a line, we omit the time of this calculation from the CFG instrumentation time.

We see that for `Lampiro` in which most methods are normal featureless ones, the analyses are, as expected, all fairly equal in speed. This is because most of the analysis effort is spent on conventional featureless methods (cf. the white bar of Figure 11). For `Prevayler` which has low feature usage, the feature-sensitive approaches are all fairly equal in speed; around 15% faster than $\mathcal{A}0$. The feature-sensitive analyses all save time on its many two-configuration methods which they, unlike $\mathcal{A}0$, do not have to compile in two variants. On the SPLs with higher feature usage (`MobileMedia08`, `BerkeleyDB`, and especially `Graph PL`), the analyses $\mathcal{A}1$ to $\mathcal{A}3$ are all substantially faster than $\mathcal{A}0$ as they do not have to compile methods with many configurations in all possible variants.

We see that the *gain factor* of $\mathcal{A}3$ relative to $\mathcal{A}0$ on each SPL is: 7.3 for `Graph PL`; 3.5 for `MobileMedia08`; 2.9 for `BerkeleyDB`; 1.2 for `Prevayler`; and, not surprisingly, close to one (0.94) for `Lampiro`. So for SPLs with high feature usage, we have a speed up of about seven, about three on SPLs with moderate feature usage, and around one for SPLs with low feature usage.

The reason for the drastic speed-up is *compilation overhead* in that $\mathcal{A}0$ has to compile each method in every single valid configuration (see Figure 9). On

the other hand, $\mathcal{A}1$ to $\mathcal{A}3$ only need to compile each method *once* to obtain the instrumented CFG on which the analyses operate, even if multiple analyses are subsequently performed. In the following, we will thus focus on the times of the analyses themselves; i.e., without compilation and instrumentation.

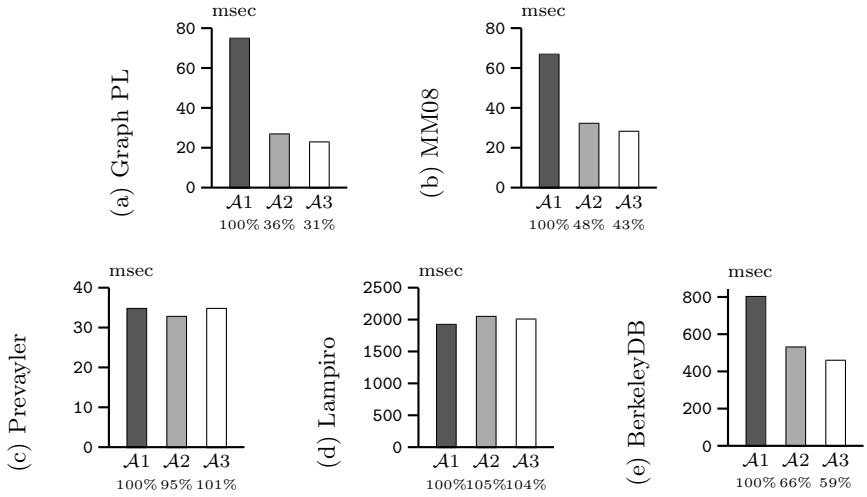


Fig. 13. Analysis-only time: $\mathcal{A}1$ (dark gray) vs. $\mathcal{A}2$ (light gray) vs. $\mathcal{A}3$ (white).

Analysis-Time Only: Figure 13 plots the analysis-time only of running the reaching definitions analysis on all configurations of all methods (i.e., excluding compilation). Obviously, we retain the same pattern for the feature-sensitive analyses as in Figure 12, but we now plot $\mathcal{A}1$ as a baseline making it easier to compare the analyses $\mathcal{A}1$ to $\mathcal{A}3$. For `Lampiro` and `Prevayler` with low feature usage, we see little difference between the speed of the analyses. For the SPLs with higher feature usage (`MobileMedia08`, `BerkeleyDB`, and `Graph PL`), we see that $\mathcal{A}2$ is between a third and two thirds faster than $\mathcal{A}1$; and we see that $\mathcal{A}3$ is slightly faster than $\mathcal{A}2$.

As we shall see, the speed difference between the analyses will increase with the number of valid configurations of a method. $\mathcal{A}2$ is faster than $\mathcal{A}1$ on many-configuration methods because it has better cache properties; and, $\mathcal{A}3$ is faster than $\mathcal{A}2$ on such methods because it can share information between its configurations and hence avoid computations.

In fact, if we look at the method with the highest number of configurations in our benchmarks (`Vertex.display()` in `Graph PL` which has 106 valid configurations and accounts for 63% of the $\mathcal{A}1$ analysis time on that entire SPL), $\mathcal{A}2$ is five times faster than $\mathcal{A}1$. $\mathcal{A}3$ is in turn three times faster than $\mathcal{A}2$ and thereby 15 times faster than $\mathcal{A}1$. This method has many equivalent configurations that

are indistinguishable by `#ifdef`s (i.e., for which the same `#ifdef` blocks apply). The sharing can thereby avoid many redundant basic transfer function computations. Here, $\mathcal{A}1$ would have to execute the fixed-point algorithm 106 times, feching and re-feching the same data for each statement over and over.

Recall that $\mathcal{A}2$ in principle has to do as many fixed-point iterations as are needed for the *slowest* converging configuration, unnecessarily reiterating already converged configurations. Our data, however, indicates that this is not a problem in practice. In the analysis, $\mathcal{A}2$ only executes as little as 0.21% more basic transfer functions than $\mathcal{A}1$ on `BerkeleyDB`; only 0.06% more on `Graph PL`; and virtually 0% more on `Lampiro`, `MobileMedia08`, and `Prevayler`.

Beyond the Sum of All Methods: So far, the discussion has focused on the sum of the time for analyzing *all* methods in each given SPL. By looking at the data for each method in isolation, we can see that the feature sensitive approaches behave differently for different kinds of methods. A number of factors have an effect on the performance of our approaches; including: *number of configurations*, *number of statements*, and the *number of assignments* (relevant for reaching definitions).

Knowledge on which analyses are good under what conditions may allow us to use heuristics to combine the analyses to obtain a combined analysis strategy which is even faster than any of $\mathcal{A}1$ to $\mathcal{A}3$, individually. We will now investigate one factor, namely the number of configurations as a predictor of which analysis is fastest and see if the differences between $\mathcal{A}1$ to $\mathcal{A}3$ are statistically significant. In the following, we deliberately *exclude* a benchmark, `Prevayler`, from our studies so that we may subsequently use it to evaluate our heuristics on a fresh SPL in an unbiassed manner. We decided to do this to increase confidence that the lessons learned and heuristics constructed are not merely coincidental on our data set and will be applicable to other SPLs as well. When we look at which analyses are best on which methods, interesting information emerges.

$\lvert\psi_{\mathtt{FM}}\rvert$	N	$\mathcal{A}1$:$\mathcal{A}1$	$\mathcal{A}1$:$\mathcal{A}2$	$\mathcal{A}1$:$\mathcal{A}3$	Fastest
106	1	1.00	5.15	**15.1**	$\mathcal{A}3$
[20..72]	11	1.00	4.17	**10.5**	
[12..18]	24	1.00	3.36	**6.19**	
8	27	1.00	2.97	**3.81**	
6	19	1.00	2.49	**3.11**	
5	4	1.00	2.11	**2.55**	
4	212	1.00	**1.84**	1.83	$\mathcal{A}2$
3	60	1.00	**1.67**	1.54	
2	1,158	1.00	**1.30**	1.12	
1	4,452	**1.00**	0.83	0.75	$\mathcal{A}1$

Fig. 14. Fastest analysis of $\mathcal{A}1$ vs. $\mathcal{A}2$ vs. $\mathcal{A}3$ according to number of configurations (average speed ratios in relation to $\mathcal{A}1$, for methods of all SPLs, excluding `Prevayler`).

Figure 14 compares the speed of $\mathcal{A}1$ vs. $\mathcal{A}2$ vs. $\mathcal{A}3$ on all methods of the five SPLs according to varying number of valid configurations. $|\psi_{\texttt{FM}}|$ is the number of valid configurations and N is the number of methods in all SPLs with that particular configuration count. The next three columns give the average speed ratio for each analysis in relation to $\mathcal{A}1$ and the final column declares the fastest analysis as the winner (the analysis with the highest ratio, shown in bold face).

For methods with only one configuration, $\mathcal{A}2$ and $\mathcal{A}3$, as expected, incur an overhead penalty, compared to $\mathcal{A}1$. For methods with two to four configurations, we see that $\mathcal{A}2$ is the fastest analysis (although there is practically no difference on four configurations). On methods with more than four configurations, $\mathcal{A}3$ seems to be the fastest. Also, we see a clear tendency for $\mathcal{A}2$, and especially $\mathcal{A}3$, to accelerate relative to $\mathcal{A}1$ as the number of configurations increase. We also note that the relationship between the ratios and the number of configurations is non-linear, so we cannot mix data across number of configurations when testing for statistical significance.

Not surprisingly, $\mathcal{A}1$ is best when considering all methods in all SPLs (excluding `Prevayler`) with *one* configuration; it outperforms $\mathcal{A}2$ and $\mathcal{A}3$ on 94% and 96% of the methods, respectively. However, when we consider methods with *more than one* configuration, $\mathcal{A}2$ and $\mathcal{A}3$ perform better than $\mathcal{A}1$ on 94% and 72% of the methods, respectively.

To further support our claim that $\mathcal{A}1$ performs differently than $\mathcal{A}2$, respectively, $\mathcal{A}3$, we execute the `Wilcoxon` paired difference test. We use this statistical test rather than a `t-test` because our data is not normally distributed. If we subscribe to the common convention of using a 95% confidence interval ($\alpha = 0.05$), we are able to reject the *null hypothesis* that $\mathcal{A}1 = \mathcal{A}2$, respectively, $\mathcal{A}1 = \mathcal{A}3$ when comparing the data for methods with one configuration. Similarly, we get statistical significant differences when comparing $\mathcal{A}1$ vs. $\mathcal{A}2$ and in turn $\mathcal{A}3$ on methods with two configurations. In conclusion, $\mathcal{A}1$ is statistically significantly better on one-configuration methods than $\mathcal{A}2$ and $\mathcal{A}3$ and worse on the other methods.

When repeating this experiment on $\mathcal{A}2$ vs. $\mathcal{A}3$, we are able to ascertain that the speed of two analyses are statistically significantly different on all groups of methods of n configurations, for all values of n, except $n = 4$, where we get a `p-value` of $0.45 > \alpha$.

This is also backed up by empirical evidence that $\mathcal{A}2$ perform better than $\mathcal{A}3$ on 76% of methods with *less* than four configurations. Conversely, $\mathcal{A}3$ is better than $\mathcal{A}2$ on 93% of the methods with *more* than four configurations. Finally, and perhaps most interestingly, on the 212 methods with *exactly four* configurations, $\mathcal{A}2$ wins on 105 methods while $\mathcal{A}3$ wins on 107; i.e., we get an almost 50 : 50 probability (we, in fact, get 49.5 : 50.5).

In conclusion, we have demostrated statistically significant difference between our analyses apart from $\mathcal{A}2$ vs. $\mathcal{A}3$ on methods with four configurations where it does not seem to matter which of the analyses we select.

Combining the Analyses: In *intra*procedural analysis, methods are analyzed individually, one at a time, which means that we can easily analyze different methods using different analyses. The following analysis strategy, $\mathcal{A}*$, select the fastest analysis (according to the lessons learned and Figure 14) depending on the number of configurations, n, when analyzing a particular method:

$$\mathcal{A}*(n) = \begin{cases} \mathcal{A}1 & n = 1 \\ \mathcal{A}2 & 1 < n \leq 4 \\ \mathcal{A}3 & 4 < n \end{cases}$$

We somewhat arbitrarily chose to select $\mathcal{A}2$ on methods with four configurations, but we might as well have chosen $\mathcal{A}3$ as the difference is not statistically significant on such methods.

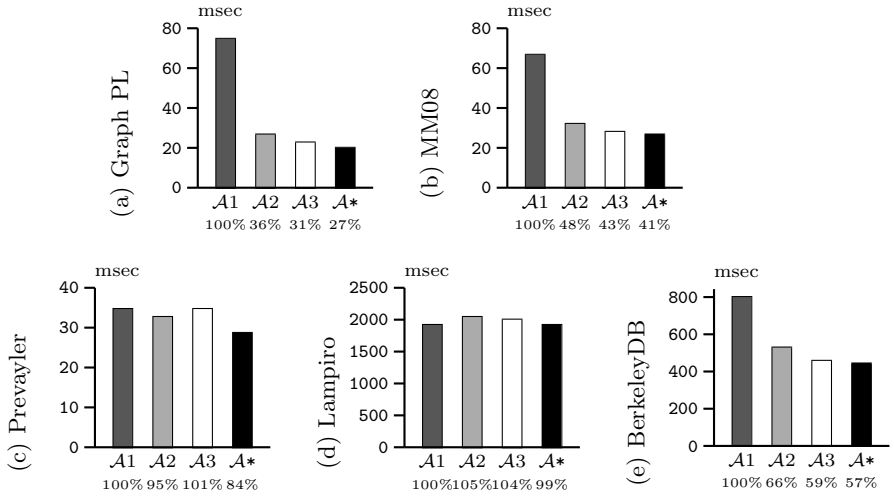


Fig. 15. Effectiveness of the combined Analysis: $\mathcal{A}*$ (shown in black).

Figure 15 plots the effectiveness of this new combined analysis $\mathcal{A}*$ against the analyses $\mathcal{A}1$ to $\mathcal{A}3$. As expected, this leads to a slightly more efficient analyses. In fact, we observe that $\mathcal{A}*$ is *consistently* faster than each of its constituent analyses, on all our benchmarks. We even see improvements on our fresh SPL, `Prevayler` (cf. Figure 15(c)), that we deliberately excluded from the study when devising $\mathcal{A}*$. Of course, this SPL only has one method with 8 configurations, so it does not exercise $\mathcal{A}3$ a lot. Even so, it increases the confidence that our lessons learned, and $\mathcal{A}*$, in particular, are, at least to some extent stable and not merely coincidental.

The gain factors for $\mathcal{A}*$ relative to $\mathcal{A}0$ are thus: 7.6 on `Graph PL`; 3.6 on `MobileMedia08`, 2.9 on `BerkeleyDB`; 1.2 on `Prevayler`; and 1.0 on `Lampiro`. In the end, we go from about *three* times faster on `Graph PL` with $\mathcal{A}3$ (31%)

compared to $\mathcal{A}1$, to about *four* times faster with $\mathcal{A}*$ (27%). Compared to $\mathcal{A}0$, its *average* gain factor across our five benchmarks is now 3.3 and, for `Graph PL`, we now obtain a gain factor of almost eight with $\mathcal{A}*$.

On Caching and Data Locality: When looking at the methods with more than one configuration in Figure 14, we see that $\mathcal{A}2$ is consistently faster than $\mathcal{A}1$. We claim that this is due to data locality and caching issues. In $\mathcal{A}2$, base lattice values are represented as a list structure and processed in sequence on statement nodes of the CFG along with their associated information (e.g., the configuration-set instrumentation and other relevant data stored for the fixed-point computation). In terms of caching, this is better strategy than $\mathcal{A}1$ which scatter computations and values across independent configuration runs, re-loading the same information over and over for every configuration.

Benchmark	Cache enabled:			Cache disabled:		
	$\mathcal{A}1$	$\mathcal{A}2$	$\mathcal{A}1$:$\mathcal{A}2$	$\mathcal{A}1$	$\mathcal{A}2$	$\mathcal{A}1$:$\mathcal{A}2$
`Graph PL`	38K	59K	1 : 1.5	139K	91K	1 : 0.7
`MM08`	43K	37K	1 : 0.9	125K	68K	1 : 0.5
`Prevayler`	317	371	1 : 1.2	1.1K	0.7K	1 : 0.6
`Lampiro`	n/a	n/a	n/a	n/a	n/a	n/a
`BerkeleyDB`	303K	341K	1 : 1.1	972K	567K	1 : 0.6

Fig. 16. Number of cache misses in $\mathcal{A}1$ vs. $\mathcal{A}2$ when analyzing methods with more than four configurations, under two different caching schemes: *cache enabled* (normal usage) vs. *cache disabled* (simulated cacheless scenario).

Figure 16, plots the cache misses of $\mathcal{A}1$ vs. $\mathcal{A}2$ when analyzing methods beyond four configurations under two different caching schemes. First, the "normal condition" where caching is *enabled*. Second, in a simulated cacheless scenario where we artificially disable caching by filling up the level three cache by traversing (and thereby loading into the cache) an 10MB bogus array prior to the execution of each transfer function. We see that $\mathcal{A}1$ generally incurs more cache misses than $\mathcal{A}2$ (except on `MobileMedia08`). However, the figure reveals that disabling caching indeed hurts $\mathcal{A}1$ more than $\mathcal{A}2$, presumably since data now has to be re-loaded on every configuration iterations of $\mathcal{A}1$. We take this as indication that $\mathcal{A}2$ has better cache properties than $\mathcal{A}1$.

On Memory Consumption of Analyses: Figure 17 lists the space consumption of the lattice information occupied by the analyses $\mathcal{A}1$ to $\mathcal{A}3$. The units for $\mathcal{A}1$ to $\mathcal{A}3$ are given as the *number of assignments* in the reaching definitions base lattice values (e.g., $|\{\texttt{x=1}, \texttt{y=2}, \texttt{z=4}\}| = 3$). For each SPL, we give numbers for the method with the highest number of configurations and the highest $\mathcal{A}2$ count (which for `Graph PL` and `BerkeleyDB` coincide).

If we compare the memory usage of $\mathcal{A}1$ and $\mathcal{A}2$, we see that $\mathcal{A}2$ takes up more memory (except, of couse, on featureless methods). Generally, it uses close

Benchmark	Max. memory consuming method	$\lvert\psi_{\mathtt{FM}}\rvert$	$\mathcal{A}1$	$\mathcal{A}2$	$\mathcal{A}3$	$\mathcal{A}1$:$\mathcal{A}2$	$\mathcal{A}2$:$\mathcal{A}3$
`Graph PL`	`Vertex.display()`	106	8.7K	569K	160K	1 : 65	3.5 : 1
`MM08`	`MediaController.handleCommand()`	20	12K	204K	146K	1 : 17	1.4 : 1
`MM08`	`MediaListScreen.initMenu()`	24	792	12K	8.3K	1 : 15	1.5 : 1
`Prevayler`	`P'J'.recoverPendingTransactions()`	2	3.9K	7.1K	6.9K	1 : 1.8	1.0 : 1
`Prevayler`	`PrevaylerFactory.publisher()`	8	102	423	174	1 : 4.1	2.4 : 1
`Lampiro`	`Inftree.clinit()`	1	3.2M	3.2M	3.2M	1 : 1.0	1.0 : 1
`Lampiro`	`RegisterScreen.xmppLogin()`	4	7.7K	31K	14K	1 : 4.0	2.1 : 1
`BerkeleyDB`	`DbRunAction.main()`	40	36K	1.2M	614K	1 : 33	2.0 : 1

Fig. 17. Memory consumption of lattice information: $\mathcal{A}1$ vs. $\mathcal{A}2$ vs. $\mathcal{A}3$.

to (although slightly less than) the number of valid configurations times more memory. This is consistent with our expectations from Section 6. Excess memory consumption has not been a problem on any of our five benchmarks.

We also see that the memory usage is reduced with the sharing of $\mathcal{A}3$. The shared analysis $\mathcal{A}3$ may reduce space usage anywhere between a factor of one to about 3.5 (in the case of `Vertex.display()` of `Graph PL`), depending on the number of configurations.

8 Related Work

Data-Flow Analysis: The idea of making dataflow analysis sensitive to statements that may or may not be executed is related to *path-sensitive* dataflow analysis. Such analyses compute different analysis information along different execution paths aiming to improve precision by disregarding spurious information from infeasible paths [37] or to optimize frequently executed paths [38]. Earlier, disabling infeasible dead statements has been exploited to improve the precision of constant propagation [39] by essentially running a dead-code analysis capable of tagging statements as executable or non-executable during constant propagation analysis.

Predicated dataflow analysis [40] introduced the idea of using propositional logic predicates over runtime values to derive so-called *optimistic* dataflow values guarded by predicates. Such analyses are capable of producing multiple analysis versions and keeping them distinct during analysis much like our simultaneous analyses. However, their predicates are over dynamic state rather than SPL feature constraints for which everything is statically decidable.

The novelty in our paper is the application of the dataflow analysis framework to the domain of SPLs giving rise to the concept of *feature-sensitive* analyses that take conditionally compiled code and feature models into consideration so as to analyze not one program, but an entire SPL family of programs.

Analysis of SPLs: Recently and inspired by our work, Bodden proposed [11] a mechanism for encoding dataflow analyses specified via the IFDS framework [42] in a graph representation using the more general IDE analysis framework [43] and use that to add SPL configurations on top of the analysis, thereby essentially

"lifting" an IFDS dataflow analysis. The idea has not yet been evaluated. Interestingly, all of our analysis approaches $\mathcal{A}1$ to $\mathcal{A}4$ apply equally to that context and insights from this paper could benefit in that work.

Thüm et al. survey analysis strategies for SPLs [29], focusing on parsing [12], type checking [13, 44], model checking [45, 46], and verification [20, 47, 21]. The surveyed work does not include product line dataflow analysis approaches, but shares with our work the general goal of checking properties of a product line with reduced redundancy and efficiency. Similar to our work, a number of approaches covered by the survey adopt a *family-based* analysis strategy, manipulating only family artifacts such as code assets and feature model as in our feature-sensitive analyses, $\mathcal{A}1$ to $\mathcal{A}4$. Contrasting, *product-based* strategies, such as the $\mathcal{A}0$ brute-force generate-and-analyze approach we use as baseline, manipulate products and therefore might be too expensive for product lines having a large number of products. Product-based strategies, however, might be appealing because they can simply reuse existing analyses.

In SPLs, there are features whose presence or absence do not influence the test outcomes, which makes many feature combinations unnecessary for a particular test. This idea of selecting only relevant features for a given test case was proposed in a recent work [41]. It uses dataflow analysis to recover a list of features that are reachable from a given test case. The unreachable features are discarded, decreasing the number of combinations to test. In contrast, we defined and demonstrated how to automatically make any conventional dataflow analysis able to analyze product lines in a feature-sensitive way. Thus, our feature-sensitive idea might be used in such a work (testing). For example, it might further reduce the time spent figuring out which relevant feature combinations to test.

Safe composition (SC) relates to the safe generation and verification of properties for SPL assets providing guarantees that the product derivation process generates products with properties that are obeyed [13, 44, 32, 31, 35]. Safe composition may help in finding type and definition-usage errors like undeclared variables, undeclared fields, multiply declared variables, unimplemented abstract methods, or ambiguous methods. We complement safe composition, since when using our feature-sensitive idea, we are able to catch any errors expressible as a dataflow analysis (e.g., uninitialized variables and null pointers).

In the parsing context, Kaestner et al. [12] provide a variability-aware parser which is capable of parsing code without preprocessing it. The parser also performs instrumentation as we do, but on tokens, instead of statements. When the parser reaches a token instrumented with feature A, it splits into branches. Then, one parser assumes that feature A is selected and another assumes that A is not. So, the former consumes the token and the latter skips it. To avoid parsing tokens repeatedly (like a parenthesis instrumented with no feature), the branches are joined. This approach is similar to our shared simultaneous analyses $\mathcal{A}3$ and $\mathcal{A}4$, where we lazily split and merge sets of configurations. In the verification context, Thüm et al. [36] show how to formally prove that contracts are satisfied by a product line implementation. They also report significant re-

duction, but not as much as we discuss here in the paper, of the verification time for a small example.

Classen et al. [45, 21] shows that behavioral models offer little means to relate different products and their respective behavioral descriptions. To minimize this limitation, they present a transition system to describe the combined behavior of an entire SPL. Additionally, they provide a model checking technique supported by a tool capable of verifying properties for all the products of an SPL once. Like our work, they claim that checking all product combinations at once instead of each product separately is faster. Their model checking algorithm was on average 3.5 times faster than verifying products separately which is comparable to our results. A related approach for model checking of product lines [33] proposes a process algebra extension to represent variability, but brings no performance results. Similarly, Lauenroth et al. [34] focus on extending I/O-automata, bringing preliminar feasibility performance results.

We recently proposed the concept of emergent interfaces [6]. These interfaces emerge on demand to give support for specific SPL maintenance and thus help developers understand and manage dependencies between features. Feature dependencies such as assigning a value to a variable used by another feature, have to be generated by feature-sensitive analyses. Thus, our present work may be used to generate emergent interfaces to support SPL maintenance. Our analyses are more efficient than the brute-force approach, which is important to improve the performance during the computation of emergent interfaces.

9 Conclusion

In this paper, we presented four approaches for taking any conventional dataflow analysis and automatically lifting it into a feature-sensitive analysis, $\mathcal{A}1$ to $\mathcal{A}4$, capable of analyzing all configurations of an SPL without having to generate all its products explicitly.

To evaluate these approaches, we applied them to the ubiquitous reaching definitions dataflow analysis and lifted it to produce four different feature-sensitive analyses. Experimental validation of three of these indicated that the feature-sensitive approaches were significantly faster than the naive brute-force approach, especially for SPLs with intensive feature usage.

When investigating closer, we found that the number of configurations may serve as a predictor for which of the analyses $\mathcal{A}1$ to $\mathcal{A}3$ that is likely to be the fastest. The consecutive approach ($\mathcal{A}1$) was statistically significantly faster on methods with only *one* configuration as $\mathcal{A}2$ and $\mathcal{A}3$ incur an overhead penalty from representing analysis values as a *list of one*, respectively, a *set of one* element. The simultaneous approach ($\mathcal{A}2$) was statistically significantly fastest on methods with *two to three* configurations. On methods with *four* configurations, there was no significant difference between $\mathcal{A}2$ and $\mathcal{A}3$. Finally, on methods *beyond four* configurations, the shared approach ($\mathcal{A}3$) was the fastest because it can share values among configurations and thereby avoid redundant computations.

We then combined these insights into an interprocedurally combined analysis strategy, $\mathcal{A}*$, which uses the number of configurations of each method analyzed as a predictor for attempting to select the fastest analysis for the method. Not surprisingly, this analysis was faster than its individual constituent analyses. This was even true on a fresh SPL which we did not use in the analysis and construction of the $\mathcal{A}*$ strategy. Hence, we have an indication that our insights and results may apply and transfer to other SPLs.

On our biggest SPL, which also took the longest to analyze, `BerkeleyDB`, we were able to cut the total analysis time from 5.4 to 1.8 seconds, compared to the existing brute-force generate-and-analyze alternative. On our most feature-intensive benchmark, `Graph PL`, we were able to reduce the total analysis time from almost 500 msec to 63 msecs (i.e., an improvement close to an order of magnetude). On average over the benchmarks, our analysis strategy is 3.3 times faster than the existing feature-oblivious alternative.

We futher conclude that our four feature-sensitive approaches ($\mathcal{A}1$ to $\mathcal{A}3$) have different performance and memory consumption characteristics.

Acknowledgments. We would like to thank CNPq, FACEPE, and National Institute of Science and Technology for Software Engineering (INES), for partially supporting this work. Also, we thank SPG[4] members for the fruitful discussions about this paper. We also thank Julia Lawall and the anonymous reviewers for many comments and suggestions that helped improve this paper.

References

1. K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.
2. P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001.
3. C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, (Leipzig, Germany), pp. 311–320, ACM, 2008.
4. B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan, "Can we refactor conditional compilation into aspects?," in *Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD'09)*, (Charlottesville, Virginia, USA), pp. 243–254, ACM, 2009.
5. G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st annual ACM symposium on Principles of programming languages (POPL'73)*, (Boston, Massachusetts), pp. 194–206, ACM, 1973.
6. M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba, "Emergent feature modularization," in *Onward! 2010, affiliated with the 1st ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH'10)*, (Reno, NV, USA), pp. 11–18, 2010.
7. M. Fowler, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[4] `http://www.cin.ufpe.br/spg`

8. F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, USA: Springer-Verlag, 1999.
9. C. Kästner, S. Apel, and M. Kuhlemann, "A model of refactoring physically and virtually separated features," in *Proceedings of the eighth international conference on Generative programming and component engineering*, GPCE '09, (New York, NY, USA), pp. 157–166, ACM, 2009.
10. C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba, "Intraprocedural dataflow analysis for software product lines," in *AOSD*, pp. 13–24, 2012.
11. E. Bodden, "Position Paper: Static flow-sensitive & context-sensitive information-flow analysis for software product lines," in *ACM SIGPLAN Seventh Workshop on Programming Languages and Analysis for Security (PLAS 2012)*, jun 2012.
12. C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'11)*, (Portland, OR, USA), pp. 805–824, ACM, 2011.
13. S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer, "Type safety for feature-oriented product lines," *Automated Software Engineering*, vol. 17, pp. 251–300, September 2010.
14. H. Spencer and G. Collyer, "#ifdef considered harmful, or portability experience with C news," in *Proceedings of the Usenix Summer 1992 Technical Conference*, (Berkeley, CA, USA), pp. 185–198, Usenix Association, June 1992.
15. M. Krone and G. Snelting, "On the inference of configuration structures from source code," in *Proceedings of the 16th International Conference on Software Engineering (ICSE'04)*, (Los Alamitos, CA, USA), pp. 49–57, IEEE Computer, 1994.
16. M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," *IEEE Transactions on Software Engineering*, vol. 28, pp. 1146–1170, December 2002.
17. J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, (Cape Town, South Africa), pp. 105–114, ACM, 2010.
18. C. Kästner, *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, Germany, May 2010.
19. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) feasibility study," tech. rep., Carnegie-Mellon University Software Engineering Institute, November 1990.
20. H. Post and C. Sinz, "Configuration lifting: Verification meets software configuration," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, (L´Aquila, Italy), pp. 347–350, IEEE Computer Society, 2008.
21. S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of feature interactions using feature-aware verification," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, (Lawrence, USA), IEEE Computer Society, November 2011.
22. K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness ocl constraints," in *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, (New York, NY, USA), pp. 211–220, ACM, 2006.
23. S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. 27, pp. 509–516, June 1978.

24. N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.
25. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON'99)*, pp. 13–, IBM Press, 1999.
26. "The colt project: Open source libraries for high performance scientific and technical computing in java." CERN: European Organization for Nuclear Research.
27. E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, and F. Dantas, "Evolving software product lines with aspects: an empirical study on design stability," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, (Leipzig, Germany), pp. 261–270, ACM, 2008.
28. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.
29. T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake, "Analysis strategies for software product lines," tech. rep., School of Computer Science, University of Magdeburg, Germany, 2012. Technical Report FIN-004-2012.
30. S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of feature interactions using feature-aware verification," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, (Washington, DC, USA), pp. 372–375, IEEE Computer Society, 2011.
31. B. Delaware, W. R. Cook, and D. Batory, "Fitting the pieces together: a machine-checked model of safe composition," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, (New York, NY, USA), pp. 243–252, ACM, 2009.
32. S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, (New York, NY, USA), pp. 95–104, ACM, 2007.
33. A. Gruler, M. Leucker, and K. D. Scheidemann, "Modeling and model checking software product lines," in *FMOODS*, pp. 113–131, 2008.
34. K. Lauenroth, K. Pohl, and S. Toehning, "Model checking of domain artifacts in product line engineering," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (Washington, DC, USA), pp. 269–280, IEEE Computer Society, 2009.
35. C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type checking annotation-based product lines," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, pp. 14:1–14:39, July 2012.
36. T. Thüm, I. Schaefer, M. Hentschel, and S. Apel, "Family-based deductive verification of software product lines," in *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, (New York, NY, USA), pp. 11–20, ACM, 2012.
37. T. Ball and S. K. Rajamani, "Bebop: a path-sensitive interprocedural dataflow engine," in *PASTE'01*, (Snowbird, Utah, USA), pp. 97–103, 2001.
38. G. Ammons and J. R. Larus, "Improving data-flow analysis with path profiles," in *Programming Language Design and Implementation (PLDI'98)*, (Montreal, Canada), pp. 72–84, 1998.

39. M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 181–210, April 1991.
40. S. Moon, M. W. Hall, and B. R. Murphy, "Predicated array data-flow analysis for run-time parallelization," in *Proceedings of the 12th International Conference on Supercomputing (ICS'98)*, (Melbourne, Australia), pp. 204–211, ACM, 1998.
41. C. Hwan, P. Kim, D. Batory, and S. Khurshid, "Reducing combinatorics in testing product lines," in *Proceedings of the 10th International Conference on Aspect-oriented Software Development (AOSD'11)*, (Porto de Galinhas, Brazil), pp. 57–68, ACM, 2011.
42. T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, (New York, NY, USA), pp. 49–61, ACM, 1995.
43. M. Sagiv, T. Reps, and S. Horwitz, "Precise interprocedural dataflow analysis with applications to constant propagation," in *Selected papers from the 6th international joint conference on Theory and practice of software development*, TAPSOFT '95, (Amsterdam, The Netherlands, The Netherlands), pp. 131–170, Elsevier Science Publishers B. V., 1996.
44. C. Kästner and S. Apel, "Type-checking software product lines - a formal approach," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, (L'Aquila, Italy), pp. 258–267, 2008.
45. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, (Cape Town, South Africa), pp. 335–344, ACM, 2010.
46. A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *ICSE*, pp. 321–330, 2011.
47. C. H. P. Kim, E. Bodden, D. Batory, and S. Khurshid, "Reducing configurations to monitor in a software product line," in *1st International Conference on Runtime Verification (RV)*, vol. 6418 of *LNCS*, (Malta), Springer, November 2010.