Intraprocedural Dataflow Analysis for Software Product Lines

Claus Brabrand[1,2] Márcio Ribeiro[2,3] Társis Tolêdo[2] Paulo Borba[2]

[1] IT University of Copenhagen, Rued Langgaards Vej 7, DK-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 Melo Mota, 57.072-970, Maceió, Brazil

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

Abstract

Software product lines (SPLs) are commonly developed using annotative approaches such as conditional compilation that come with an inherent risk of constructing erroneous products. For this reason, it is essential to be able to analyze SPLs. However, as dataflow analysis techniques are not able to deal with SPLs, developers must generate and analyze all valid methods 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 three different ways. All are capable of analyzing *all* valid methods of an SPL without having to generate all of them explicitly. We have implemented all analyses as extensions of SOOT's intraprocedural dataflow analysis framework and experimentally evaluated their performance and memory characteristics on four qualitatively different SPLs. The results indicate that the feature-sensitive analyses are on average 5.6 times faster than the brute force approach on our SPLs, and that they have different time and space tradeoffs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.2 [*Theory of Computation*]: Semantics of Programming Languages — *Program Analysis*

General Terms Performance, Experimentation, Design

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 [7, 24]. Developers often implement variant behavior and associated features with conditional compilation constructs like `#ifdef` [1, 16], 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.

Since code assets might not be valid programs or program elements, existing standard dataflow analyses, which are for instance essential for supporting optimization [18] and maintenance [26] 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 [10]. Also, this is bad for maintenance tools [26] that help developers understand and manage dependencies between features.

To solve this problem and enable more efficient dataflow analysis of SPLs, we propose three approaches for taking any standard intraprocedural dataflow analysis and automatically lifting it into a corresponding *feature-sensitive* analysis that we can use to directly analyze code assets. The approaches analyze all configurations and thus avoid explicitly generating all possible methods of an SPL. Although we focus on SPLs developed with conditional compilation constructs, our results apply to other similar annotative variability mechanisms [16].

We evaluate our three *feature-sensitive* approaches (*consecutive*, *simultaneous*, and *shared simultaneous*) and compare them with a *brute force* intraprocedural approach that

AOSD'12, March 25–30, 2012, Potsdam, Germany.

generates and analyzes all possible methods individually. We report on a number of performance and memory consumption experiments using two dataflow analyses (*definite assignments* and *reaching definitions* [23]) and four SPLs from different domains, with qualitatively different numbers of features, products, `#ifdef` statements, and other factors that might impact performance and memory usage results. We find that, for the analyses and SPLs used, when analyzing all configurations simultaneously (*simultaneous* approach), we reduce analysis time by a factor of up to more than eight times on SPLs with intensive feature usage (intensive `#ifdef` presence). For SPLs with low feature usage the *simultaneous* approach is only slightly faster than the brute force approach. In addition, the former consumes more memory when compared to the *shared simultaneous* approach, which shares values corresponding to configurations during the analysis.

We organize the rest of this paper as follows. Using a concrete example, Section 2 discusses and motivates the need for dataflow analysis of software product lines. Then, we introduce conditional compilation and feature models. After that, we briefly recall basic dataflow analysis concepts and present the main contributions of this paper:

- a *consecutive feature-sensitive* approach that analyzes all SPL configurations, one at a time and *simultaneous* and *shared simultaneous feature-sensitive* approaches that analyze all SPL configurations at the same time (Section 5);
- an *experimental prototype* implementation of the three above analyses; and
- *empirical evidence* of the superiority of our feature-sensitive approaches; in particular the *simultaneous* and *shared simultaneous* approaches, which are faster than the *consecutive* one, but use more memory (Section 6).

2. Motivating Example

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 `#ifdefs`.

Figure 1 shows a code snippet extracted from `Lampiro` implemented in Java with the Antenna[2] preprocessor. As can be seen, if the `GLIDER` feature is *not* present (see the `#ifndef` statement), the `logo` variable receives an image instantiated by the `createImage` method, so it is *initialized*. However, this variable is *uninitialized* if the `GLIDER` feature is *present* in the product. Such mistakes—and others like undeclared variables, unused variables, and null pointers—commonly occur when using conditional compilation. Indeed, despite their widespread usage to implement variability in SPLs, `#ifdefs` pollute the code, lack separation of concerns, and make maintenance tasks harder [8, 20, 21, 27].

```
Image logo;
...
//#ifndef GLIDER
...
logo = Image.createImage("/icons/lampiro_icon.png");
...
//#endif
...
UILabel uimg = new UILabel(logo);
```

Figure 1. Uninitialized variable when `GLIDER` is present.

Thus, to maintain this kind of SPL, it is important to analyze its code and determine whether developers are introducing errors. For instance, consider the case where a developer is supposed to change the value of a variable that belongs to feature A. An analysis could be useful to warn of another feature, B, using the same variable just modified. Such feature dependency information is a signal to investigate feature B, to make sure that the modification did not introduce any problems in it. We proposed an idea to provide information about this kind of feature dependency [26]. This was our original motivation for adapting dataflow analysis for SPLs.

To capture these dependencies and consequently problems like uninitialized variables in SPLs, 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 6, we are able to decrease such costs.

3. Conditional Compilation

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

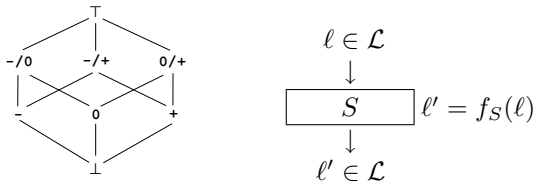
$$\begin{array}{lll} S & ::= & \texttt{"ifdef" "("}\ \phi\ \texttt{")"}\ S \\ \phi & ::= & f \in \mathbb{F} \mid \neg\phi \mid \phi \wedge \phi \end{array}$$

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, $\mathbb{F}$. We further eliminate `#elif` and `#else` branches by turning them into the normalized syntactic `ifdef` form listed in BNF above.

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

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

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



(a) Lattice for sign analysis. (b) Effect of transfer function, f_S.

Figure 2. Lattice and transfer function.

To yield only valid configurations, sets of configurations are usually further restricted by a so-called *feature model* [13]. Conceptually, a feature model is just a propositional logic formula. Here is an example of a feature model with alphabet $\mathbb{F} = \{\texttt{Car}, \texttt{Air}, \texttt{Basic}, \texttt{Turbo}\}$:

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

corresponding to the following set of *valid* configurations:

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

4. Dataflow Analysis

A Dataflow Analysis [18] 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 [23]. Lattices are usually described diagrammatically using *Hasse 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 2(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 [23] 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 $\texttt{-} \sqcup \texttt{0/+} = \top$.

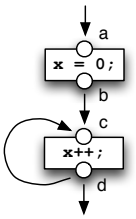
Transfer Functions: Each statement, S, will have an associated *transfer function*, $f_S : \mathcal{L} \rightarrow \mathcal{L}$, which simulates the execution of S at compile-time (with respect to what is being analyzed). Figure 2(b) illustrates the effect of executing transfer function f_S. Lattice 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 2(a):

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

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`. 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 [23].

Analysis: Figure 3 shows how to combine the control-flow graph, lattice, and transfer functions to perform dataflow analysis on a tiny example program.

First (cf. Figure 3(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 3(b)), the annotated CFG is turned into a *whole-program transfer function*, $T : \mathcal{L}^4 \rightarrow \mathcal{L}^4$, 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 3(c)), we use the Fixed-Point Theorem [23] to compute the 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. As seen in Figure 3(c), we reach the fixed-point in five iterations (since $T^4(\bot) = T^5(\bot)$) and the least-fixed point, and hence the 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.)



(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.

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

(c) Fixed-point iteration.

Figure 3. Combining CFG, lattice, and transfer functions to perform dataflow analysis (as a fixed-point iteration).

```
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 $\{A, B\}$).

Figure 4. A tiny example of an SPL method along with its three distinct method variants.

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 intraprocedural dataflow analysis for software product lines (summarized in Figure 6). 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 program (cf. Figure 4(a)) that increases and decreases a variable, depending on the features enabled.

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: $[\![\psi_{\text{FM}}]\!] = \{\{A\}, \{B\}, \{A, B\}\}$.

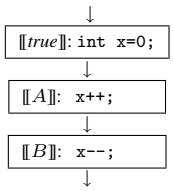
$\mathcal{A}$1: Brute Force Analysis (Feature-Oblivious)

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, we have to build and analyze the *three* distinct methods as illustrated in Figure 4(b).

$\mathcal{A}$2: Consecutive Feature-Sensitive 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.

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*. Here is the instrumented CFG for our tiny method of Figure 4(a):

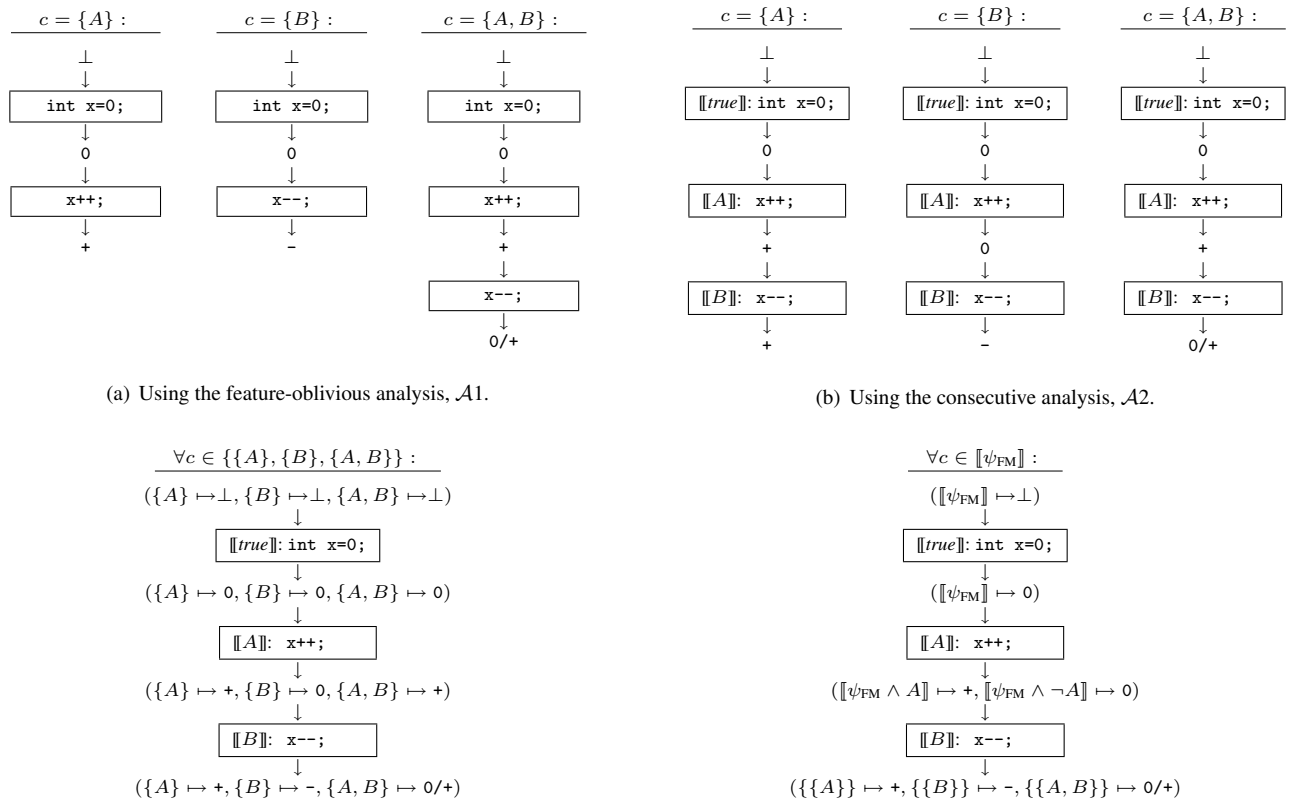


We label each node with "$[\![\phi]\!]$: S" where S is the statement and $[\![\phi]\!]$ is the configuration set associated with the statement. Unconditionally executed statements (e.g., `int x=0;`) are associated with the set of all configurations, $[\![true]\!]$. Statements that are nested inside several `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 execute the feature-oblivious transfer function, f_S, in a given configuration, c; i.e., deciding $c \in [\![\phi]\!]$ (cf. Figure 6). Since the lifting only either applies the feature-oblivious 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}$2, all we need to do is to combine the CFG, lattice, and transfer functions as explained in Section 4. Figure 5(b) shows the



(a) Using the feature-oblivious analysis, $\mathcal{A}1$.

(b) Using the consecutive analysis, $\mathcal{A}2$.

(c) Using the simultaneous analysis, $\mathcal{A}3$.

(d) Using the shared simultaneous analysis, $\mathcal{A}4$.

Figure 5. Results of using the four analyses on our tiny example program `m` (that increases and decreases variable, x).

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. $\mathcal{A}1$ and $\mathcal{A}2$ 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.

$\mathcal{A}3$: Simultaneous Feature-Sensitive 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. 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 $\mathcal{A}3$ is the same as that of $\mathcal{A}2$ 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 feature-oblivious lattice, $\mathcal{L}$, such that it has one element per valid configuration:

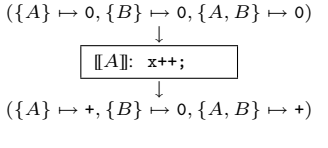
$$\mathcal{L}_3 \quad = \quad [\![\psi_{\text{FM}}]\!] \to \mathcal{L}$$

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:

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

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

Transfer Functions: We *lift* the transfer functions correspondingly so they work on elements of the lifted lattice in a point-wise manner. The feature-oblivious 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 $(\{A\} \mapsto \texttt{0}, \{B\} \mapsto \texttt{0}, \{A, B\} \mapsto \texttt{0})$ is:



The transfer function of the feature-oblivious analysis 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 `0`. Figure 6 depicts and summarizes the effect of the lifted transfer function on the lifted lattice.

Since the feature-sensitive transfer function on $[\![\psi_{\text{FM}}]\!] \to \mathcal{L}$ only applies monotone transfer functions on $\mathcal{L}$ in a pointwise 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 5(c) 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 $\mathcal{A}2$, 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 $\mathcal{A}2$ in order to reach its fixed-point because of the pointwise lifted lattice. Again, it is fairly obvious that $\mathcal{A}2$ and $\mathcal{A}3$ compute the same information; the only difference being that $\mathcal{A}2$ does one fixed-point iteration per valid configuration whereas $\mathcal{A}3$ computes the same information in one iteration in a pointwise manner.

$\mathcal{A}4$: Shared Simultaneous Feature-Sensitive Analysis

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.

Control-Flow Graph: The CFG of $\mathcal{A}4$ is the same as that of $\mathcal{A}3$.

Lattice: To accomodate the sharing, the lifted lattice of $\mathcal{A}4$ will, instead of mapping configurations to base lattice values, map *sets of configurations* to base lattice values:

$$\mathcal{L}_4 \quad = \quad 2^{[\![\psi_{\text{FM}}]\!]} \hookrightarrow \mathcal{L}$$

This allows $\mathcal{A}4$ 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 $\mathcal{A}3$, $(\{A\} \mapsto \ell, \{B\} \mapsto \ell, \{A, B\} \mapsto \ell)$, can now be represented by $([\![A \vee B]\!] \mapsto \ell)$ where the three configurations, $[\![A \vee B]\!] = \{\{A\}, \{B\}, \{A, B\}\}$, *share* the base lattice value, ℓ.

Transfer Functions: The transfer functions of $\mathcal{A}4$ work by lazily *splitting* sets of configurations, $[\![\psi]\!]$, in two disjoint parts, depending on the feature constraint, ϕ, attached with

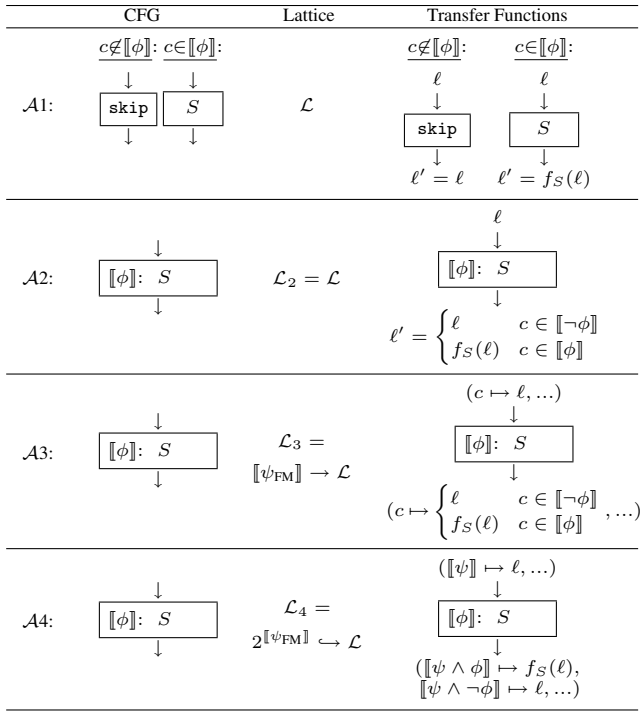
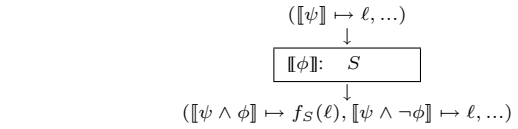


Figure 6. Summary of dataflow analyses for SPLs.

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.:



Note that $[\![\psi \wedge \phi]\!] \cup [\![\psi \wedge \neg\phi]\!] = [\![(\psi \wedge \phi) \vee (\psi \wedge \neg\phi)]\!] = [\![\psi \wedge (\phi \vee \neg\phi)]\!] = [\![\psi \wedge \textit{true}]\!] = [\![\psi]\!]$. In cases where $[\![\psi]\!]$ would be split into "nothing", $\emptyset$, and "everything", $[\![\psi]\!]$ (which happens whenever $\psi \wedge \phi \equiv \textit{false}$ or $\psi \wedge \neg\phi \equiv \textit{false}$), we eliminate the *false* constituents in order to ensure a canonical (minimal and finite) representation. It is also possible to join lattice values that are equal. However, this might compromise performance (in exchange for memory gains) due to the equality comparisons needed to determine if joins are possible.

Analysis: As always for the analysis, we simply combine the CFG, lattice, and transfer functions to achieve our shared simultaneous analysis. Figure 5(d) shows how this analysis will analyze our tiny program example from earlier. (For legibility, the last line of the figure is written with the expanded *sets of configurations* rather than with formula notation.) $\mathcal{A}3$ and $\mathcal{A}4$ compute the same information; $\mathcal{A}4$ just represents the same information more compactly using sharing.

$$
\begin{array}{lcl}
\text{tasks}(\mathcal{A}1) & = & |[\![\psi_{\text{FM}}]\!]| \cdot \text{compile} + |[\![\psi_{\text{FM}}]\!]| \cdot \text{analyze}_{\mathcal{A}1} \\
\text{tasks}(\mathcal{A}2) & = & \text{compile} + \text{instrument} + |[\![\psi_{\text{FM}}]\!]| \cdot \text{analyze}_{\mathcal{A}2} \\
\text{tasks}(\mathcal{A}3) & = & \text{compile} + \text{instrument} + \text{analyze}_{\mathcal{A}3} \\
\text{tasks}(\mathcal{A}4) & = & \text{compile} + \text{instrument} + \text{analyze}_{\mathcal{A}4}
\end{array}
$$

Figure 7. Overall tasks performed by each of the analyses.

Other Analysis Approaches

A couple of variations of the feature-sensitive analyses are possible. One could retain the instrumented CFG calculated in $\mathcal{A}2$ and $\mathcal{A}3$, but then *specialize* [12] 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 $\mathcal{A}2$ 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 `ifdefs` into normal `ifs` and turn feature names into static booleans [4, 25] which could then be resolved by techniques such as *partial evaluation* [12] 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.

Asymptotic complexity (TIME and SPACE)

We now consider and compare the asymptotic complexity of $\mathcal{A}2$, $\mathcal{A}3$, and $\mathcal{A}4$ in terms of first overall tasks, then performance, and finally memory consumption.

Total Time (including compilation): Figure 7 considers the overall tasks performed for each SPL method analyzed in each of the analyses. Apart from $\mathcal{A}3$ vs. $\mathcal{A}4$, they all differ substantially in the number of times each of the tasks are performed. Not surprisingly, $\mathcal{A}1$ needs to do a lot of (brute force) compilation. 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. $\mathcal{A}2$ performs the analysis (i.e., the fixed-point computation) for every valid configuration whereas $\mathcal{A}3$ and $\mathcal{A}4$ only do this once. We return to these considerations, in practice, when we discuss our experimental results (cf. Section 6).

Performance of Analyses (TIME): The asymptotic time complexity of the $\mathcal{A}2$ analysis is:

$$\text{TIME}(\mathcal{A}2) = \mathcal{O}(|[\![\psi_{\text{FM}}]\!]| \cdot |\mathcal{G}| \cdot T_2 \cdot h(\mathcal{L}_2))$$

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_2 is the execution time of a transfer function on the $\mathcal{L}_2$ lattice; and $h(\mathcal{L}_2)$ is the height of the $\mathcal{L}_2$ 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_2, and can be executed a worst-case maximum of $h(\mathcal{L}_2)$ number of times.

Analogously, we can quantify the asymptotic time complexity of $\mathcal{A}3$:

$$\text{TIME}(\mathcal{A}3) = \mathcal{O}(|\mathcal{G}| \cdot T_3 \cdot h(\mathcal{L}_3))$$

which is similar to $\mathcal{A}2$, except that we do not need to analyze $|[\![\psi_{\text{FM}}]\!]|$ times and that the numbers are parameterized by the $\mathcal{A}3$ lattice and transfer functions. For the height of the lattice $\mathcal{L}_3$, we have:

$$
\begin{array}{lcllcl}
h(\mathcal{L}_3) & = & h([\![\psi_{\text{FM}}]\!] \to \mathcal{L}_2) & = & h(\mathcal{L}_2^{|[\![\psi_{\text{FM}}]\!]|}) \\
 & = & \sum_{c \in [\![\psi_{\text{FM}}]\!]} h(\mathcal{L}_2) & = & |[\![\psi_{\text{FM}}]\!]| \cdot h(\mathcal{L}_2)
\end{array}
$$

Note, however, that this is a purely theoretically worst case that does not naturally arise in practice because of the pointwise nature of $\mathcal{A}3$. Since all configurations are independent, the penalty for $\mathcal{A}3$ will not be the *sum*, but rather only the *maximum* number of fixed-point iterations of $\mathcal{A}2$. In practice, we have not observed any significant cost on behalf of $\mathcal{A}3$ from this effect, as we will see in Section 6. The remaining speed factor between $\mathcal{A}2$ and $\mathcal{A}3$ thus boils down to:

$$\mathcal{A}2 : \mathcal{A}3 \quad = \quad |[\![\psi_{\text{FM}}]\!]| \cdot T_2 \ : \ T_3$$

In theory, we would not expect any difference in the speed of the two analyses; $\mathcal{A}2$ makes a sequence of n analyses and $\mathcal{A}3$ makes one analysis in which each step costs n. However, as we will see in Section 6, $\mathcal{A}3$ has better cache performance than $\mathcal{A}2$, since statement nodes only have to be retrieved and evaluated once per transfer function in $\mathcal{A}3$, instead of once per configuration as in $\mathcal{A}2$. Apart from data, also the fixed-point iteration code only runs once instead of once per configuration.

Memory Consumption of Analyses (SPACE): The asymptotic space complexity of the analyses $\mathcal{A}2$ and $\mathcal{A}3$ is simply proportional to the amount of data occupied by the lattice values:

$$
\begin{array}{lcl}
\text{SPACE}(\mathcal{A}2) & = & \mathcal{O}(|\mathcal{G}| \cdot log(|\mathcal{L}_2|)) \\
\text{SPACE}(\mathcal{A}3) & = & \mathcal{O}(|\mathcal{G}| \cdot log(|\mathcal{L}_3|))
\end{array}
$$

Comparing the two, we can derive that:

$$
\begin{array}{lcllcl}
log(|\mathcal{L}_3|) & = & log(|[\![\psi_{\text{FM}}]\!] \to \mathcal{L}_2|) & = & log(|\mathcal{L}_2^{|[\![\psi_{\text{FM}}]\!]|}|) \\
 & = & log(|\mathcal{L}_2|^{|[\![\psi_{\text{FM}}]\!]|}) & = & |[\![\psi_{\text{FM}}]\!]| \cdot log(|\mathcal{L}_2|)
\end{array}
$$

which thus means that the difference is:

$$\text{SPACE}(\mathcal{A}3) = |[\![\psi_{\text{FM}}]\!]| \cdot \text{SPACE}(\mathcal{A}2)$$

This relationship is also evident when comparing Figures 5(b) and 5(c). Although $\mathcal{A}3$ requires $n = |[\![\psi_{\text{FM}}]\!]|$ times more memory to run, it is always possible to cut the $\mathcal{A}3$ lattice into k *slices* of n/k columns (i.e., analyze n/k number of configurations at a time). This provides a way of combining the time and space characteristics of $\mathcal{A}2$ and $\mathcal{A}3$ (and $\mathcal{A}4$).

Benchmark	LOC	$\lvert\mathbb{F}\rvert$	$\lvert 2^{\mathbb{F}_{local}}\rvert$	#methods	cc%
Graph PL	1,350	18	$2^9 = 512$	135 (964)	82%
MobileMedia08	5,700	14	$2^7 = 128$	285 (821)	45%
Lampiro	45,000	11	$2^2 = 4$	1,949 (1,980)	1.5%
BerkeleyDB	84,000	42	$2^8 = 256$	3,605 (7,446)	40%

Figure 8. Size metrics for our four SPL benchmarks.

6. Evaluation

We first present our study settings and then present our results in terms of total analysis time, performance, and memory consumption.

6.1 Study settings

To validate the ideas, we have implemented and evaluated the performance and memory characteristics of two ubiquitous intraprocedural dataflow analyses; namely *reaching definitions* and *definite assignments* (both of which are implemented using SOOT's interprocedural dataflow analysis framework for analyzing Java programs [28]).

We have subsequently lifted them into consecutive, simultaneous, and shared simultaneous feature-sensitive analyses for SPLs. Since we are using intraprocedural analyses which analyze one method at a time, we can use the *local* set of configurations, $\mathbb{F}_{local}$, local to each method which significantly reduces the size of the lattices we work with. However, instead of using the set of valid configurations, $[\![\psi_{\text{FM}}]\!]$, we use the set of all feature combinations, $2^{\mathbb{F}_{local}}$. Restricting to valid configurations only would make all feature-sensitive analyses faster.

We have chosen four qualitatively different SPL benchmarks, summarized in Figure 8. `Graph PL` (GPL) is a product line of small size with intensive feature usage [16] for desktop applications. `MobileMedia08` is a product line of small size and moderate feature usage [9] for mobile applications for dealing with multi-media. `Lampiro` is a product line with low feature usage for instant-messaging clients [14]. Last but not least, `BerkeleyDB` is a product line for databases [16] of moderate feature usage. The table presented in Figure 8 summarizes: LOC, is the number of lines of code; $|\mathbb{F}|$, is the number of features in the SPL; $|2^{\mathbb{F}_{local}}|$, is the maximum number of configurations of any one method in the SPL; #methods, is the number of methods (with the total number of different method variants, in parentheses); and cc%, is the percentage of methods with conditional compilation (`ifdef`) feature usage. Methods completely encompassed by `ifdefs` are also counted.

The histograms in Figure 9 illustrate the distribution of the number of configurations per method for each of the SPLs. `MobileMedia08` (depicted in Figure 9(b)), for instance, has 157 methods without features, 78 methods with one feature (i.e., $2^1 * 78 = 156$ different method variants), 37 methods with two features (i.e., $2^2 * 37 = 148$ different method variants) etc, and one method with seven features (i.e., $2^7 * 1 = 128$ different method variants). The area shown in the histograms is thus directly proportional to the number of method variants possible. As can be seen, the four benchmark SPLs have qualitatively different feature usage profiles.

Our analyses currently interface with CIDE (Colored IDE) [16] for retrieving the conditional compilation statements. CIDE is a tool that enables developers to annotate feature code using background colors rather than `ifdefs` directives, reducing code pollution and improving comprehensibility. Conceptually, CIDE uses a restricted form of `ifdefs` for which only conjunction of features is permitted.

Our analyses assume that each line never has two parts with different CIDE colorings (i.e., different formulae). This is a fair assumption as lines with multiple configurations could be accomodated by inserting appropriate line breaks.

We have executed the analyses on a 64-bit machine with a Intel® Core™ i7 920 CPU running at a 2.6 GHz frequency with 8 GB of memory and 8MB L2 cache on a Linux Ubuntu 2.6.32-30-generic operating system.

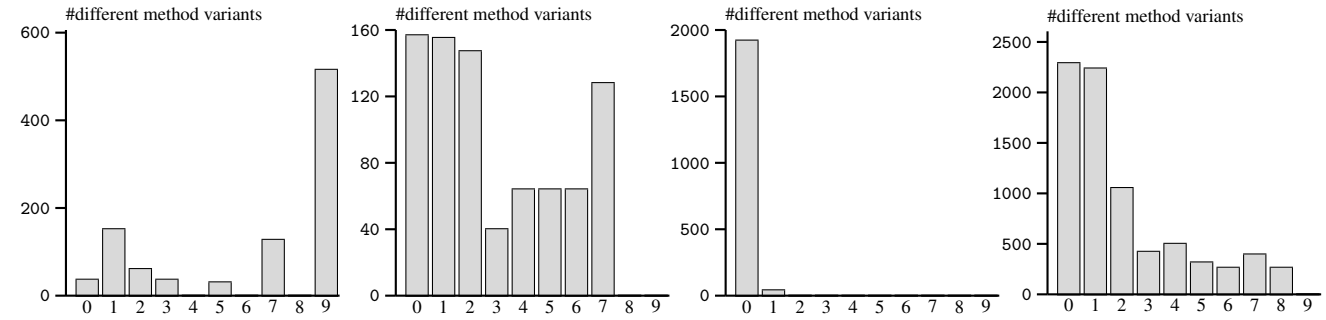
6.2 Results and Discussion

We now present the results[3] obtained from our empirical study. We first present and discuss our results pertaining to the total time, then the performance of the analysis only, and finally, memory consumption. All times given are averages over ten runs with the highest and lowest number removed.

Total Time (including compilation): Figure 10 plots the total time (including compilation) of the reaching definitions analysis on each of our four benchmarks. For the feature-oblivious brute force analysis, $\mathcal{A}1$, the total time is shown in black whereas the feature-sensitive analyses, $\mathcal{A}2$, $\mathcal{A}3$, and $\mathcal{A}4$, are plotted in dark gray, light gray, and white, respectively. All times comprise the tasks outlined in Figure 7. The compilation time given for $\mathcal{A}1$ is the average of the *slowest* configuration to compile ($c = 2^{\mathbb{F}}$) and the *fastest* configuration to compile ($c = \emptyset$) times the number of configurations to be compiled. 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.

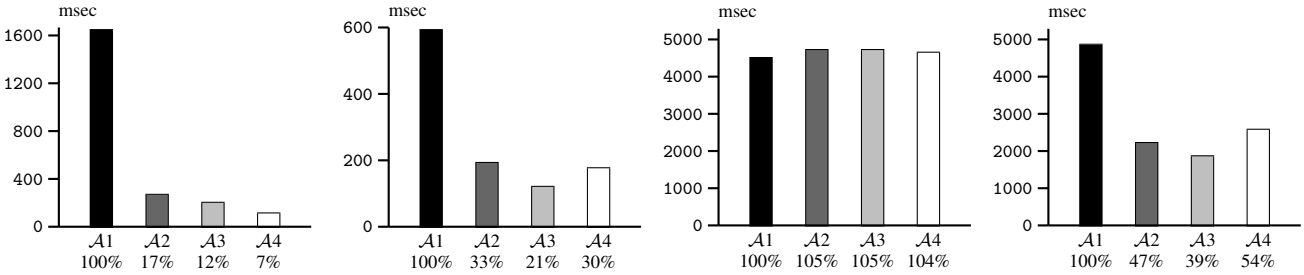
We see that the feature-sensitive analyses, $\mathcal{A}2$, $\mathcal{A}3$, and $\mathcal{A}4$ are all faster than the brute-force approach, $\mathcal{A}1$, except on `Lampiro` where they are all fairly equal. If we take the best gain factors of the feature-sensitive analyses (`Graph PL`: 14; `MobileMedia08`: 4.8; `Lampiro`: 1.0; and `BerkeleyDB`: 2.6), for each benchmark, they give an average speed-up of 5.6 when compared to $\mathcal{A}1$. Continuing the comparison to $\mathcal{A}1$, $\mathcal{A}3$ (in general, the fastest) does the same analysis but takes 12% of the time on `Graph PL`, 21% on `MobileMedia08`, 105% on `Lampiro`, and 39%

[3] All results including equivalence proofs are available at: `http://twiki.cin.ufpe.br/twiki/bin/view/SPG/EmergentAndDFA`



(a) Graph PL (b) MobileMedia08 (c) Lampiro (d) BerkeleyDB

Figure 9. Histogram showing the distribution of number of configurations per methods.



(a) Graph PL (b) MobileMedia08 (c) Lampiro (d) BerkeleyDB

Figure 10. The total time (including compilation) of RD: $\mathcal{A}1$ (black) vs. $\mathcal{A}2$ (dark gray) vs. $\mathcal{A}3$ (light gray) vs. $\mathcal{A}4$ (white).

on `BerkeleyDB` which translates into a gain factors of respectively: 8.3, 4.8, 1.0, and 2.6. So, $\mathcal{A}3$ is anywhere from slightly to slightly more than eight times faster than $\mathcal{A}1$.

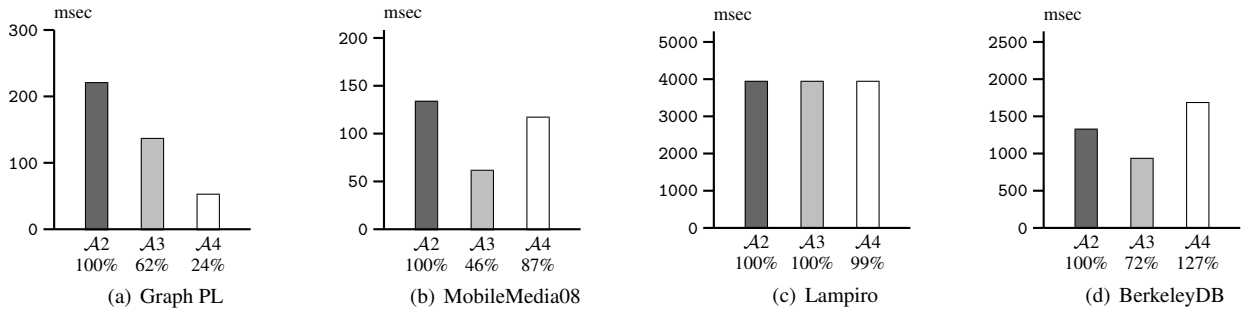
The reason for this is *compilation overhead* that $\mathcal{A}1$ has to "pay" for each different method variant (see Figure 7). When considering $\mathcal{A}2$, $\mathcal{A}3$, and $\mathcal{A}4$, the *compilation* time is an overhead we only have to pay once, even if many analyses are performed. In the following, we will thus focus on the times of the analyses themselves without *compilation* time.

Performance of Analyses (*TIME*): Figure 11 plots the relative difference between the speed of the $\mathcal{A}2$, $\mathcal{A}3$, and $\mathcal{A}4$ feature-sensitive analyses (using reaching definitions). We observe that the $\mathcal{A}3$ is generally the fastest. Compared to $\mathcal{A}2$, it spends only 62% of the time on `GPL`, 46% on `MobileMedia08`, 100% on `Lampiro`, and 72% on `BerkeleyDB`. Figure 12 shows the numbers for the *definite assignments* analysis. Again, $\mathcal{A}3$ is, in general, the fastest. Compared to $\mathcal{A}2$, it spends only 47% of the time on `GPL`, 56% on `MobileMedia08`, and 199% on `Lampiro`, and 65% on `BerkeleyDB`. However, $\mathcal{A}4$ is fastest on GPL for both analyses because there are many sets of configurations that are indistinguishable by `#ifdefs`. In general, $\mathcal{A}4$ is slower than $\mathcal{A}3$ due to many comparisons it has to perform during the analyses to split sets of configurations.

The analysis time is virtually the same for all four analyses, $\mathcal{A}1$, $\mathcal{A}2$, $\mathcal{A}3$, and $\mathcal{A}4$ for `Lampiro`, except in Figure 12(c). This is because it has limited feature usage (only 1.5% of the methods have features and the average number of configurations per method is only 1.02), and thus most of the cost is the time of the analysis itself without any overhead from features. We take this as indication that the overhead of our approaches is almost nothing for SPLs with low feature usage and that it does not matter much which of the feature-sensitive analyses are used in such cases.

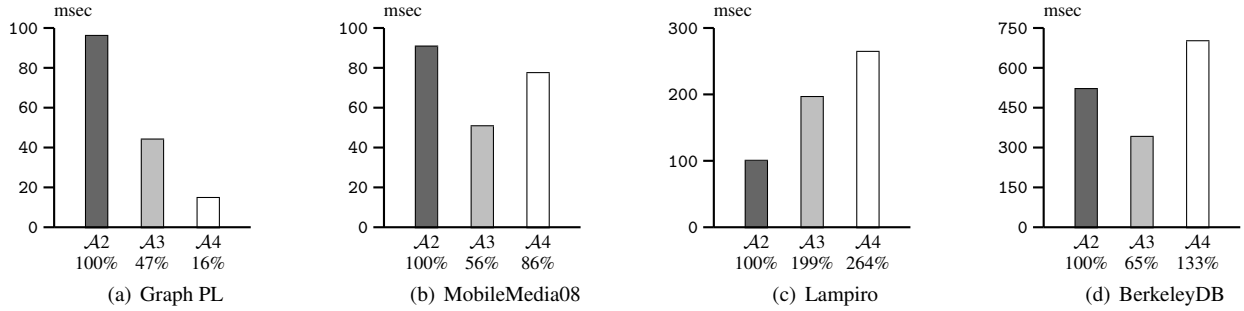
Recall that $\mathcal{A}3$ in principle has to do as many fixed-point iterations as are needed for the slowest converging configuration, unnecessarily reiterating already converged configurations. Our data, however, indicates that this is not a problem in practice. For example, in the reaching definitions analysis, $\mathcal{A}3$ only executes as little as 0.22% more unlifted transfer functions than $\mathcal{A}2$ on `BerkeleyDB`; only 0.05% more on `GPL`; and virtually 0% more on `Lampiro` and `MobileMedia08`.

As for caching, the first data column, *normal cache* of Figure 13, shows the relative difference between the number of cache misses in $\mathcal{A}2$ vs $\mathcal{A}3$. As expected, $\mathcal{A}2$ incurs quite a lot more cache misses than $\mathcal{A}3$, making the former comparatively slower since data has to be re-fetched on every configuration. To further substantiate this claim, we instrumented *both* $\mathcal{A}2$ and $\mathcal{A}3$ with instructions to fill up the L2 cache (traversing an 8MB array) prior to transfer function execution. The second data column, *full cache*, reveals that this indeed hurts $\mathcal{A}2$ more than $\mathcal{A}3$. We take this as evidence that $\mathcal{A}3$ has better cache properties than $\mathcal{A}2$.



(a) Graph PL (b) MobileMedia08 (c) Lampiro (d) BerkeleyDB

Figure 11. The analysis time of *reaching definitions*: $\mathcal{A}2$ (dark gray) vs. $\mathcal{A}3$ (light gray) vs. $\mathcal{A}4$ (white).



(a) Graph PL (b) MobileMedia08 (c) Lampiro (d) BerkeleyDB

Figure 12. The analysis time of *definite assignments*: $\mathcal{A}2$ (dark gray) vs. $\mathcal{A}3$ (light gray) vs. $\mathcal{A}4$ (white).

Benchmark	$\mathcal{A}2 : \mathcal{A}3$ (*normal cache*)	$\mathcal{A}2 : \mathcal{A}3$ (*full cache*)
Graph PL	+ 4 %	+ 13 %
MobileMedia08	+ 36 %	+ 45 %
Lampiro	+ 12 %	+ 18 %
BerkeleyDB	+ 21 %	+ 29 %

Figure 13. Cache misses in $\mathcal{A}2$ vs $\mathcal{A}3$.

We thus have evidence that, in general, $\mathcal{A}3$ seems to be the fastest. If we average the speed ratio over the two analyses on the four benchmarks, $\mathcal{A}3$ outperforms $\mathcal{A}2$ in using only about 56% of the time to calculate the same information.

Memory Consumption of Analyses (*SPACE*): Figure 14 lists the space consumption by the maximum memory consuming method (wrt. $\mathcal{A}3$). Our experimental data confirms that $\mathcal{A}3$ requires almost $|2^{\mathbb{F}_{\text{local}}}|$ times more memory than $\mathcal{A}2$ since it has to keep all configurations in memory during its fixed-point computation. This does, however, not appear to be a problem in practice for intraprocedural analysis as it only needs to keep data for one method in memory at a time. Indeed, this has not been a problem on any of our four benchmarks. The shared analysis $\mathcal{A}4$ may reduce space usage anywhere between a factor of one to 15, depending on the SPL.

7. 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 [5] or to optimize frequently executed paths [2]. Earlier, disabling infeasible dead statements has been exploited to improve the precision of constant propagation [29] by essentially running a dead-code analysis capable of tagging statements as executable or non-executable during constant propagation analysis.

Predicated dataflow analysis [22] 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 $\mathcal{A}3$ and $\mathcal{A}4$ 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: 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 [11]. 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 au-

Benchmark	Max. memory consuming method	$\lvert 2^{\mathbb{F}_{\text{local}}} \rvert$	$\mathcal{A}2$	$\mathcal{A}3$ ($\mathcal{A}2$:$\mathcal{A}3$)	$\mathcal{A}4$ ($\mathcal{A}3$:$\mathcal{A}4$)
Graph PL	Vertex.display()	$2^9 = 512$	37 KB	9.9 MB (1:281)	1.4 MB (7.2:1)
MobileMedia08	MediaController.handleCommand()	$2^6 = 64$	93 KB	4.8 MB (1:53)	2.5 MB (1.9:1)
Lampiro	InfTree.clinit()	$2^0 = 1$	12 MB	12 MB (1:1)	12 MB (1:1)
BerkeleyDB	DbRunAction.main()	$2^7 = 128$	212 KB	20 MB (1:96)	1.3 MB (15:1)

Figure 14. Maximum memory consumption of lattice information during analysis ($\mathcal{A}2$ vs. $\mathcal{A}3$ vs. $\mathcal{A}4$).

tomatically make any conventional dataflow analysis able to analyze SPLs 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.

We recently proposed the concept of emergent interfaces [26]. 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.

Lifting for SPLs: Researchers already lifted automated analysis and processing such as for parsing [17], model checking [6], monitoring [19], type checking [3], and verification [4, 25]. Kaestner et al. [17] provides 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 analysis $\mathcal{A}4$, where we lazily split sets of configurations. However, we do not perform join. On the one hand, we could join lattices of different configurations that are exactly the same in favor of memory usage. On the other hand, we would increase the performance overhead, since we need to verify the equality of lattices for each statement and potentially for many configurations.

Classen et al. [6] 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.

Safe composition: 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 [3, 15]. Safe composition may help in finding problems like undeclared variables. 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).

8. Conclusion

In this paper, we presented three approaches for taking any standard one-program dataflow analysis and automatically lifting it into a feature-sensitive analysis capable of analyzing all configurations of an SPL. To evaluate these approaches, we took two intraprocedural dataflow analyses and made them feature-sensitive. Experimental evaluation shows that the feature-sensitive approaches are faster than the naive brute-force approach. For SPLs with low feature usage, they are only slightly faster than the naive approach. For SPLs with high feature usage, the simultaneous feature-sensitive analysis ($\mathcal{A}3$), in particular, is up to more than eight times faster than the existing alternative ($\mathcal{A}1$).

We also conclude that our three approaches have different performance and memory consumption characteristics. The simultaneous feature-sensitive analysis ($\mathcal{A}3$) is, in general, the fastest. On the other hand, in terms of memory consumption, $\mathcal{A}4$ performs better than $\mathcal{A}3$. However, this does not show up as a problem in practice for intraprocedural analysis on the benchmarks we used.

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 for the comments that helped to improve this paper.

References

[1] 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)*, pages 243–254, Charlottesville, Virginia, USA, 2009. ACM.

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

[2] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *Programming Language Design and Implementation (PLDI'98)*, pages 72–84, Montreal, Canada, 1998.

[3] S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17:251–300, September 2010.

[4] 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, November 2011. IEEE Computer Society.

[5] T. Ball and S. K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *PASTE'01*, pages 97–103, Snowbird, Utah, USA.

[6] 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)*, pages 335–344, Cape Town, South Africa, 2010. ACM.

[7] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[8] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28:1146–1170, December 2002.

[9] 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)*, pages 261–270, Leipzig, Germany, 2008. ACM.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] 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)*, pages 57–68, Porto de Galinhas, Brazil, 2011. ACM.

[12] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[14] C. Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, Germany, May 2010.

[15] 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)*, pages 258–267, L'Aquila, Italy, 2008.

[16] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 311–320, Leipzig, Germany, 2008. ACM.

[17] 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)*, pages 805–824, Portland, OR, USA, 2011. ACM.

[18] 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)*, pages 194–206, Boston, Massachusetts, 1973. ACM.

[19] 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)*, volume 6418 of *LNCS*, Malta, November 2010. Springer.

[20] 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)*, pages 49–57, Los Alamitos, CA, USA, 1994. IEEE Computer.

[21] 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)*, pages 105–114, Cape Town, South Africa, 2010. ACM.

[22] 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)*, pages 204–211, Melbourne, Australia, 1998. ACM.

[23] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Secaucus, USA, 1999.

[24] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[25] 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)*, pages 347–350, L'Aquila, Italy, 2008. IEEE Computer Society.

[26] 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)*, pages 11–18, Reno, NV, USA, 2010.

[27] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Proceedings of the Usenix Summer 1992 Technical Conference*, pages 185–198, Berkeley, CA, USA, June 1992. Usenix Association.

[28] 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)*, pages 13–. IBM Press, 1999.

[29] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13:181–210, April 1991.