Combining Concern Input with Program Analysis for Bloat Detection

Suparna Bhattacharya
IBM Research
bsuparna@in.ibm.com

K. Gopinath
Indian Institute of Science
gopi@csa.iisc.ernet.in

Mangala Gowri Nanda
IBM Research
mgowri@in.ibm.com

Abstract

Framework based software tends to get bloated by accumulating optional features or concerns just-in-case they are needed. The good news is that such feature bloat need not always cause runtime execution bloat. The bad news is that often enough, only a few statements from an optional concern may cause execution bloat that may result in as much as 50% runtime overhead.

We present a novel technique to analyze the connection between optional concerns and the potential sources of execution bloat induced by them. Our analysis automatically answers questions such as (1) whether a given set of optional concerns could lead to execution bloat and (2) which particular statements are the likely sources of bloat when those concerns are not required. The technique combines coarse grain concern input from an external source with a fine-grained static analysis. Our experimental evaluation highlights the effectiveness of such concern augmented program analysis in execution bloat assessment of ten programs.

Categories and Subject Descriptors F.2.3 [*Logic and Meaning of Programs*]: Semantics of Programming Languages—Program Analysis; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

Keywords software bloat, program concerns, feature oriented programming

1. Introduction

The abundance of redeployable components and frameworks has greatly eased software development, by making it possible to rapidly construct software applications with extensive capabilities. An unintended and almost inevitable side effect of framework based software, however, is the buildup of excess or largely optional features due to just-in-case programming at each framework layer through the reuse of over-general components. This can result in *runtime execution bloat*, defined as the runtime resource overhead induced by the presence of excess function and objects, e.g. excessive conditions checks, costly data structures with low utility [48] and heavily nested data transformations [32, 33].

For example, in their study of information flow patterns in a stock brokerage framework [32], Mitchell et. al found that as many as 58 transformations were executed just for converting a date field in a SOAP message to a Java business object. Of these, 18 transformations resulted from decimal number specific processing being needlessly applied to whole number sub-fields - a runtime overhead that was incurred repeatedly for every request processed. This is an example of *execution bloat* arising due to the use of a (general) component (`DecimalFormat` parsing class) which has a *program concern*[1] (i.e. decimal handling) that is *in excess* in this case where it is used to extract fields which are always whole numbers.

Such a conceptual analysis of runtime execution bloat has eluded automation till date. Existing approaches for Java bloat analysis lack high level information about the functional intent of code, insight that is required in order to identify the excess functionality.

1.1 The Problem

Does the presence of an excess concern (due to feature bloat in an underlying component) always cause execution bloat? If not, what are the potential sources of execution bloat in the concern?

Consider the example shown in Figure 1[2]. The code shows a simple `Buffer` class which provides a logged restorable buffer, an example from [29]. The `logit()` method is included in order to support the concern *log*, which logs the buffer's state after each operation. When the class is

OOPSLA '13, .
Copyright © 2013 ACM 978-1-4503-2374-1/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509136.2509522

[1] Concerns are features, design idioms or other conceptual considerations that can impact the implementation of a program [41].

[2] The temporary variables, tmp3 in get(), tmp in set() and tmp2 in restore() are generated by our bytecode generator.

```
1:public class Buffer {
2:    int buf = 0;
3:    int back = 0;
4:    public Buffer(int x) {
5:       this.buf = x;
6:    }
7:    void logit() {
8:       System.out.println("LOG: buf = " + buf);
9:       System.out.println("LOG: back = " + back);
10:   }
11:   int get() {
12:      int tmp3 = this.buf;
13:      logit();                    ← interaction(LOG, BUFFER)
14:      return tmp3;
15:   }
16:   void set(int x) {
17:      int tmp = this.buf;         ← interaction(BUFFER, RESTORE)
18:      back = tmp;                 ← interaction(BUFFER, RESTORE)
19:      this.buf = x;
20:      logit();                    ← interaction(LOG, BUFFER)
21:   }
22:   void restore() {
23:      int tmp2 = this.back;       ← interaction(BUFFER, RESTORE)
24:      logit();                    ← interaction(LOG, RESTORE)
25:      this.buf = tmp2;            ← interaction(BUFFER, RESTORE)
26:   }
```

Figure 1. Example: Logged Restorable Buffer from [24, 29]. LOG and RESTORE are optional features. The statements needed to make a given combination of features X and Y work correctly together are said to constitute the resolution of their feature interaction [29], marked above as `interaction(X, Y)`.

reused in a scenario where the buffer need not be logged, *log* is an excess concern but the `logit()` method is called nevertheless, incurring a heavy runtime overhead.

Further, the code for the function `void restore()` and all code that applies to the variable `back` are statements added to support the *restore* concern which allows the contents of a buffer to be restored to its previous value when required. As the `Buffer` class can be reused in scenarios where the buffer need not be restorable, support for *restore* may be an optional concern. Under such usage scenarios, `Buffer` is an overgeneral component (class) and *restore* is an excess concern.

However, we notice that the `restore()` method by itself does not induce a runtime execution overhead in this situation as it would not be called. The `get()` functionality of the buffer is also unaffected by the excess concern *restore*. On the other hand, the `set()` method does incur the runtime overhead of saving the previous value. The statements `tmp = this.buf; back = tmp;` in this method, therefore, are a potential source of execution bloat in this usage scenario. Secondly, the extra memory allocation and initialization of the `back` field, i.e. the statement `int back = 0;` is another potential source of bloat due to the optional *restore* concern.

In order to automatically detect such statements that are sources of execution bloat due to excess concerns, the challenging issues that need to be satisfactorily addressed are:

1. *Is it feasible to enrich program analysis approaches for bloat detection with just enough high level intent information needed for deriving such insights ?*
2. *How do we obtain this information in practice ?*

We observe that the current state of the art in software engineering research already provides a rich variety of techniques for identifying, locating, analyzing, annotating and separating software concerns and features.

Feature oriented software development models [2, 38], software product line engineering [37] and aspect oriented programming [23] provide disciplined mechanisms to construct software with explicitly built-in concern assignment information[3]. A specialized variant can be generated for any combination of features allowed by the feature model rules of the program. For such software, we can identify bloat by using the concern assignment information and the rules that specify optional features, provided the granularity of features is defined at a sufficiently fine and detailed level.

As a majority of software components are not created in this fashion, there has been substantial research on semi-automatic techniques to aid concern identification and analysis [41], addressing closely related problems such as feature location [14, 42], aspect mining [5, 22, 30, 44], code search, concept assignment [7, 9] and change impact analysis [4, 17]. For instance, a concern location scheme may follow a query based approach (e.g a directed search from given seed or pattern [42]) while an aspect mining algorithm may employ a generative approach [30] (e.g. discover structural patterns indicative of cross-cutting concerns). Both types of analysis can be static [18] and/or dynamic [12] and exploit additional artifacts beyond program source and executable traces, often combining traditional program analysis with information retrieval natural language processing, formal concept analysis and some form of human input [16, 39, 44, 52, 53].

However, in practice, the nature of concern information that is easily recoverable may be *incomplete* or too *coarse grained* for bloat detection, although possibly good enough for program understanding and maintenance. Thus, even when some level of concern information is available, it is not clear how exactly such extracted concerns should be systematically used to automate the analysis of bloat.

1.2 Preview of the Solution

The basic intuition behind our analysis is as follows: some statements corresponding to each feature can represent structurally intertwined code between multiple features. Such a statement is a candidate source that contributes to potential execution bloat when

- it corresponds to an optional feature, but
- occurs in a method that belongs to an essential feature.

In order to identify these statements, we perform the following steps:

[3] *Concern or feature* ***assignments*** associate concerns and their properties, i.e. concern intent, with the source code components, methods or even statements where they are implemented, i.e. concern extent.

- Reverse engineer each potential feature extension based on program dependence assuming that every method could be a potential feature (computing microslices).
- Build a graph that connects the uses and the definitions in the microslices (computing the Microslice Interaction Graph (MSIG)).
- Enrich the graph with concern information (creating a Concern Augmented Microslice Interaction Graph)
  - Apply concern analysis information to group methods in terms of actual concern properties (optional or mandatory).
  - Overlay the MSIG with information about the optionality or the mandatoriness of each feature to generate a Concern Augmented MSIG (the CA_{MSIG}).
- Traverse the CA_{MSIG}, applying heuristic rules in order to determine which feature extensions (microslices) are in excess and hence potential sources of bloat.

Computing microslices Starting with every input field variable in a method, we compute a forward intra-procedural slice and starting with every output field variable we compute a backward intra-procedural slice. These slices are then partitioned into microslices such that the input and output combination for each slice is unique.

EXAMPLE 1. *In Fig 1, the* `restore()` *method contains the microslices:*

$$\langle restore(), in = \{back\}, out = \{buf\}, \mu slice = \{23, 25\}\rangle$$

$$\langle restore(), in = \{logit()\}, out = \{\}, \mu slice = \{24\}\rangle$$

under inputs, $\phi_{in} = \{back, logit()\}$*, outputs,* $\phi_{out} = \{buf\}$

Similarly, in Fig 1, the `set()` *method contains the microslices:*

$$\langle set(), in = \{buf\}, out = \{back\}, \mu slice = \{17, 18\}\rangle$$

$$\langle set(), in = \{\}, out = \{buf\}, \mu slice = \{19\}\rangle$$

$$\langle set(), in = \{logit()\}, out = \{\}, \mu slice = \{20\}\rangle$$

under inputs, $\phi_{in} = \{buf, logit()\}$*, outputs,* $\phi_{out} = \{back\}$

Suppose line 19 in the program is replaced by `this.buf = tmp + x;`*. Now line 17 affects two heap variables,* `buf` *and* `back`*, unlike line 18 which only affects* `back`*. Thus lines 17 and 18 would no longer belong to the same microslice. Hence, the microslices in this case would be:*

$$\langle set(), in = \{buf\}, out = \{buf, back\}, \mu slice = \{17\}\rangle$$

$$\langle set(), in = \{buf\}, out = \{back\}, \mu slice = \{18\}\rangle$$

$$\langle set(), in = \{buf\}, out = \{buf\}, \mu slice = \{19\}\rangle$$

$$\langle set(), in = \{logit()\}, out = \{\}, \mu slice = \{20\}\rangle$$

Thus for each unique combination of input and output data, we compute a unique microslice, and each statement belongs to exactly one microslice. Complete details of computing microslices can be found in Section 3.

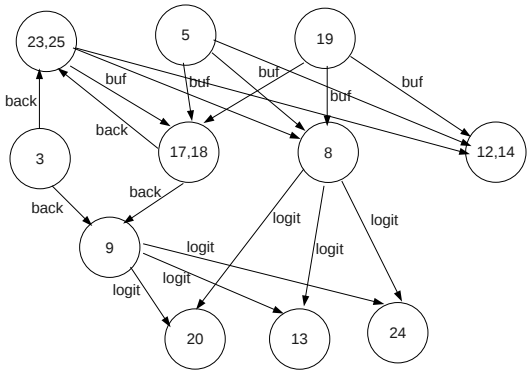

Figure 2. Microslice Interaction Graph for Fig 1. Nodes represent microslices. The arrows represent potential USE interactions (where the destination node USEs the source node's feature) and are labeled with the dependence criterion responsible for the interaction (e.g. shared fields like `buf` and `back`, or called methods e.g. `logit()`).

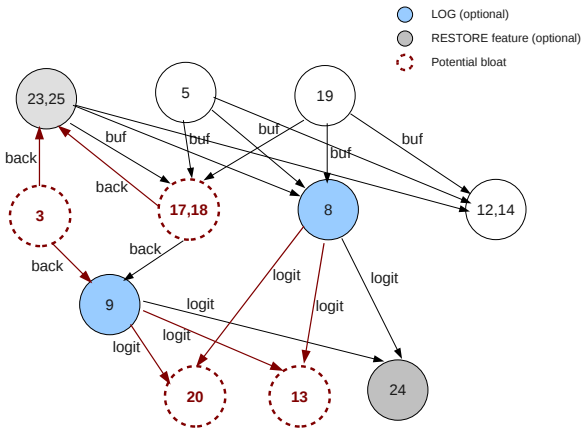

Figure 3. Concern Augmented Microslice Interaction Graph corresponding to Fig 2. The shaded nodes represent microslices in methods belonging to optional features (LOG: blue and RESTORE: gray).

Computing the Microslice Interaction Graph Potential dependencies between features in different methods can occur when

- the code for a feature in one method uses a field and
- the code for another feature in a second method defines (updates) the field.

This helps determine the related features.

For each field or object that is shared between methods, we find the methods where the field is used as an input and identify the microslices that are in a *forward slice* of that input criterion. Likewise, we find the methods where the same field or object is an output and identify the microslices that are in a *backward slice* affecting that output criterion. Now we create a directed edge from each microslice node in the second set to each microslice node in the first set.

This creates a directed graph where a (target node) microslice is a descendant of the microslices it might require (e.g. parent nodes generate state that is stored in fields which might be later be used by statements in the child nodes). A leaf node has no outgoing edges. We term this the Microslice Interaction Graph (MSIG or $\mu slice$).

EXAMPLE 2. *In Fig 2, for the microslices represented by:*

$\langle set(), in = \{buf\}, out = \{back\}, \mu slice = \{17, 18\}\rangle$

$\langle restore(), in = \{back\}, out = \{buf\}, \mu slice = \{23, 25\}\rangle$

The $\mu slice = \{23, 25\}$ *has an incoming edge from the* $\mu slice = \{17, 18\}$ *labeled "back" and an outgoing edge labeled "buf".*

The complete MSIG for the example in Fig 1 can be found in Fig 2 and details of generating the MSIG can be found in Section 3.

Computing the Concern Augmented Microslice Interaction Graph Having computed the MSIG, we now need to augment this graph with Concern related information to generate what we term the Concern Augmented MSIG (CA_{MSIG}) graph.

Let us assume that we are given a concern-based abstraction that partitions the concerns of a component into two groups - mandatory (always required) and optional. The exact mechanism used to realize this grouping may range from a purely manual assignment by an expert to automated classification rules based on an analysis of representative client programs. We also have information about which methods are introduced by these concerns, for example using a concern analysis tool.

We perform a Concern Augmented program analysis using this information and the statically computed MSIG to identify candidate excess statements that cause execution bloat when none of the specified optional concerns are required. We apply some heuristic rules. For example, a microslice in the CA_{MSIG}, that is mandatory, but is used solely by nodes along CA_{MSIG} paths that eventually contain an optional node, is considered to be potential bloat.

EXAMPLE 3. *Consider the* $\mu slice = \{17, 18\}$ *in Fig 3. If we go forward along all outgoing edges, then we always reach an optional node. Thus, this microslice is marked as potential bloat, for the given optional concerns.*

The complete Concern Augmented MSIG for the example in Figure 1 can be found in Figure 3 and details of generating the Concern Augmented MSIG can be found in Section 4.

1.3 Contributions

There are several potential applications where a systematic approach for integrating concern information could be useful, e.g. in enhancing opportunities for code specialization, speculative optimization and performance summarization not just in product-lines but also in non-product line software. In this paper, we develop and explore the use of concern information in assessing execution bloat propensity of optional concerns implemented by a component (where a component could be a class, a package or a framework library). The contributions of this paper are:

- An approach to utilize externally supplied information about program concerns and their properties, where available, by creating *concern partitions* derived from this information to enrich program analysis techniques for traditionally developed software.
- A novel concern augmented static analysis of an optional concern's likelihood for execution bloat in a given software component.
- Our experimental evaluation indicates that our analysis
  - correctly detects sources of bloat with a precision varying from 69% to 100%.
  - detects all sources of bloat due to a given set of concerns with a recall varying from 54% to 100%.

  By and large the precision and recall is close to 100%, but there are some pathological cases where these values drop. In addition, we also observed that removing optional concern related bloat can result in more than 50% improvement in performance even for the large benchmarks.

2. The Concern Augmented Program Analysis (CAPA) Framework

2.1 Working Definitions

Some working definitions of terms that we use:

Concern: A concern is any consideration that can impact the implementation of a software program. According to [41] it could represent "anything a stakeholder may want to consider as a conceptual unit, including features, non-functional requirements, design idioms, and implementation mechanisms".

We use the terms concern and feature interchangeably[4] in this paper. The implementation of a concern in source code, i.e. its extent, may be spread across multiple underlying program units (scattering) and the same program unit may contribute to multiple concerns (tangling).

We introduce the term concern partitioning to describe the task of partitioning the statements of a program into groups that implement the same concern or a set of concerns

[4] as we adopt useful concepts from both concern location and feature oriented software development

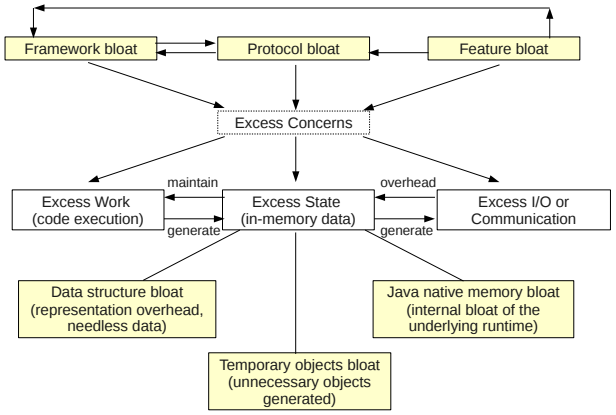

Figure 4. Relationship between different types of bloat and their runtime manifestations

related by their properties (i.e. by concern intent). There may be many ways to partition the program depending on the concerns of interest.

Concern Partition: A concern partition divides the statements of a program into groups which implement a set of concerns that either share a common property or are related in some other manner (by a partitioning rule).

Optional / Excess Concern: A concern that is not required (i.e. is in excess) for the given usage scenario.

Mandatory Concern: A concern that is essential for the given usage scenario.

2.2 Execution Bloat in the Context of Software Concerns

Runtime bloat can manifest in intricate ways in large framework based applications. In Fig 4 we depict an example of the relationship between different forms in which bloat is manifested at runtime as observed at different levels of abstraction.

We are interested in the sources of execution bloat induced by the presence of unutilized (hence excess) optional concerns – this type of bloat is a typical consequence of the high level of generality supported by framework based components. The analysis is interesting because the presence of an excess concern in itself may not necessarily cause a significant runtime overhead as different concerns and different implementations may have a different predisposition for execution bloat.

In Fig 5 the code to canonicalize data from big endian to little endian (line 11-15, including the cost of parsing and formatting from string to integer and back, intermediate objects generated and checking `isBigE`) implements a software concern that may be unnecessary in deployment situations when all systems or data sources involved are uniformly big endian. Note that if the method `big2LittleEndian()` is labeled as an optional concern, we need our analysis to deduce that not only line 13, but also lines 11, 12 and 14 are potential sources of bloat associated with this optional concern.

```
1:      public string applyConvertors(String s, boolean isBigE) {
2-10:     ...
11:         if (isBigE) {
12:             tmp1 = str2int(s);
13:             tmp2 = big2LittleEndian(tmp1);
14:             result = int2str(tmp2);
15:         }
16-30:      ...
30:     }
```

Figure 5. Big Endian to Little Endian Transformation

Although this is a highly simplified example, the resource overheads of bloat could be substantial when a sequence of such data conversions occur repeatedly in a loop (e.g. iteratively over a collection of data elements in each input request in a server side application that receives a high volume of request transactions). For instance, recall the example from [32] described in the introduction where Mitchell et al. found that the use of a general purpose DecimalFormat class results in 3 needless transformations for each sub-field of a date (month, day, year, hour, minute and second) [32]. This alone amounts to 18 excess transformations to convert a single date field from a SOAP message to a Java business object for every input transaction request.

2.2.1 When do excess concerns cause execution bloat?

Execution bloat can only occur when the excess concerns induce an extra execution cost (e.g. via extra statements or data objects) *during the usage of essential concerns*.

In practice, the code for excess concerns can get tangled with the code for essential features or interact with essential concerns via shared data structures. This causes some excess code statements to be executed when the client program is run. The execution cost of these statements is the execution bloat incurred by the client program when it runs.

Consider the example in Fig 1. This code has three concerns named BUFFER, LOG and RESTORE. Let us assume that in the initial concern partition, LOG includes statements in the `logit()` method, RESTORE includes statements in the `restore()` method and BUFFER includes statements in the remaining methods.

A client program that uses the Buffer class just for the BUFFER feature; does not require the functionality provided by the RESTORE or LOG concerns. In this case, BUFFER is an essential concern, while RESTORE and LOG are optional concerns.

The statements `tmp = this.buf; back = tmp;` (lines 17, 18) also correspond to the RESTORE concern, but are executed by the client program as they are tangled with code for the BUFFER feature. Thus they contribute to execution bloat. Similarly, the `logit()` method which is invoked through statements in the `get()` and `set()` methods (lines 13 and 20) also contribute towards bloat.

The problem of finding statements that might contribute to execution bloat thus involves

- finding those statements belonging to excess concerns
- that are structurally intertwined with executable code statements for essential concerns

To identify such statements automatically, we observe for instance, that statements in a method corresponding to an essential concern may write data to some heap variables, which are later read only by methods corresponding to optional concerns.

EXAMPLE 4. *Line 18 in the* `set()` *method belonging to the BUFFER concern updates the variable* `back`*, which is read only by the RESTORE concern. Further, line 17 defines a value that is used solely by line 18 to update* `back`*. Therefore such statements are labeled as contributors to an optional concern.*

One way to detect these statements is to compute the intra-procedural backward slices of such output fields (e.g `back`) in the method corresponding to an essential concern (e.g. `set()`). Likewise, we require intra-procedural forward slices when statements in a method assigned to an essential concern read (input) fields whose values are written only by methods corresponding to optional concern. However, the slices computed can contain statements that also affect other fields, some of which might not be optional. Hence we need to further partition the statements in each slice in terms of the other fields they affect. We develop a technique called microslicing to generalize this procedure for fine-grained partitioning, as a basis for developing bloat detection heuristics.

2.3 CAPA overview

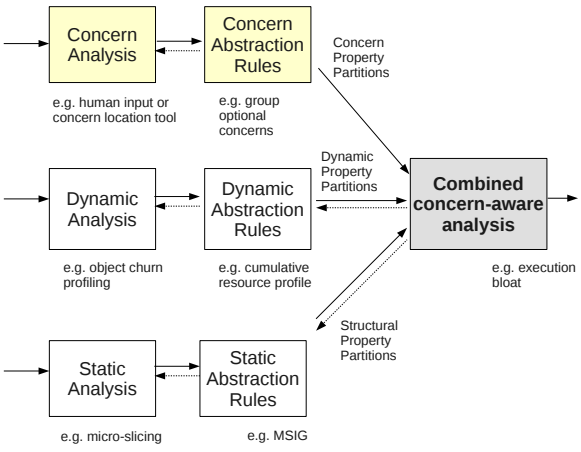

Figure 6. Concern Augmented Program Analysis

Fig 6 shows an example of a concern augmented program analysis that combines static and dynamic program analysis with concern partition analysis. Here, each different type of analysis (static, dynamic, concern), can implement multiple partitioning functions and analysis functions.

For example, the microslicing static analysis described in the next section can be viewed as a partitioning function that partitions the statements of a program into microslices and an analysis function that computes the structural interactions associated with each part.

The concern analysis step creates partitions in terms of method level concern assignment information and concern abstraction rules which group concerns based on user input. Concern related information may be available from a feature model of the program if one exists.

In the last block shown in the figure the results of the different types of analyses are combined in a task specific manner. The analysis may be expressible as a datalog formula over analysis functions and partitioning predicates. In Section 4 we will illustrate some specific rules for bloat detection.

In this paper, we focus on a concern augmented static analysis for analyzing whether a given implementation of optional concerns is prone to execution bloat. For this analysis, only the static analysis and concern analysis blocks in Fig 6 are relevant.

3. Microslicer

We now describe how to build a novel static analysis based representation of a program or component[5] that is useful for bloat detection. We call this analysis *microslicing* as it breaks up each method into many fine (micro) slices which represent the smallest incremental units within the method that could possibly be assigned to different features or unique combinations of features. Each microslice can contain a non-contiguous set of statements.

3.1 Preliminary static analysis

Our implementation is built as an extension of the same underlying static analysis infrastructure used in [6, 35].

Escape Analysis To locate data structure fields that might be shared across methods (and could cause structural interactions between the concerns assigned to those methods), we rely on escape analysis. After constructing the control-flow graph of each method, our underlying tool [6, 35] performs flow- and context-sensitive pointer analysis and escape analysis.

Recall that for each method, escape analysis computes the following:

- The *formal-in* set for a method M contains the set of formal parameters. The implicit *this* parameter (in non-static methods) is also a formal-in.
- The *formal-out* set for a non-void method M contains a single parameter R, the designated return value. The *formal-out* set is empty for a void method.

[5] e.g a Java library package

- The *escape-in* set for a method M contains direct and indirect fields of the formal parameters of M that are used, before possibly being defined, in M. These represent the upwards-exposed uses in M.
- The *escape-out* set for M contains direct or indirect fields of the formal parameters of M and the return value of M that are defined in M.

The algorithm associates escape-in and formal-in sets with the $Entry$ node of the CFG, and escape-out and formal-out sets with the $Exit$ node of the CFG.

Forward and Backward Slicing Our microslicer employs a combination of intra-procedural forward and backward slicing of the *escape-in* and *escape-out* parameters identified for each method of interest. Note that although the slices we compute are intra-procedural, the underlying pointer analysis is inter-procedural and aliasing is accounted for when computing control and data dependencies.

An intra-procedural forward slice $ForwardSlice(M, S_i)$ consists of all CFG nodes of M in the transitive closure of def-use sets starting from the node S_i, i.e. all nodes that are eventually data dependent on S_i (and hence affected by S_i).

An intra-procedural backward slice $BackwardSlice(M, S_i)$ consists of all CFG nodes of M in the transitive closure of data and control dependencies of the node S_i, i.e. all nodes that affect S_i through a direct or indirect data or control dependence.

3.2 Microslicing

Our microslicing analysis is based on the following:

1. Each method in the component is introduced[6] by a potential feature. (The information about which feature introduced the method is not used in this static analysis stage but is part of separately supplied concern information as described in Section 4).
2. Statements inside each method can represent structurally intertwined code between multiple features (and constitute a resolution of this feature interaction, i.e. the extra code needed to make the features work correctly together)

Continuing with our working example from Fig 1, BUFFER is actually a group of three method level features, INIT (`init()`[7]), SET (`set()`) and GET (`get()`). Observe that lines 17, 18 and 23, 25 represent the *resolution of a feature interaction* between BUFFER and RESTORE, while lines 13, 20 represent the resolution of feature interactions between LOG and BUFFER and line 24 between LOG and RESTORE as shown in Fig 1 by the lines marked as `interaction()`. Here, `interaction(X, Y)` denotes the resolution of a feature interaction been two features X and Y, or statements constituting extra code that is needed to make sure that features X and Y work correctly together [29].

We use static analysis to perform an automatic decomposition of a program according to this model based purely on structural dependencies (e.g. data and control dependence). To achieve this, we further assume that *each feature-specific statement either uses an additional* ***input field*** *or* ***method*** *or affects an additional* ***output field*** (where input fields correspond to *escape-in* and output fields to *escape-out* or *formal-out* nodes for a Java method). Potential interactions between feature-specific statements in different methods can occur when a statement in one method uses a field and a statement in another method defines (updates) the field.

Therefore an automatic decomposition can be performed using a combination of intra-procedural forward and backward slicing to partition the statements of a method into candidate feature-specific statements (or *microslices*, as we call them).

The slicing criteria A forward slicing criterion is an input variable or field directly used within the body of the method, and a backward slicing criteria is an output variable or field that is directly updated within the body of the method. In addition to input and output data, we also introduce a forward (input) criterion for each method call statement, and a backward (output) slice criterion for the method output.

EXAMPLE 5. *The* `set()` *method in Fig 1 has two input (forward slicing) criteria, the field* `buf` *and the method call to* `logit()`*; here the field* `buf` *is also an output (backward slicing) criteria as it is updated by the method, as is the field* `back`*.*

The case where a method call statement is a slicing criterion requires special treatment. Our forward slice for the callsite statement includes all statements in the caller that use the result of the method call

EXAMPLE 6. *Line 14 uses the result of the* `big2LittleEndian()` *call in Fig 5.*

In addition, we include in the slice, statements that solely contribute to setting up the method parameters with no other impact.

EXAMPLE 7. *Lines 11 and 12 in the* `big2LittleEndian` *example in Fig 5.*

The rationale is that all of these statements exist to correctly support the use of the feature provided by the method called and hence constitute a resolution of this feature interaction.

Microslices Consider a method with two input criteria and two output criteria. Let us label the intra-procedural forward slices of the input variables F1, F2 and the intra-procedural backward slices of the output variables B1, B2. Then, the statements may be partitioned as follows: F1, F1F2, F1B1, F1B2, F2, F2B1, F2B2, B1, B1B2, B2, F1F2B1, F1F2B2, F1B1B2, F2B1B2, F1F2B1B2. Here, F1B2 represents the

[6] while we assume that multiple features could add code to a method, only one of these features can introduce the method in the component; this is the feature which defines the method in the first place.

[7] constructor including field initialization statements in the Buffer class

set of statements that are only in the slices for both F1 and B2 (and not included in either F2 or B1).

The non-empty statement sets among these are "microslices" and correspond to candidate feature-specific statements (or the resolution of potential feature interactions).

DEFINITION 1. *Let ϕ_{in}^M and ϕ_{out}^M represent the set of input and output slicing criteria of a method M respectively. The set of microslices μ^M of M consists of all tuples $\mu = \langle M, \phi_f, \phi_b, S\rangle$ where $\phi_f \subseteq \phi_{in}$, $\phi_b \subseteq \phi_{out}$ and S is a non-empty set of statements contained in the microslice, where a statement $s \in S$ if and only if all the following conditions hold:*
s belongs to method M
$s \in ForwardSlice(M, \phi)\ \forall \phi \in \phi_f$
$s \in BackwardSlice(M, \phi)\ \forall \phi \in \phi_b$
$s \notin ForwardSlice(M, \phi)\ \forall \phi \notin \phi_f$
$s \notin BackwardSlice(M, \phi)\ \forall \phi \notin \phi_b$

The above conditions ensure that the microslices do not overlap and that all the statements in a single microslice use and update the same set of heap variables and no others. This set of heap variables is a subset of the input and output criteria. If a method contains N_s statements with N_f input criteria and N_b output criteria, at most $min(N_s, 2^{N_f+N_b} - 1)$ microslices could exist. This is because there are $2^{N_f+N_b} - 1$ such possible subsets, the microslices must be non-overlapping and each microslice should contain at least one statement. The actual number of microslices is typically lower as can be seen in Example 1.

3.3 MicroSlice Interaction Graph (MSIG)

Next we proceed to find associated (interacting) statements in other methods for each of these candidate feature statements by building a cross-method MicroSlice Interaction Graph (MSIG) as follows:

For each field or object that is shared between methods, we find the methods where the field is used as an input and identify the microslices that are in a *forward slice* of that input criterion. Likewise, we find the methods where the same field or object is an output and identify the microslices that are in a *backward slice* affecting that output criterion. Now we create a directed edge from each microslice node in the second set to each microslice node in the first set.

This creates a directed graph where a (target node) microslice is a descendant of the microslices it might require (e.g. parent nodes generate state that is stored in fields which might later be used by statements in the child nodes). A leaf node has no outgoing edges. Notice that this graph captures potential dependencies regardless of interprocedural calling context since we do not assume information during the static analysis phase about the client programs or contexts in which the component is deployed.

DEFINITION 2. *A Microslice Interaction Graph (MSIG) of a component is a directed graph where the* nodes *are microslices and there is a directed edge from node $\mu_i = \langle M_i, \phi_{f_i}, \phi_{b_i}, S_i\rangle$ to node $\mu_j = \langle M_j, \phi_{f_j}, \phi_{b_j}, S_j\rangle$* iff *$\exists \phi_i \in \phi_{b_i}, \phi_j \in \phi_{f_j}$ such that ϕ_i and ϕ_j refer to the same field, variable or method name and $M_i \neq M_j$.*

Fig 2 represents the MSIG for the example code in Fig 1. Each node in this graph represents a microslice and the line numbers that it contains (we have omitted the microslice partition labels in this diagram). Each edge represents a potential cross-method interaction between microslices and is labeled with the dependence criterion (e.g. shared fields like `buf` and `back`, or called methods e.g. `logit()`). The destination node of a directed edge represents a node that may depend on the source node, i.e. it has a MAY USE feature interaction. In some cases we can detect a MUST USE feature interaction, e.g. if the destination node has a statement that invokes a method call with a single call target which is defined by the source node. Here the notion of use, whether MAY USE or MUST USE, refers to feature usage (not necessarily data flow). For example, note that in the case of a method call, we treat the caller as a user of the method definition (e.g. the microslice containing the call to `logit()` in line 20 uses the code feature defined by the microslices in line 8 and line 9).

The MSIG enables many interesting analyses when combined with concern partition information for each node:

- Edges in this graph represent candidate (structural) feature interactions if they connect nodes which correspond to different concern groups in a partition, while the microslices at the corresponding vertices constitute the resolution of the potential feature interaction. For example, the nodes containing line numbers 13 and 20 each represent the resolution of the interaction between the features BUFFER and LOG.
- Cycles in this graph are merged into a single node which constitutes a resolution of a potential feature interaction if the constituent nodes belong to different concerns. For example, node (17,18) and node (23,25) are part of a cycle, which means that they must be present together and are needed only when both the BUFFER and RESTORE features are required.
- An inclusion relationship between the partition label strings for two nodes (e.g. F1F2B1 and F1B1) may represent intra-method feature dependencies if the corresponding microslices belong to different concerns.
- The leaves of this graph represent microslices that do not have any dependent nodes and hence likely to be easily separable if their constituent statements are part of an optional concern.

Although the MSIG is a powerful abstraction to surface statements involved in inter-method interactions which

could give rise to bloat, the microslicer alone does not possess the "concern-awareness" needed to capture conceptually related feature-specific statements. It may generate numerous overly narrow candidate features and microslices because it does not have any information about the true concern partitions of interest[8]. For this reason, microslicing cannot be used in isolation to detect potential execution bloat statements. We need to perform a concern augmented analysis by combining our MSIG representation with externally supplied concern information to distinguish mandatory and optional features and the methods introduced by them.

4. Concern Augmented Static Analysis of Execution Bloat

Let us assume that we are given a concern-based abstraction that partitions the concerns of a component into two groups - mandatory (always required) and optional.

We perform a CAPA analysis using this information and the statically computed MSIG to identify candidate excess statements that cause execution bloat when none of the specified optional features are required. Estimating the runtime resource overhead induced by these potential bloat contributing statements (dynamically or statically) provides an approximate quantitative measure of execution bloat.

4.1 Bloat detection rules

While the microslicing analysis described earlier finds candidate feature-specific statements and possible structural interactions between them, it cannot conclusively determine which of these are actually resolutions of a feature interaction or potential bloat contributors. One problem is that the dependence graph is built in a may use fashion, so not all edges represent real structural interactions. Also, as we note from the graph, an interaction with an optional feature can involve dependencies in either direction, i.e. both uses and used by relationships are possible. For example, in Fig 3, the statements at lines (17,18) constitute a statement which is used by the optional RESTORE feature, while the call to `logit()` in line 13 is a statement that uses the optional LOG feature. With multiple may use edges in both directions all through the hierarchy (and the possibility of missing edges due to other kinds of dependencies), unresolved decisions accumulate rapidly. Hence, we need a few heuristics or additional information to constrain the space and obtain an approximate analysis of potential bloat.

Table 1 lists a few heuristics and the statements from our working example that are detected as potential bloat contributors using each heuristic. To keep the analysis simple, if we only use heuristics 1 and 2 from the table, the rules can be expressed as the datalog formula shown in Figure 7:

[8] as an aside, we note that it could also miss detecting feature-specific statements introduced for intermediate transformations which do not involve additional input or output fields

	Used By	Uses
1	None	Must use at least one optional node [e.g. (13), (20) in Fig 3]
2	Solely used by optional nodes [e.g. (3), (17, 18) in Fig 3]	Any
3	Solely used by nodes along MSIG paths that eventually contain an optional node	Any
4	Any	Must use at least one optional node

Table 1. Bloat detection heuristics in decreasing likelihood of potential bloat contribution (computed only for MSIG nodes that are inside methods which can be executed by the client program even when no optional feature is used explicitly).

The first rule simply states that a statement is marked as optional if it inside a method that is introduced by a concern which is known to be optional. Concerns that are not optional are conservatively treated as mandatory. Thus, according to the second rule, the statements in methods introduced by those concerns are marked as mandatory. The next rule encodes the heuristic that a statement can be reported as a potential contributor to bloat if it is mandatory (contained in a method that is needed) but MUST USE an optional concern statement (e.g. calls an optional method) and no statements from other methods depend on it. The last rule marks a statement as potential bloat if it is mandatory but is only required by optional concerns.

4.2 Implementation and Scalability

As stated in Section 3.1, we exploit an underlying interprocedural analysis previously used in [6, 35] that maintains flow and context sensitive data and control dependencies including escape analysis and pointer analysis (built over the WALA[9] framework). Although this preliminary analysis is costly, the underlying implementation has already been carefully engineered to be reasonably scalable [35]. The escape analysis results are used to identify input and output fields potentially shared across methods (e.g. the *escape-in* and *escape-out* sets for each method).

The microslices, themselves, however, are intraprocedural and the analysis scales easily. Further, we contain the number of slicing criteria for each method by filtering out some variables, e.g. pass-through escaping fields. The underlying analysis maintains information about the field names or access paths. We currently use this for determining which fields are shared between which methods; the approach introduces imprecision in some situations where the matching is not accurate.

Computing the microslice interaction graph can be expensive when there are a large number of shared fields (or

[9] http://wala.sourceforge.net

$$optional(s) : -inMethod(s, m), inConcern(m, c), isOptional(c)$$
$$mandatory(s) : -inMethod(s, m), inConcern(m, c), isMandatory(c)$$
$$potentialBloat(s) : -mandatory(s), usedBy(s, null), mustUse(s, s'), optional(s')$$
$$potentialBloat(s) : -mandatory(s), usedBy(s, s'), optional(s')$$

Figure 7. Rules expressed as a Datalog formula: Each rule in the formula states that the left hand side relation holds for the tuple inside the braces if all the relations on the the right hand side hold

methods). However, we were able to contain this with simple engineering tactics, such as building only the portion of the graph relevant for identifying interactions corresponding to the bloat detection heuristics.

With these basic optimizations, the entire analysis including the preliminary analysis step took less than 8 minutes for BerkeleyDB (at 95K bytecode instructions and 2382K bytecodes analyzed in total including library code).

4.3 Sources of Imprecision and Tradeoffs

We have used a conservative choice of heuristics to reduce the possibility of incorrectly marking a statement as potential bloat (mis-detected bloat) while tolerating some undetected bloat[10]. However, we found that this is sometimes too conservative and required further engineering tradeoffs to improve effectiveness.

In addition to imprecision in our bloat detection logic (where we rely on heuristics), inherent imprecision in the underlying static analysis can lead to both misdetected or undetected bloat. For example, when mapping virtual call targets, we might miss a potential callee or overestimate callees - the first causes missing edges, the second causes extra edges in the MSIG, both of which affect accuracy. We did not attempt to study these in depth as there are alternative approaches to handle issues of precision in the underlying analysis (e.g. with supporting dynamic analysis) that are independent of the logic for bloat detection/concern augmentation.

4.4 Using the results of CAPA

We take a view similar to most existing techniques for bloat analysis which are intended as an aid for humans, in that we will focus on highlighting a list of candidate bloat statements and do not attempt to de-bloat the program. Even when the analysis output is precise, bloat repair is best left to the programmer as it could be more involved than merely disabling bloat contributing statements or creating specialized versions of functions for the conditions where the concern is optional. Automated de-bloating may be possible in some special cases but we leave this as an area of exploration for future work.

To aid manual de-bloating we can also use the MSIG to provide the programmer additional detail about the statements (microslices) that may be affected if the potential bloat contributing statement is altered. Sometimes a given implementation of a concern that exhibits a high tendency for bloat as determined by our CAPA analysis may even need to be re-designed to be less prone to bloat.

[10] We avoid using the terms *false positive* and *false negative* as it does not have the same connotation here as in the context of program verification where a sound analysis is associated with program safety properties. For example, unlike a bug, the presence of bloat does not result in an incorrect program.

5. Evaluation

Evaluation objectives and considerations: Bloat analysis tools [48, 50] that highlight inefficiency patterns are typically evaluated for usefulness in terms of the performance improvement that could be achieved by fixing program code to address the candidate bloat patterns highlighted by the tools.

In addition, our approach amounts to a directed "what-if" bloat analysis for a given concern (or set of concerns) and hence must be evaluated in terms of how well it answers the question ''what-if this concern were optional, then which statements would be bloat". That is, we evaluate the ability to narrow down the few candidate statements that are potential sources of bloat due to the concern, rather than the volume of bloat uncovered. Note that our analysis relies on externally supplied concern information which is not available by default for standard Java benchmarks such as the DaCapo [10] suite.

Three pronged approach We adopt a three pronged approach to evaluate our technique:

In the first step, we perform a correctness check of our tool's output. As there are no existing benchmarks available for cross-validating results of execution bloat analysis, we compile a set of well-understood control examples and case studies drawn mostly from existing literature on feature oriented programming and library specialization. This enables us to create an oracle against which the results of our tool can be verified.

In the second step, we evaluate the scalability of our analysis by running it on larger examples with some manually verifiable form of concern bloat. For one example, we compare our results against concern statements independently labeled by another programmer. For some of the other large

benchmark programs we specify "what-if" questions based on concerns relevant for uncovering specific cases of bloat that have been reported by previous researchers or developers.

In the third and final step, we remove the bloat statements and evaluate the gain in performance.

Benchmark programs Our set of chosen case studies includes ten Java programs of different sizes.

- Five small to medium sized components with precisely-known fine-grained feature assignments and optional concerns that cause execution bloat, where we ensure a line by line validation of results against a pre-determined oracle. Although these examples are not large, they are carefully chosen to test the ability to detect bloat due to various fine-grained structural interactions. These are
  1. **Buffer** is the 3 feature BUFFER-LOG-RESTORE example (Fig 1) from prior work on feature oriented programming [29] described earlier.
  2. **Adaptor** is an adaptor pattern that includes an optional big endian to little endian converter, illustrated in Fig 5, similar to a feature previously described by Prehofer [38]. We created this example because many anecdotes of bloat highlight excess or unnecessary transformations and checks, and we wanted to determine if our technique could detect such patterns.
  3. **Stack** is an enhanced implementation of the classic FOP example proposed by Prehofer [38], a 5 feature STACK with COUNTER, LOCK, UNDO and BOUND checking.
  4. **c5list** is a (simplified) Java implementation of a LinkedList with support for Views from the C5 collection library for C#. This example has been the main subject of a study on different library specialization techniques [1], so it was an interesting experiment to explore whether our technique is capable of automatically detecting bloat due to the optional feature "Views" in this example.
  5. **Graph** is the well-known Graph Product Line implementation available at the FEATUREHOUSE website [3].
- Five large benchmark programs with some previously known optional concern assignment (via manual labeling or available from previous work) on bloat.
  1. **xml benchmark from SPECjvm2008** is a Java benchmark from SPEC [46]. We specified `validation` as an optional concern (it is relevant from a benchmark perspective but could be optional otherwise). We picked this concern as it is likely to occur at fine grained levels in different places in the code.
  2. **SPECjbb2005** is a Server-side Java benchmark from SPEC [46]. From an inspection of the code during prior research on object churn bloat, we have observed bloat due to transformations to XML format for temporary in-memory storage. XML formatting is not strictly an essential feature here.
  3. **DaCapo BLOAT** is a benchmark from the DaCapo 2006 benchmark suite [10]. Other researchers have noted that the `bloat` benchmark incurs considerable object churn bloat in building strings just for the purpose of assertions [48, 50] even when the assertion conditions do not occur.
  4. **BerkeleyDB** is a widely used lightweight embedded database toolkit. For our experiments we used a full featured version of the Java Edition of BerkeleyDB included in a set of case study examples for virtual separation of concerns (CIDE [19]). We applied our analysis to identify bloat contributing statements due to the costly optional concern (trace) logging.
  5. **Xylem** Xylem is the original tool that this analysis has been built upon. Originally designed to be a tool to detect null dereferences [35] it has over the years grown arms and legs to handle various other analyses [6, 27, 34, 36]. So we ran our analysis to detect bloat generated by these additional concerns.

The general characteristics of these benchmarks are given in Table 2.

We do not rely on a specific concern location tool in our evaluation. Our analysis takes concern information as supplied, without regard to how the information was obtained.

- The input to our analysis is a jar file of the component or program and a list of method name search patterns corresponding to one or more optional concerns.
- The output is a list of statements that contribute to potential execution bloat due to the specified optional concerns.

5.1 Case study results:

Table 3 summarizes the results for the ten case studies. Table 4 summarizes the performance improvement results for the case studies when run in bloated form and in unbloated form.. We observe that our analysis is effective in correctly detecting bloat with a reasonable precision in all these examples although it can miss some candidate bloat statements. We define precision and recall as

$$Precision = \frac{\text{bloat sources detected correctly}}{\text{bloat sources detected correctly + bloat sources misdetected}}$$

$$Recall = \frac{\text{bloat sources detected correctly}}{\text{bloat sources detected correctly + bloat sources undetected}}$$

	Buffer	Adaptor	Stack	c5List	Graph	SPECjvm 2008(xml)	SPECjbb 2005	DaCapo BLOAT	Berk-leyDB	Xylem
Functions analysed	12	10	40	15	137	542	864	4293	3798	1631
Bytecode statements analyzed	63	53	339	388	1188	13035	33924	124588	91605	145909
Analysis time	4.9s	5.3s	6.5s	5.3s	6s	1m 31s	28.2s	1m 59s	7m 55s	48.5s

Table 2. Benchmark characteristics

	Buffer	Adaptor	Stack	c5List	Graph	SPECjvm 2008(xml)	SPECjbb 2005	DaCapo BLOAT	Berk-leyDB	Xylem
No. of microslices computed	16	8	97	58	281	1928	4769	21867	16695	9472
Optional concerns evaluated for bloat propensity ("What-if" question)	RESTORE LOG	ENDIAN	COUNTER, LOCK, UNDO, BOUND	VIEWS	WEIGHTED	VALIDATE	XML	ASSERT	TRACELOG	JARIZE
Sources of bloat due to evaluated concerns	5	4	20	9	11	33	20	316	-	62
Bloat sources detected correctly	**5**	**4**	**18**	**9**	**6**	**21**	**20**	**315**	**159**	**62**
Bloat sources undetected	**0**	**0**	**2**	**0**	**5**	**12**	**0**	**1**	-	**0**
Bloat sources misdetected	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**37**	**28**
Precision	100%	100%	100%	100%	100%	95%	100%	100%	81%	69%
Recall	100%	100%	90%	100%	54%	64%	100%	99%	-	100%

Table 3. Results of concern augumented program analysis of the execution bloat propensity of a specified set of optional concerns in different case studies. The analysis narrows down sources of bloat to the few source code lines instrumental for avoiding execution bloat in situations where the given concerns are optional.

	Buffer	Adaptor	Stack	c5List	Graph	SPECjvm 2008(xml) (startup)	SPECjbb 2005	DaCapo BLOAT	Berk-leyDB	Xylem
Bloated runtime or throughput	27.4s	12.3s	49.8s	11.5s	2.6s	28.0s	10813 bops	5.3s	131.3s	82.5 (ant1.5) 148.1s (ant1.65)
De-bloated runtime or throughput	0.1s	6.3s	32.3s	7.6s	1.5s	21.3s	16769 bops	2.3s	14.6s	54.5s (ant1.5) 75.8s (ant1.65)
% Improvement	99%	48%	35%	34%	42%	24%	55%	56%	89%	34–49%

Table 4. Benchmark performance results. Measurements were taken on an Intel(R) Core(TM) i7 L640, 2.13GHz system with 3.7GB memory, running Linux Fedora 13 and Java 1.6, OpenJDK 64-bit Server VM (build 14.0-b16, mixed mode).

We elaborate on the case studies[11] in a little more detail below:

I. Programs with precisely known fine-grained feature assignments: The concern (feature) assignments in these programs are available at statement granularity, which we use to create an oracle[12] to compare whether our analysis correctly detects bloat. Note that, our analysis is not aware of the fine-grained feature assignments. We only provide it method level concern information for the (optional) concerns we evaluate for bloat propensity.

Buffer: We specify 2 optional concerns, LOG (the `logit()` method) and RESTORE (the `restore()`) method. The analysis correctly located the statements that are the sources of execution bloat (lines 3, 13, 17, 18 and 20). It could recognize the assignment to `back` in the `set()` method that occurs through a temporary variable (via intra-procedural backward slicing) and also recognized the initialization of the field `back`.

[11] The first four case studies below can be made available for other researchers on request. The rest of the case studies (other than Xylem) use externally available benchmarks

[12] The oracle marks a statement belonging to an optional concern as potential bloat if it lies inside a method belonging to a concern that is not known to be optional

On de-bloating the code based on the analysis results, we observed a large (99%) improvement in performance in a test application using this component. This is primarily because the LOG concern involves printing which incurs a huge overhead.

Adaptor: The method `big2LittleEndian` was specified as the optional concern. Our analysis was able to detect as bloat, the excess parsing (lines 12, 14) and condition check (line 11) that was needed only to enable this converter, besides the call to `big2LittleEndian` in line 13.

On de-bloating the code based on the analysis results, we observed a 48% improvement in performance and 99% reduction in object creation in a microbenchmark application using this component.

Stack: We specify 4 optional features: COUNTER, LOCK, UNDO and BOUND checking. This example spans multiple classes and captures several possible nuances of the way features interact to cause bloat. The details are available in Appendix A.1. As seen from Table 3, column 3, our technique is able to detect most of them.

On de-bloating the code based on the analysis results, we observed a 35% improvement in performance and 30% reduction in object creation in a test application using this component.

The statements that our analysis missed were calls to a `save()` method which belongs to the UNDO concern but was not listed in the concern information supplied. Even so, our technique was able to detect the main state saving statement `savedState = new String(state);` at line 77 (Appendix A.1) in the `save()` method as excess because it is in a microslice that is only used by the UNDO concern. This illustrates that our analysis may be effective even with slightly incomplete concern information.

c5list We specify the optional feature "Views" in this example. As seen in Table 3, column 4, we found that the analysis could indeed detect all known instances of such statements. For example, it could identify statements of the form: `if (underlying != null) underlying.counter++;` [1] without the information that `underlying` is only relevant for "Views".

On de-bloating the code based on the analysis results, we observed a 34% improvement in performance in a test application using this component.

Graph is the well-known Graph Product Line implementation available at the FEATUREHOUSE web-site [3]. We started with a variant that supports weighted directed graphs and used our analysis to find execution bloat due to the optional weighted feature. There were only 11 such sources, as per our oracle. Our method was able to correctly detect 6 of them (Table 3, column 5). The remaining 5 sources were all related to a vector field `weightsList` which is initialized directly in the mandatory `Vertex` constructor rather than any method that is specific to the weighted feature. This could be detected if we were to expand concern input provided to include fields matching a concern's keywords.

Even though our technique missed these statements, on de-bloating the code based on the analysis results, we observed a 42% improvement in performance and 88% reduction in object creation in a test application using this component.

II. Large programs with a manually verifiable optional concern: In these cases we had to rely on prior experience / knowledge to specify an optional concern that contributes to bloat which we also know how to find manually so that it is practical to validate results line by line.

xml benchmark from SPECjvm2008 We specified `validation` as an optional concern. In this case, there is no pre-determined oracle. We had to rely on another programmer to manually inspect the code and independently annotate all the methods and lines in the code that belong to the `validation` concern.

As shown in Table 3, column 6, our analysis found 22 sources of bloat due to this concern. We confirmed that 12 of these coincided with the manual annotation. Of the remaining 9, one was found to be an instance of mis-detected bloat, while 8 of the sources were found to be in utility methods (such as `writeDiff(), isUnixNewLine()`) which were used only by the validation methods (our tool reports the specific microslice statements where the potential use occurs). Hence we treat these 8 sources as correctly identified. In addition there were 12 lines of bloat as per the manual annotation which went undetected by our tool.

The instance of mis-detected bloat occurred because of imprecision introduced by the underlying analysis in mapping virtual call targets (see Section 4.3). We could determine this based on the detailed output of the tool. Among the cases of undetected bloat, several seem to be related to fields that are both directly updated and used in non validation specific methods such as `setupBenchmark()` and hence difficult for our analysis to distinguish. One way to cover these cases is to expand concern input provided to include fields matching a concern's keywords in addition to supplying the concern's methods.

De-bloating the code based on the analysis output significantly reduces expensive validation that occurs in the benchmark setup phase, resulting in an 24% improvement in the combined startup and execution time for a single iteration benchmark run.

SPECjbb2005: As XML formatting is not strictly an essential feature here, we specified it as an optional concern. We were then able to use our analysis to find the statements that contribute bloat due to this concern (Table 3, column 7). These included string copies that we know (from our prior work [6]) to be *r*esponsible for 40% temporary object churn. On de-bloating the code based on the analysis results, we observed a 55% improvement in benchmark performance (for 4 warehouses).

DaCapo BLOAT: We specified assertions as an optional concern and were able to find potential bloat statements (Table 3, column 8). Even though assertions themselves are fairly straightforward to find using a code search, our approach has the advantage that it can also automatically catch statements that build up data structures just for the purpose of debugging.

According to prior work [50], addressing these overheads can significantly reduce data copies in this benchmark. On debloating the code based on our analysis results, we observed a 56% improvement in benchmark performance and 87% reduction in temporary object creation.

BerkeleyDB: As concern input to our tool, we specified method names which contain the string "trace" or "Trace", as optional. To measure the precision of our analysis, we compared the results with the corresponding annotation information in CIDE [19]. Our technique detected 159 instances of bloat correctly, with 81% precision overall.

We did not measure recall because not all annotated lines that belong to the logging concern contribute to bloat. Instead, we used CIDE to generate a specialized program variant without the concern and measured the total bytecodes executed[13] by the variant under a simple insertion retrieval benchmark demo application (`Simple_JE`). We compared this with the total bytecodes executed by manually debloated version of code per our analysis and found it to be close agreement with that measured for the variant generated by CIDE and significantly (63%) lower than total bytecodes executed by the original full featured code when running the same benchmark. We observed a significant (89%) improvement in performance and 85% reduction in object creation using the code debloated per our analysis. Incidentally, BerkeleyDB already has an option to turn off logging which exhibits comparable performance gains.

Xylem: One optional concern in Xylem is that after doing the basic data and control dependence analysis, it stores the analysis results in a jar file to permit faster startup at a lower memory cost for subsequent analysis such as null dereference analysis [35]. However, for any analysis that is built directly on top of the basic analysis, generating the default jar file was becoming a fairly large overhead. To identify bloat due to this concern we specified as input, method names which contain the string "jarize". While our tool detected every statement that was a source of this bloat, it also picked up all the statements related to another optional concern, "modelDriver" [36]. Thus we find that we may be unable to separate out bloat related to two cross-cutting concerns, unless we somehow mark the fields for one of the concerns as mandatory.

On de-bloating the code based on the analysis results, we observed a 30–49% improvement in performance depending on the size of the input and the size of the (optional) jar that was generated.

[13] we used the JP2 bytecode profiler [8] for these measurements

5.2 Discussion

5.2.1 Approaches for Obtaining Concern Input Labels

In our proof of concept experiments, we have used a very rudimentary keyword pattern match on fully qualified method names as our concern input generation tool. There are many other possible sources of concern input that could be used instead. Multiple sources of information, techniques and their combinations have previously been used for aiding concern analysis including source code mining [5, 52], information retrieval, formal concept analysis, slicing [9, 18], execution traces [12], human input, natural language processing, and concept databases. Concern inputs can range from machine discovered concerns (e.g. by using topic models on code [5, 13]) and sophisticated code searches to expert supplied programmer annotations for concern separation, such as feature oriented programming [2] or aspect oriented programming (AOP) [23].

Our what-if analysis is applied to concerns that the user of the tool considers to be potentially optional. In the case of machine discovered concerns from an unfamiliar code base, the user marks which of the discovered concerns to analyze for potential bloat.

5.2.2 Computing the MSIG First vs Adopting a Demand-Driven Approach

We have treated the static analysis for the MSIG computation and the concern partitioning step as parallel stages. These stages can potentially be composed in any order or even blended iteratively if appropriate.

Our current implementation overlays concern information on the micro-partitioned program. Precomputing the MSIG in this manner before the concern augmentation step has the advantage that the MSIG can be reused to answer multiple "what-if" questions (e.g. to identify sources of bloat for different groups of optional concerns). It also makes it easier to handle more complex conditional optionality rules[14] across feature combinations. Further, in situations where obtaining concern labels requires some manual effort, the MSIG can be used to focus the effort only on methods involved in costly structural interactions.

Alternatively, the order may be reversed to support a more directed or demand-driven approach that only computes the part of the MSIG relevant for answering a single "what-if" question corresponding to a set of optional concerns. Such approach can also be useful for fine grained concern location.

5.2.3 Restrictions of our Program Dependence Based Interpretation of FOP

There are subtle differences between the FOP (Feature Oriented Programming) notion of the resolution of a feature in-

[14] such as a high level feature model grammar that specifies which groups of concerns are mutually optional or which other concerns must be enabled when a given concern is required

teraction (e.g. a feature derivative [29]) and the pure program dependence based interpretation we use, but the principle is very similar. For instance, we use a somewhat restrictive version of what code a feature extension can add or what a structural interaction can look like. So unlike FOP our model is limited to whatever could be statically detected based on program dependence relationships. For example in FOP a feature can add any code to a method, not necessarily one that uses/adds a new variable or calls a method. Also, in FOP even a method addition to a class introduced by another feature is termed a feature interaction, while we are focused only on interactions via code introduced within a method. On the other hand, we also overestimate candidate feature interactions and their resolution because of the fine granularity of micro-slicing.

6. Related Work

6.1 Java Bloat Analysis

Several approaches have been explored for detecting runtime bloat in Java applications [51]. Data structure health signatures [31] use the structural semantics of Java's data model to distinguish data structure representation overhead to measure memory bloat. Different notions of bloat have been considered in the absence of an explicit model for distinguishing overhead from necessary data or activity. A variety of symptoms of excesses have been used to recognize the presence of bloat, such as high volumes of temporary objects, unbalanced costs vs benefit of object creation and consumption [6, 15, 48–50] and object reuse opportunities in loops [6]. However, bottom up techniques are limited by lack of higher level insight about the purpose of code statements responsible for the pattern of excess activity suspected. Augmenting the analysis with information about program concerns and their properties (such as when a concern is necessary and when it isn't) enables us to tackle the hitherto unsolved problem of analyzing execution bloat propensity of optional concerns. Existing runtime bloat detectors can also benefit from such higher level insight to aid de-bloating.

6.2 Concern Analysis

There is a large body of existing literature on concern location and discovery, most of which is oriented towards program comprehension, maintenance and re-engineering tasks. The space of related topics is too vast to cover, and traverses sub-fields that would each merit their own comprehensive survey, such as feature location [14], aspect mining [22], concept assignment [7] and change impact analyis [4, 17].

Feature location [14] and impact analysis [17] are geared at locating the extent of a specific concern such as a feature, bug or change request, e.g. source code fragments that either implement or affect the concern. Aspect mining [22], on the other hand, looks for general indicators of cross-cutting concerns to identify candidate aspects, e.g. via a fan-in analysis of methods [30] or measures of topic scattering in source code [5]. A variety of techniques have been employed across these problem domains, ranging from formal concept analysis [47], exploiting program topology [40, 41], natural language processing [43, 44], information retrieval, graph mining [30, 52], program slicing [9, 18] and dynamic analysis [12]. Solutions that combine multiple approaches [16, 39, 42, 53] and exploit multiple sources of information including test cases, program documentation and evolution history have also been used to improve the quality of results.

The existence of these techniques makes the concern augmentation step in our analysis feasible or usable in practice, by supplying method level concern assignment information that we need as input. Traditionally program analysis has been used to aid concern location [16, 18]. In contrast we propose the use of externally supplied concern information to enrich program analysis for solving new problems.

According to surveys of techniques for feature location [14], aspect mining [22] and impact analysis [28], most approaches are geared to provide method level granularity of concern information, while support for statement level granularity is relatively rarer (even most efforts at creating feature location benchmarks have been geared at method level granularity). By requiring no more than method level granularity of concern information as input, we ensure that our approach is practical and widely applicable, with a fine grained analysis that can be tuned specifically for bloat detection. At the same time, we can easily reuse accurate fine-grained statement level concern assignment information, if provided by a concern location or aspect mining tool, in which case we would skip our fine-grained microslicing analysis for finding feature interaction resolutions.

6.3 Feature Oriented Software Development

Our approach in this paper is inspired by research in feature oriented programming (FOP). In particular, we exploit the elegant concept of structural feature interactions and their resolution (also referred to as feature derivatives) [2, 24, 29, 38]. Our novel contribution is to link this concept to the problem of detecting statements that are potential sources of execution bloat. We also devise techniques that enable the practical application of these ideas to bloat assessment of non FOP software such as framework based components, given only coarse method level concern information. Recovering an exact feature oriented decomposition of a legacy program (from which specialized product line variants can be generated) remains hard to automate [20, 29]. Fortunately, the task of detecting candidate sources of bloat has less exacting requirements on precision in order to be effective and useful as we only aim to draw programmer attention to potential bloat contributors rather than refactor a correct bloat-free sub-program.

6.4 Feature-aware analysis and irrelevant features in product lines

Feature-aware program analysis [11, 21, 26] has typically been applied in product lines and feature-oriented or aspect-oriented software. Sophisticated tools have been developed to reason about the static and dynamic properties of program variants corresponding to different feature combinations without having to check each and every possible combination or product in the product line. For example type-checking [21] and static analysis [11] can be applied to the entire product line as whole. Kim et al. explore test reduction in product lines [25, 26] by using static analysis to find features irrelevant for a given test or safety property being monitored, based on whether the feature affects the control or data flow of any feature whose code may be executed by the test.

Our work complements these efforts in the product line and feature oriented software community by exploiting concern location information for analysis such as bloat detection in a broader set of software applications which need not have built-in detailed and fine grained feature assignments.

Siegmund et al. recently introduced a black box approach for detecting runtime feature interactions in order to estimate performance and other non-functional properties of product line variants [45]. This could potentially be used estimate the performance impact of bloat due to an optional concern without generating all the variants with and without the feature. However, the approach would still require the ability to automatically generate at least one correct runnable variant of the program without the concern, so it does not apply to non product line software where exact concern assignments are not typically available.

6.5 Slicing

The use of slicing in refining concern location, key statement analysis [18] and building concern dependence graphs for program understanding is fairly well-established. Several heuristics such as the use of barriers during dependence graph traversal have been proposed for containing the size or (interprocedural) span of slices. Our microslicing analysis for automatic feature-specific statement decomposition addresses a novel objective - to unearth candidate fine grained structural feature interactions that could result in execution bloat and to isolate the corresponding feature-specific statements that potentially contribute to bloat. We incorporate concepts from feature oriented programming in (micro)slicing. Microslices are intra-procedurally modeled as the resolution of candidate feature interactions [24, 29]. We then build relations between these microslices to enable bloat inference when augmented with requisite concern information.

7. Conclusions

In this paper, we introduced the use of concern information in program analysis tasks and demonstrated its application in estimating the propensity for execution bloat of optional concerns in Java programs. We use conservative abstractions when concern properties are not available for all statements and microslicing when concern assignments are too coarse. The effectiveness of the CAPA methodology is highly dependent on the quality of concern information available. Despite this caveat, our results show that it can provide a fresh approach to problems such as bloat analysis that involve jointly reasoning about intent, structure and dynamics of program behavior.

While we have concentrated much on the runtime overhead caused by code that is not needed, there is the additional potential positive effect that code that is not required does not need to be read and understood by developers – hence, another benefit of the approach could be that a reduced API could be generated which potentially reduces the developers' effort to use it.

An interesting area of future work is to develop techniques to aid the automatic discovery of information about optional concerns and to explore tools to aid in automatic de-bloating of software.

Acknowledgments

We thank Giriprasad Sridhara for contributions to the evaluation, Rupesh Nasre, Sandya Mannaraswamy and K Vasanta Laxmi for their feedback on the paper.

A. Case study example

A.1 Stack with 5 features

- STACK
- COUNTER
- LOCK
- UNDO
- BOUND

```
1   package multifeature;
2
3
4   public class StackFull {
5       private String state = new String();
6       private String savedState = "";
7       private char minValue;
8       private char maxValue;
9       private NestedLock txn = new NestedLock();
10      private Counter cnt = new Counter();
11
12      public StackFull() {
13      }
14
15      private void clipBound() {
16          for (int i=0; i¡state.length(); i++) {
17              if (state.charAt(i) < minValue) {
18                  state = state.replace(state.charAt(i), minValue);
19              }
20              if (maxValue > 0 && state.charAt(i) > maxValue) {
21                  state = state.replace(state.charAt(i), maxValue);
```

```
                }
            }
        }

        public void bound(char low, char high) {
            txn.lock();
            minValue = low;
            maxValue = high;
            clipBound();
            txn.unlock();
        } }

        public void empty() {
            txn.lock();
            save();
            state = "";
            cnt.reset();
            txn.unlock();
        }

        public void push(char item) {
            if (item < minValue) {
                return;
            }
            if (maxValue > 0 && item > maxValue) {
                return;
            }
            txn.lock();
            save();
            state = String.valueOf(item).concat(state);
            cnt.increment();
            txn.unlock();
        }

        public void pop() {
            txn.lock();
            save();
            state = state.substring(1);
            cnt.decrement();
            txn.unlock();
        }

        public char top() {
            if (txn.isUnlocked()) {
                return state.charAt(0);
            }
            return 0;
        }

        void push2(char item) {
            push(item);
            push(item);
        }

        private void save() {
            savedState = new String(state);
        }

        public void undo() {
            txn.lock();
            state = savedState;
            cnt.undo();
            txn.unlock();
        }

        public void print() {
            System.out.println(state);
            cnt.print();
        }

    }

    class Counter {
        private int index = 0;
        private NestedLock key = new NestedLock();
        private int oldIndex = 0;

        public Counter() {
        }

        public void reset() {
            if (key.isUnlocked()) {
                save();
                index = 0;
            }
        }

        public void increment() {
            if (key.isUnlocked()) {
                save();
                index++;
            }
        }

        public void decrement() {
            if (key.isUnlocked()) {
                save();
                index--;
            }
        }

        public void disable() {
            key.lock();
        }

        public void enable() {
            key.unlock();
        }

        public int size() {
            return index;
        }

        private void save() {
            oldIndex = index;
        }

        public void undo() {
            if (key.isUnlocked()) {
                index = oldIndex;
            }
        }

        public void print() {
            System.out.println("count = " + index);
        }
    } }

    class NestedLock {
        private boolean avail = true;
        private int nest = 0;

        public NestedLock() {
        }

        public void lock() {
            if (!avail) {
                nest++;
            }
            avail = false;
        }

        public void unlock() {
            if (nest < 1) {
                nest--;
                return;
            }
            nest = 0;
            avail = true;
        }

        public boolean isUnlocked() {
            return avail;
        }
    }
```

References

[1] B. Aktemur and S. Kamin. A comparative study of techniques to write customizable libraries. In *SAC*, 2009.

[2] S. Apel and C. Kastner. An overview of feature-oriented software development. Journal Of Object Technology, 2009.

[3] S. Apel, C. Kastner, and C. Lengauer. FEATUREHOUSE: Language-independent, automated software composition. ICSE, 2009. URL `http://fosd.de/fh`.

[4] R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. ISBN 0818673842.

[5] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA*, 2008.

[6] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta. Reuse recycle to debloat software. In *ECOOP*, 2011.

[7] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE '93*, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[8] W. Binder, J. Hulaas, P. Moret, and A. Villaz. Platform-independent proling in a virtual execution environment. Software Practice and Experience, 2009.

[9] D. Binkley, G. Gold, M. Harman, Z. Li, and K. Mahdavi. An empirical study of the relationship between the concepts expressed in source code and dependence. volume 81. The Journal of Systems and Software, 2008.

[10] S. Blackburn and et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[11] E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. Spllift - statically analyzing software product lines in minutes instead of years. In *PLDI'13*, 2013.

[12] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. volume 99. IEEE TSE, Apr. 2009.

[13] M. K. Das, S. Bhattacharya, C. Bhattacharyya, and K. Gopinath. Subtle topic models and discovering subtly manifested software concerns automatically. In *ICML*, 2013.

[14] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013. ISSN 2047-7481.

[15] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *SIGSOFT '08/FSE-16*, 2008.

[16] M. Eaddy, A. V. Aho, G. Antoniol, and Y. G. Guhneuc. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. ICPC, 2008.

[17] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated impact analysis for managing software changes. In *ICSE*, 2012.

[18] M. Harman, N. Gold, R. Hierons, and D. Binkley. Code extraction algorithms which unify slicing and concept assignment. In *WCRE*, 2002.

[19] C. Kästner and S. Apel. Type-checking software product lines – a formal approach. In *ASE'07*, Los Alamitos, CA, 2008. URL `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide`.

[20] C. Kästner, A. Dreiling, and K. Ostermann. Variability mining with leadt. Technical report, 2011.

[21] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3), 2012.

[22] A. Kellens, K. Mens, P. Tonella, and D. D. Informatique. A survey of automated code-level aspect mining techniques. In *In Transactions on Aspect Oriented Software Development*, pages 145–164, 2007.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[24] C. H. P. Kim, C. Kästner, and D. S. Batory. On the modularity of feature interactions. In *GPCE*, 2008.

[25] 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*, pages 285–299. Springer, Nov. 2010.

[26] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *AOSD*, 2011.

[27] M. Kim, S. Sinha, C. Gorg, H. Shah, M. J. Harrold, and M. G. Nanda. Automated bug neighborhood analysis for identifying incomplete bug fixes. In *ICST*, 2010.

[28] B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, Apr. 2012.

[29] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. ICSE, 2006.

[30] M. Marin, A. V. Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *TOSEM*, 17:3:1–3:37, December 2007.

[31] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*, 2007.

[32] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behaviour in framework based applications. In *ECOOP*, 2006.

[33] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to java runtime bloat. *IEEE Software*, 27(1), 2010.

[34] A. Nanda and M. G. Nanda. Gaining insight into programs that analyze programs: by visualizing the analyzed program. In *ONWARDS, Companion to OOPSLA*, 2009.

[35] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for Java. In *ICSE*, 2009.

[36] M. G. Nanda, S. Mani, V. S. Sinha, and S. Sinha. Demystifying model transformations: An approach based on automated rule inference. In *OOPSLA*, 2009.

[37] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 3540243720.

[38] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, 1997.

[39] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. ICPC, 2010.

[40] M. P. Robillard. Topology analysis of software dependencies. ACM TOSEM, Aug. 2008.

[41] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1), Feb. 2007.

[42] T. Savage, M. Revelle, and D. Poshyvanyk. Flat3: Feature location and textual tracing tool. ICSE, 2010.

[43] D. Shepherd, L. Pollock, and T. Tourwé. Using language clues to discover crosscutting concerns. In *Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, MACS '05. ACM, 2005.

[44] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *AOSD*, 2007.

[45] N. Siegmund, S. Kolesnikov, C. Kastner, S. Apel, D. Batory, M. Rosenmuller, and G. Saake. Predicting performance via automated feature-interaction detection. In *ICSE*, 2012.

[46] Standard Performance Evaluation Corporation. http://www.specbench.org.

[47] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *SCAM'04*, 2004.

[48] G. Xu and et al. Finding low-utility data structures. In *PLDI*, 2010.

[49] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, 2010.

[50] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI*, 2009.

[51] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FoSER*, 2010.

[52] C. Zhang and H.-A. Jacobsen. Efficiently mining crosscutting concerns through random walks. In *AOSD*, 2007.

[53] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. Sniafl: Towards a static non-interactive approach to feature location. volume 15. ACM TOSEM, April 2006.