A Choice of Variational Stacks: Exploring Variational Data Structures

Meng Meng,[1] Jens Meinicke,[2,3] Chu-Pan Wong,[2] Eric Walkingshaw,[1] Christian Kästner[2]

[1]Oregon State University, Corvallis, OR, USA
[2]Carnegie Mellon University, Pittsburgh, PA, USA
[3]University of Magdeburg, Magdeburg, Germany

ABSTRACT

Many applications require not only representing variability in software and data, but also computing with it. To do so efficiently requires variational data structures that make the variability explicit in the underlying data and the operations used to manipulate it. Variational data structures have been developed ad hoc for many applications, but there is little general understanding of how to design them or what tradeoffs exist among them. In this paper, we strive for a more systematic exploration and analysis of a variational data structure. We want to know how different design decisions affect the performance and scalability of a variational data structure, and what properties of the underlying data and operation sequences need to be considered. Specifically, we study several alternative designs of a variational stack, a data structure that supports efficiently representing and computing with multiple variants of a plain stack, and that is a common building block in many algorithms. The different variational stacks are presented as a small product line organized by three design decisions. We analyze how these design decisions affect the performance of a variational stack with different usage profiles. Finally, we evaluate how these design decisions affect the performance of the variational stack in a real-world scenario: in the interpreter VarexJ when executing real software containing variability.

Keywords

Variational data structures, variability-aware execution

1. INTRODUCTION

Variation is common in both software systems and data computation. In software systems, variability is introduced so that users can configure the software according to different use cases, for example, using command line options, plugins, or preprocessor directives. In data computation, variability arises by running a program or part of a program many times with inputs that are varied only slightly, as in configuration testing, uncertainty analysis, and speculative analysis [8, 9, 15, 16, 18]. Although variation makes software and computation flexible, it also requires more efficient techniques for analyzing and executing programs since it is usually not possible to explore each variant individually due to the combinatorial explosion of possibilities. Recent research in several domains addresses the combinatorial explosion problem with a variety of solutions that share some common ideas: analyze, manipulate, and compute with an explicit representation of variation in code and data, and exploit sharing among the variants.

By encoding variation explicitly in code and data, many approaches gain significant performance improvement without sacrificing precisions of the results. For example, uncertainty analysis proposed by Sumner et al. [18] runs faster by operating on a vector of uncertain input values all at once, rather than on individual values sequentially, since many computations are independent and can be shared. Encoding and manipulating variability explicitly has also been extremely successful in analyzing software product lines. Such variational analyses are shown to scale to large configuration spaces but still provide sound results [13, 19].

Data structures for computing with explicit variation are often reinvented and optimized in an ad-hoc way with little reuse across projects. We envision that more general *variational data structures* can be essential building blocks to be reused across all of these domains, but they are not currently well understood [22]. This paper is a step towards a more systematic exploration and analysis of the design and implementation of variational data structures. Specifically, we focus on variational stacks. We analyze how different design decisions and optimizations affect the performance of variational stacks, and we evaluate how different variational stacks perform in practice when used in the implementation of the the variability-aware Java interpreter VarexJ, where variational stacks play a central role [12, 14].

The main contributions of this paper are:

- The description of a small family of variational stack data structures (Section 3). The family consists of two alternative core stack implementations and two independently optional optimizations, leading to eight different variational stacks.
- An exploratory analysis of variational stacks on artificially generated operation sequences (Section 4.1) that vary in the number of configuration options they reference and in the distribution of variational operations.
- An empirical analysis of the performance of each of the eight variational stacks when used as the varia-

VaMoS '17, February 01 - 03, 2017, Eindhoven, Netherlands

ISBN 978-1-4503-4811-9/17/02…$15.00

DOI: http://dx.doi.org/10.1145/3023956.3023966

tional operand stack of the variational Java interpreter VarexJ (Section 4.2). Our experiments show that both optimizations are highly effective in practice, and also demonstrate that choosing the right variational data structure can have a significant impact on the overall performance of programs that compute with variability.

2. BACKGROUND AND RELATED WORK

In this section, we provide necessary background on variational data structures and also on variability-aware execution, an analysis strategy that extensively uses variational data structures to share common execution paths across variants. We use the variability-aware interpreter VarexJ in later sections to evaluate our variational stack implementations.

2.1 Variational Data Structures

Conceptually, *variational data* represents many different concrete data values at once. However, variational data is not just a flat set of variants, but also describes which configurations each variant is associated with. Variation on atomic data values can be expressed in different forms, such as by trees of choices between alternatives or by maps from variants to the configuration context where each is relevant [3, 22].

The choice calculus is one way to express variational data [3, 6, 21]. For example, consider the value $x = \mathit{Choice}(\alpha, 1, 3)$, a choice that represents either the concrete value 1 if the *configuration option* α evaluates to true, or 3 otherwise. Computations on x will operate on both 1 and 3, preserving the fact that 1 is associated with the variational context α and 3 with the context $\neg\alpha$. Clearly, the size of the configuration space for variational data is exponential in the number of independent configuration options it contains.

Variational data structures are designed to compactly represent and compute with variational data [22]. In previous work, we discussed some designs of variational lists, maps, and sets [22]. However, it is still unclear how the design decisions described in that work affect the scalability of variational computations in practice, and which design decisions have yet to be identified. Thus, a systematic evaluation and exploration of the design space is needed.

Variational data structures have diverse applications. They are commonly used for variational program analysis, such as variational ASTs for type checking [7], variational type inference [2], and variational executions [4, 12, 16]. They are also proposed for variational representations of software artifacts, such as test suits, formal specification, and deductive verification [22]. Further applications from other domains already need to cope with variation, such as travel planning, uncertainty in analysis, and context-oriented programming [22].

2.2 Variability-Aware Execution

Variability-aware execution makes extensive use of variational data structures. The core idea is to combine repeated computations into a single execution, tracking differences while sharing as much data and execution as possible. This technique is useful, for example, in testing highly configurable systems [8, 12, 14, 16], where it enables faster testing of all configurations and scales well to large configuration spaces.

A *variability-aware interpreter* is a programming language interpreter that supports variability-aware execution [8, 12, 14, 16]. In such an interpreter, data values are variational and instructions are executed within variational contexts (corresponding to the selection of some or all configuration options). In previous work we have designed VarexJ, a variability-aware interpreter for Java [12, 14]. Since VarexJ computes with variational data, there are many applications for variational data structures.

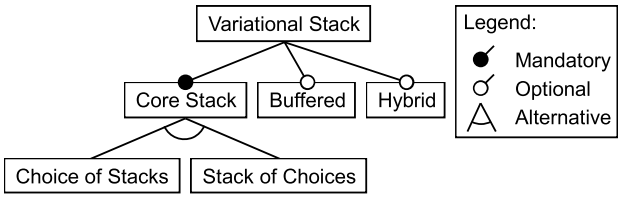


Figure 1: Feature diagram for variational stacks.

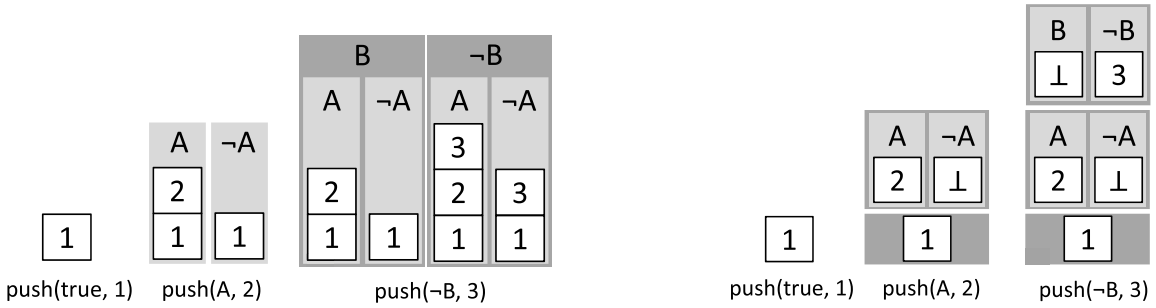
Computation in the JVM centers around operand stacks, so the operand stack is a central data structure that must handle variation efficiently. In the JVM, there are instructions for pushing constants, field values, or local variables onto the stack; arithmetic operations pop inputs from the stack and push their results; and method inputs and outputs are passed via the operand stack. There are many ways variation could be encoded in operand stacks. For example, one possibility is to keep the stack implementation as-is and but manage multiple versions dependent on the variational context (i.e. a choice of stacks). Another possibility is to make the entries in the stack variational (i.e. a stack of choices). Both implementations can represent the same data, but have different, non-obvious, and scenario-dependent effects on performance.

Since the operand stack is central to the JVM, the design of a variational stack has immediate effects on the performance of the interpreter. In the rest of this paper, we explore different designs of a variational stack and investigate how the design decisions affect performance in different scenarios.

3. ALTERNATIVE VARIATIONAL STACKS

To explore the dimensions of the design space of variational data structures, and to evaluate their impact on performance, we need some implementations. In this section, we describe a small family of variational stack data structures, illustrated by the feature diagram in Figure 1. It consists of two mutually exclusive core variational stack implementations and two conceptually independent, optional optimizations, leading to eight different variational stacks. Throughout this section, it is important not to confuse the variation between the different variational stack implementations described in Figure 1, which is an implementation-time concern about which variational stack to pick, and the variation *within* a particular variational stack, which is the runtime concern that variational stacks are designed to handle efficiently.

Each variational stack implements the interface shown in Listing 1. For simplicity, we only discuss the operations `push` and `pop`. Note that both push and pop take an argument `ctx` to indicate in what variational context the operations are performed. The main challenge of designing an efficient variational stack is that the height of the variant stacks can differ if values are pushed and popped in different contexts. In the rest of this section, we show how unbalanced variant stacks can be managed with different tradeoffs, and how these implementations can be further optimized by exploiting properties of different usage profiles. Implementation details and other stack operations, such as `peek` and `switch`, are available as open source at http://meinicke.github.io/VarexJ/.



(a) Choice-of-stacks. **(b) Stack-of-choices.**

Figure 2: Comparison of two different core variational stack implementations.

```
1  interface VariationalStack {
2    void push(FeatureExpr ctx, Conditional value);
3    Conditional pop(FeatureExpr ctx);
4  }
```

Listing 1: Interface for variational stacks.

```
1  class ChoiceOfStacks implements VariationalStack {
2     Conditional<Stack> stack;
3     void push(FeatureExpr ctx, Conditional value) {
4        value.foreach((FeatureExpr c, Object v) -> {
5           push(c.and(ctx), v);
6        });
7     }
8     private void push(FeatureExpr ctx, Object value) {
9        stack = stack.flatMap((FeatureExpr f, Stack s)->{
10          if (f.and(ctx).isContradiction()) {
11             return new One(s); }
12          if (f.andNot(ctx).isContradiction()) {
13             s.push(value);
14             return new One(s); }
15          final Stack clone = s.copy();
16          clone.push(value);
17          return new Choice(ctx, clone, s);
18       });
19    }}
```

Listing 2: Implementation of choice-of-stacks.

3.1 Choice-of-Stacks

One way to implement a variational stack is as a choice among non-variational stacks. Since each stack may be a different size, this design represents unbalanced stacks naturally.

In Figure 2a, we illustrate how a choice-of-stacks stores variational data for a sequence of three conditional push operations. On the second push, the stack splits because of the non-trivial context. Since it only splits on the relevant configuration option A, these two alternative stacks will be shared among all configurations that differ in other, irrelevant options. Also note that as the stacks split, the original value 1 is redundantly stored in the first position of all variant stacks. If operations are performed in many different variational contexts, the number of variant stacks and potential redundancy grows exponentially.

Listing 2 shows the implementation of `push` for choice-of-stacks. A `Conditional` type represents a value that may be variational, encoded as a choice calculus expression with plain values at the leaves. A `FeatureExpr` is a Boolean expression of configuration options, and is used to represent variational contexts. The underlying data structure for the choice-of-stacks implementation is `Conditional<Stack>`, that is, a choice calculus expression with stacks at the leaves. Since the argument `value` is also variational, the `push` operation first iterates over all of the plain values in the argument (line 5). The `push` helper method pushes a plain value to all of the variant stacks. For each variant stack, it checks the variational context of the stack, `f`, against the variational context of the push operation, `ctx`. There are three possibilities: at line 10, the push applies in no contexts that include this stack, so it is returned unchanged (the `One` constructor builds a `Conditional` value with just one variant); at line 12, the push applies in all contexts that include this stack, so the value is simply pushed; at line 15, the push applies to some contexts that include this stack, so we must clone the stack to capture the two different execution paths going forward.

The major drawback of the choice-of-stacks implementation is that if just one value of a stack differs in two contexts, all of the values common to both contexts must be duplicated, which has both space and time implications. If the two variant stacks later converge, identifying and merging them is expensive since we must iterate over all of the variants.

3.2 Stack-of-Choices

The next implementation inverts the relationship of stacks and choices by representing a variational stack as a stack of conditional values (stack-of-choices). The idea is to avoid the cloning required by the choice-of-stacks implementation by using a single stack and using choices within that stack to encode difference among entries in different contexts.

Figure 2b illustrates how stack-of-choices shares the common `1` value among different contexts. To handle unbalanced stack sizes, we conditionally store *null* values ($\perp$) to the stack, which we call holes. For example, when the number 2 is pushed under context A, we push the choice $Choice(A, 2, \perp)$. The rest of this subsection describes how the push and pop operations handle holes in the stack.

The push operation is straightforward. If a value is pushed for the trivial context *true* (meaning the same value is pushed in all contexts), the value can be pushed directly onto the stack. However, if a value is pushed with a non-trivial context, such as A, we must introduce a hole to represent the absence of the value under the contradictory context, $\neg A$.

The pop operation is more complicated since it might need to eliminate holes in order to return meaningful values. For example, consider the rightmost stack of Figure 2b; a pop in context *true* cannot simply return $Choice(B, \perp, 3)$ since there are still values on the stack under context B. Thus,

```
1  class StackOfChoices implements VariationalStack {
2     Conditional[] stack;
3     int top = -1;
4     void push(FeatureExpr ctx, Conditional value) {
5        stack[++top] = new Choice(ctx, value, null);
6     }
7     Conditional pop(FeatureExpr ctx) {
8        Conditional pop = null;
9        for (int i = top; i >= 0; i--) {
10          Conditional current = stack[i].get(ctx);
11          pop = new Choice(ctx, current, pop);
12          stack[i] = new Choice(ctx, null, stack[i]);
13          if (i == top && stack[i].isNull()) top--;
14          ctx = ctx.and(getCtxOfNull(current));
15          if (ctx.isContradiction()) break;
16       }
17       return pop;
18    }}
```

Listing 3: Implementation of stack-of-choices.

```
1  class BufferedStack implements VariationalStack {
2     LinkedList buffer;
3     FeatureExpr bufferCTX;
4     VariationalStack coreStack;
5     void push(FeatureExpr ctx, Conditional value) {
6        if (!bufferCTX.equals(ctx)) {
7           debufferAll();
8           bufferCTX = ctx;
9        }
10       buffer.push(value);
11    }
12    Conditional pop(FeatureExpr ctx) {
13       if (bufferCTX.equals(ctx) && !buffer.isEmpty()) {
14          return buffer.pop();
15       } else {
16          debufferAll();
17       }
18       return coreStack.pop(ctx);
19    }}
```

Listing 4: Buffered-stack decorator.

pop first fills the hole with $Choice(A, 2, 1)$ then removes and returns the value $Choice(B, Choice(A, 2, 1), 3)$.

To handle holes, the pop operation traverses the stack top-down and assembles values from different contexts until either there is no hole in the return value, or the whole stack is traversed. For example, consider again popping a value from the rightmost stack in Figure 2b. When calling pop under context $\neg B$, the top element can be removed and the value 3 returned directly since 3 is a value with no holes. However, if pop is called under B, the hole is filled by traversing the rest of the stack, eventually returning $Choice(A, 1, 2)$. Since pop is applied under context B, the part of the top entry corresponding to context $\neg B$ must be kept on the stack.

Listing 3 shows the implementation of stack-of-choices. The member variable `stack` is an array of conditional values and `top` is the index of the topmost element. The method `push` adds an entry to the stack as a choice with a hole (`null`). Popping a value from the stack works as follows: The variable `pop` is used to incrementally collect values while traversing the stack. Starting from `top`, the value under the current context is retrieved (line 10) and stored in the current `pop` variable (line 11). Next, the popped value is removed from the current position in the stack. If this leaves the top entry empty, `top` can be decremented. To fill the remaining holes in `pop`, the context of the null value is determined by calling `getCtxOfNull`, and this context is used for the next iteration, until there are no holes left (i.e. `ctx` is a contradiction) or all entries are traversed. Finally, the value `pop` is returned.

Stack-of-choices supports more sharing than the choice-of-stacks implementation, but this comes at the cost of complicating stack operations (e.g. pop). In the best case, the pop operation can simply return part of the topmost entry if it does not contain a hole in the given context. In the worst case, however, the whole stack must be traversed to pop a value. We discuss and evaluate these tradeoffs in Section 3.

3.3 Buffered Stack Decorator

Both choice-of-stacks and stack-of-choices support push and pop operations with arbitrary contexts. During the development of VarexJ, we discovered that values are usually popped from the stack under the same context as they were pushed. We turn this insight into a decorator for an underlying core variational stack that exploits this property.

```
1  class HybridStack implements VariationalStack {
2     FeatureExpr stackCTX;
3     VariationalStack stack = new Stack();
4     boolean switched = false;
5     void push(FeatureExpr ctx, Conditional value) {
6        checkParameter(ctx, value);
7        stack.push(ctx, value)
8     }
9     void checkParameter(FetureExpr ctx, Conditional
          value) {
10       if (switched) return;
11       if (!ctx.equals(stackCTX) || !value.isOne()) {
12          // create a variational stack
13          // push all current values
14          switched = true;
15       }
16    }}
```

Listing 5: Hybrid-stack decorator.

The idea is to buffer pushed values in a plain stack as long as the variational context doesn't change. When several sequential pushes and pops are called in the same context, we can just push and pop from the buffer, saving the cost of manipulating variational values. When a push or pop is invoked in a different variational context, the buffered-stack decorator default to the core variational stack implementation, pushing all currently buffered values to the core stack in the context associated with the buffer. The implementation of the buffered stack is shown in Listing 4. The `push` and `pop` operation check whether they were invoked in the same context as the buffer. If not, they fall back to the core implementation using `debufferAll`. As the implementation illustrates, when pushes and pop occur in the same context, there are no map calls at all, so even very large choice values can be pushed and popped with no performance overhead.

3.4 Hybrid Stack Decorator

All of the implementations described so far assume that the stack always needs to handle variational values and operations in different variational contexts. During the development of VarexJ, we discovered that most calls to the stack do not involve variation at all. To exploit this property, we implemented the hybrid stack decorator, shown in Listing 5. The hybrid stack uses a plain stack until a variational stack

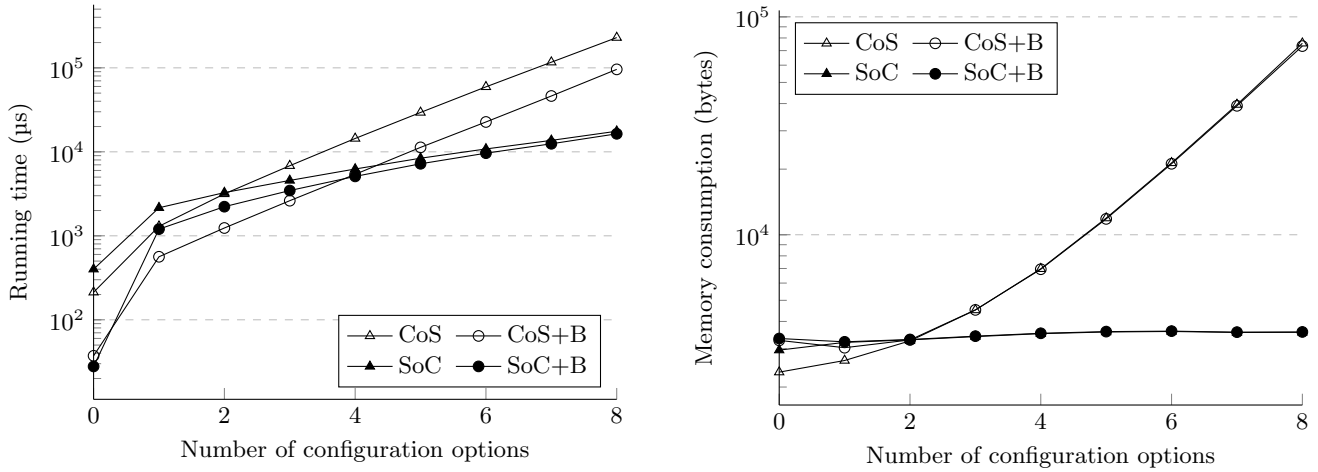


Figure 3: Comparing variational stacks on artificially generated operation sequences with different numbers of configuration options that may appear in the variational contexts. Y-axes are on a logarithmic scale.

is necessary. The implementation of `push` checks whether the context is different from the plain stack or whether the pushed value is variational. If neither is true, it keeps using the plain stack. Otherwise, it switches to a variational stack initialized by the current contents of the plain stacks.

The hybrid stack decorator exploits the fact that in many scenarios, stack operations are simple and do not need a variational stack. The decorator exploits these cases by working with more efficient plain values as long as possible.

4. EXPERIMENTAL ANALYSIS

Choosing the right data structure for an application often depends on both the particular data that will be stored in the data structure, and on patterns of its usage within the application. The same is true for variational data structures, except we must also take into account properties of data and usage related to variability. In this section, we identify a few of these properties and present two sets of experiments to analyze their impact and evaluate the performance of the family of variational stacks described in Section 3.

In the first set of experiments, described in Section 4.1, we focus on understanding how individual properties of data and usage influence the performance of variational stacks in order to help recommend a specific variational stack for a particular use case. In the second set of experiments, described in Section 4.2, we evaluate the performance of variational stacks in a real-world setting, when used as the operand stack of the VarexJ interpreter.

4.1 Analyzing Tradeoffs in Generated Data

In principle, we can view the range of applications for variational data structures as as an n-dimensional space, where each dimension represents a different property of data or usage that influences which variational data structure to choose. If we fully understood this space, we could partition it into regions for each class of data structure (e.g. variational stacks) and then prescribe a particular variational data structure for a given application.

Examples of data and usage properties that impact the performance of variational data structures include: the number of configuration options and unique variants, the density and complexity of variation points within the data, and the ratio and distribution of variational operations. In this subsection, we experimentally analyze the impact of two of these properties on variational stacks: number of configuration options and one aspect of the distribution of variational operations.

4.1.1 Number of Configuration Options

The most efficient variational data structure for an application with only two variants (e.g. in delta execution [20]) is unlikely to be the same as an application where the data varies in 10s or 100s of independent configuration options (e.g. when analyzing software product lines [7]). Within our family of variational stacks, we expect this tradeoff to be illustrated by the choice of which core stack implementation to choose. For a small number of variants, we expect the directness of the choice-of-stacks implementation of Section 3.1 will win out over the relatively more complicated stack-of-choices implementation of Section 3.2. For a large number of variants with enough commonalities, we expect the increased sharing in the stack-of-choices implementation has a chance to pay off. This experiment attempts to identify the threshold of variants where the stack-of-choices core stack implementation pays off.

Experimental setup. For each number of configuration options from 0 to 8, we artificially generate a sequence of 500 push/pop operations. The generated operation sequences are constrained to prevent stack underflow errors, and also to approximate realistic data by preferring simple variational contexts [10] and sequential operations in the same variational context (see Section 3.3). More specifically, each operation sequence is generated in the following way: start by generating a push operation with a random feature selected from among the 0–8 available configuration options and the trivial *true* context; for operation $n + 1$, 90% of operations will use the same variational context while the remainder will choose a new random feature; if any variant stack in the chosen context is empty, produce a push operation, otherwise randomly push or pop; for push operations, choose a random integer or, in 10% of cases, push a choice in a random configuration option between random integers.

For each operation sequence, we measure the runtime[1]

[1]All measurements throughout the paper were performed on a machine with an Intel Core i7-5600 CPU (4 cores, 2.6 GHz), 11.6 GB of RAM, running 64-bit Ubuntu Linux.

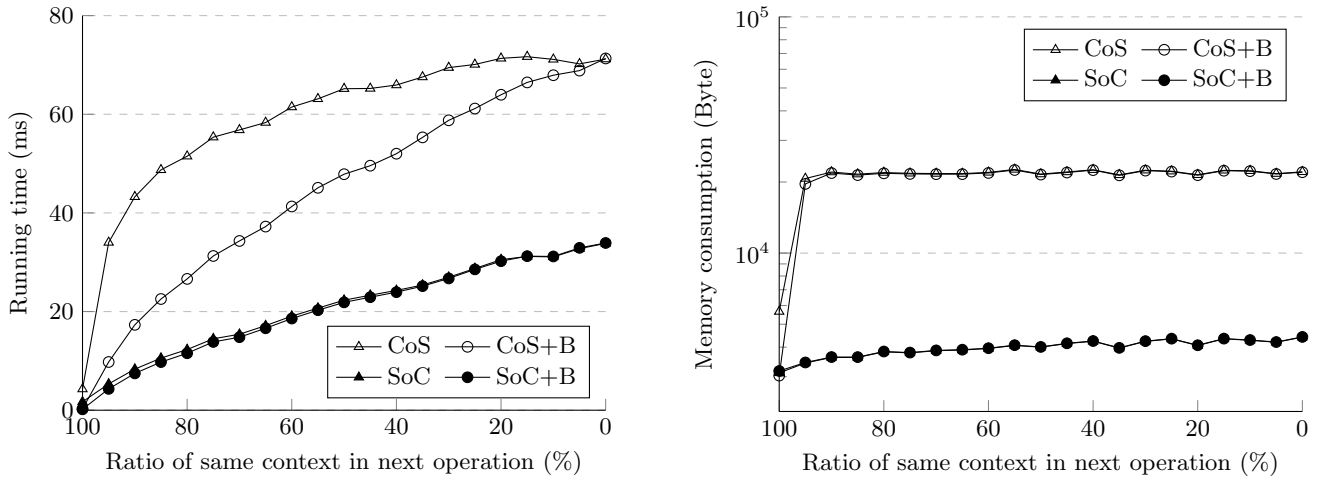


Figure 4: Comparing variational stacks on randomly generated operation sequences with different probability that an operation is executed in the same variational context as the previous operation.

and the maximum memory consumption[2] of four of the variational stacks produced by the product line described in Section 3. Each operation sequence is executed on each stack 10 times, choosing the fastest execution. We omit the four stack variants that include the hybrid optimization since this optimization cannot provide benefits (and only causes small overhead) in such a variation-dense scenario.

Results and analysis. The results of the experiment are presented in Figure 3. The independent variables are: (1) the number of configuration options, plotted on the x-axis, and (2) the choice of stack, indicated by separate lines (*CoS* represents the core choice-of-stacks implementation, *SoC* is the stack-of-choices implementation, and *+B* indicates a stack decorated by the buffered optimization); the dependent variable is the runtime of the operation sequence in the left graph and amount of memory consumed in the right graph.

In general, the precise threshold where the stack-of-choices core stack implementation pays off can depend on many variables that are fixed in the experiment, such as the ratio of operations with a non-trivial context. However, what we observe in the results is that this threshold is quite low for the values of these variables that we analyzed. In our experiment, the stack-of-choices core stack implementation outperforms the choice-of-stacks implementation at three configuration options, and outperforms the buffered choice-of-stacks implementation at five configuration options.

Despite the low threshold for switching from choice-of-stacks to stack-of-choices, we expect there are many applications where this threshold is not exceeded, either because the number of configuration options is low [1, 20] or because they do not interact much [14].

In the memory measurement of Figure 3, we observe the stack-of choices outperforms the choice-of-stacks already for three features. As expected, the choice-of-stacks implementation requires memory that is exponential in the number of configuration options. In contrast, the memory consumption of the stack-of-choices stays almost constant, independent of the number of involved configuration options. This is because adding more configuration options does not increase redundancy in the stack-of-choices, but instead just changes the context associated with each choice in the stack.

[2]https://github.com/meinicke/ObjectSizeMeasure.

4.1.2 Distribution of Variational Operations

In Figure 3 we observe that the buffered implementations outperform their unbuffered counterparts. This is not surprising since in the generated sequences, 90% of operations are in the same variational context as their predecessor, which is exactly the situation the buffered optimization is intended to exploit. In the next experiment we attempt to measure the runtime and memory performance of this optimization with respect to each of our core variational stacks and to the ratio of sequential operations in the same context.

Experimental setup. For r from 0 to 100 in increments of 5, we artificially generate a sequence of 500 push/pop operations where exactly r% of sequential pairs of operations occur in the same variational context. We arbitrarily fix the variation space at six independent configuration options. As before, we constrain the operation sequences to avoid stack underflow errors. For each operation sequence, we measure the runtime of four variational stacks: the two core stacks and the two core stacks decorated by the buffered optimization. As before, each operation sequence is executed on each stack 10 times, choosing the fastest execution.

Results and analysis. The results of the experiment are presented in Figure 4. The independent variables are: (1) the ratio of sequential operations in the same variational context, plotted on the x-axis, and (2) the choice of stack, indicated by separate lines, labeled as before; the dependent variables are runtime and memory consumption of the operation sequence.

We observe that for the choice-of-stacks core implementation, the buffered optimization has a significant effect on runtime and remains profitable all the way until the ratio is nearly zero. In contrast, the buffered optimization has a very small effect for the stack-of-choices core implementation, and the effect nearly vanishes at a relatively high ratio of sequential operation in the same context. This reflects the fact that the stack-of-choices implementation already supports relatively well the scenario that the buffered optimization addresses; for example, pushing and popping a value in the same context will simply push and pop a corresponding choice from the stack. In contrast, the choice-of-stacks implementation would require splitting and copying all variant stacks

	Choice-of-stacks				Stack-of-choices			
	Core	+H	+B	+HB	Core	+H	+B	+HB
Email	556.8	492.0	499.7	517.8	508.1	502.6	506.2	501.7
Elevator	792.2	645.3	610.2	596.4	786.3	651.2	643.7	582.8
ZipMe	8 385.6	4 687.7	5 244.7	4 549.8	6 482.7	4 599.3	5 077.6	4 588.8
GPL	35 962.1	21 043.5	25 866.5	20 466.0	29 188.5	20 822.5	25 637.8	20 713.3

Table 1: Running time (ms) of each variational stack used as the operand stack in VarexJ while executing different variational programs. Each row measures the overall running time of VarexJ using each stack in-situ.

	LOC	Opt	Conf	Stacks	λH	λB	λHB
Email	644	9	40	4 938	195	21	18
Elevator	730	6	20	14 154	1 772	1 499	1 454
ZipMe	2 827	15	10	76 392	93	169	28
GPL	662	15	146	533 162	7 345	500	497

Table 2: Overview of four variational programs. Columns indicate: lines of code (LOC), number of configuration options (Opt), and total number of configurations (Conf) for each example; the total number of operand stacks created while executing the example in VarexJ (Stacks); and the failure rates for the hybrid optimization (λH), buffered optimization (λB), and both optimizations combined (λHB).

on the push operation, even if the next pop operation is in the same context (rendering the copied variants irrelevant).

In the graph of memory consumption in Figure 4, we can see that the memory required by choice-of-stacks is significantly higher than the memory required by stack-of-choices for all ratios except for 100%. For all ratios below 100%, the memory consumption of choice-of-stacks is approximately 21 kilobytes since the stack has to represent all 2^6 variant stacks. The stack-of-choices is again more efficient since common values are shared across variant stacks. For both implementations, the buffered-stack optimization has only minimal effects on memory consumption.

Neither of the experiments in this subsection analyzed variational stacks containing the hybrid optimization described in Section 3.4. The improved performance of hybrid stacks depends on whether variability occurs at all in each stack instance. This is something that is best measured on real-world examples, which we do in the next subsection.

4.2 Variational Stacks in VarexJ

In this subsection we consider how our variational stacks perform in practice, when used as the operand stack in the variational Java interpreter VarexJ (see Section 2.2). We use VarexJ to execute all configurations of four systems that have previously been used as benchmarks in research on configurable software: the systems Email [5] and Elevator [17] are small academic Java programs that were designed with many interacting configuration options, ZipMe[3] is a small open-source library for accessing ZIP archives, and GPL [11] is a small-scale configurable graph library often used for evaluations in the software product line community.

Experimental setup. We use VarexJ to execute each of the four systems, using each of the eight possible variational stacks described in Section 3 as VarexJ's variational operand stack. For each combination of system and stack, we configure VarexJ to use the corresponding stack implementation as its operand stack, then use VarexJ to execute all configurations of the system 10 times, choosing the fastest execution.

Additionally, we count how many total operand stacks are created during the execution of each system. For each stack implementation that includes either the hybrid or buffered optimization, we count how many times these optimizations *miss* during the execution of each system. For the hybrid optimization, the optimization misses when a variational operation is first performed on a particular operand stack. For the buffered optimization, the optimization misses when an operation is first performed on some operand stack in a different variational context than the previous operation.

Results and analysis. The runtime results are presented in Table 1, while Table 2 presents some basic characteristics of each system: lines of code, number of configuration options, and number of unique configurations. Table 2 also shows the number of operand stacks created during the execution of each system in VarexJ, the miss rates for each optimization in isolation (columns λH and λB), and the miss rate of both optimizations combined (λHB).

In the runtime results, we observe that the stack-of-choices core implementation outperforms the choice-of-stacks core implementation for all systems. More interestingly, we observe that both optimizations are highly relevant in practice—either optimization alone produces substantial speedups with the choice-of-stacks implementation, and including both optimizations renders the choice of core stack implementation moot. This suggests that the optimizations capture an overwhelming majority of the cases in this application scenario. This observation is confirmed by the miss rates in Table 2. A core stack is created only when the included optimizations miss, and we observe that the miss rate for both optimizations combined (λHB) is less than 1% of the total operand stacks created in 3 out of 4 systems. The exception is the Elevator system, which was specifically designed to exhibit many interactions [17], and has a miss rate for the combined optimizations of approximately 10%. This still seems low enough that the choice of core stack is insignificant. For larger applications we expect even higher success rates for the optimizations since such systems tend to have fewer interactions between configuration options and lower variation density than our test cases [14].

Overall, the results demonstrate that choosing the right implementation of a variational data structure can have a significant impact on the overall performance of a variational computation, even if that data structure is only a relatively small part of the tool, as is the case for the operand stack in VarexJ. For example, when executing GPL, switching from the choice-of-stacks to the stack-of-choices implementation saves 19% of the overall runtime and adding both optimizations saves overall 43% of the runtime.

[3]https://sourceforge.net/projects/zipme/

5. CONCLUSION

Variational data structures are needed to efficiently compute with variability in data and code. Toward a systematic understanding of the design space for variational data structures, we have presented a family of variational stack implementations. We evaluated the performance of these variational stacks when used as the operand stack in the variational interpreter VarexJ. The results demonstrate that the choice of variational data structure can have a significant impact on the performance of a program that computes with variability.

A distinguishing feature of the application domain targeted by VarexJ is that it involves the creation of many short-lived stacks, where relatively few contain variation in multiple different variational contexts (see Section 4.2). As future work we should evaluate the family of variational stacks in application scenarios that involve longer-lived variational stacks with different variability profiles.

Acknowledgments

Thanks to Hanzhong Xu for ideas on improving the stack-of-choices implementation. This work is supported by AFRL Contract FA8750-16-C-0044 (Raytheon BBN Technologies) under the DARPA BRASS program, the NSF awards 1318808 and 1552944, the Science of Security Lablet (H9823014C0140), and AFRL and DARPA (FA8750-16-2-0042).

6. REFERENCES

[1] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted Execution of Policy-Agnostic Programs. In *Proc. Workshop Programming Languages and Analysis for Security (PLAS)*, pages 15–26. ACM, 2013.

[2] S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. Programming Languages and Systems (TOPLAS)*, 36(1):1:1–1:54, 2014.

[3] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *Trans. Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011.

[4] M. Erwig and E. Walkingshaw. Variation Programming with the Choice Calculus. In *Proc. Generative and Transformational Techniques in Software Engineering*, pages 55–100. Springer, 2013.

[5] R. J. Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering (ASE)*, 12(1):41–79, 2005.

[6] S. Hubbard and E. Walkingshaw. Formula Choice Calculus. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 49–57, 2016.

[7] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM, 2011.

[8] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-Aware Testing. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 1–8. ACM, 2012.

[9] C. H. P. Kim, S. Khurshid, and D. Batory. Shared Execution for Efficiently Testing Product Lines. In *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*, pages 221–230. IEEE, 2012.

[10] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. IEEE, 2010.

[11] R. E. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. Int'l Symposium Generative and Component-Based Software Engineering (GCSE)*, pages 10–24. Springer, 2001.

[12] J. Meinicke. VarexJ: A Variability-Aware Interpreter for Java Applications. Master's thesis, University of Magdeburg, 2014.

[13] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, and G. Saake. An Overview on Analysis Tools for Software Product Lines. In *Proc. Workshop Software Product Line Analysis Tools (SPLat)*, pages 94–101. ACM, 2014.

[14] J. Meinicke, C. P. Wong, C. Kästner, T. Thüm, and G. Saake. On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 483–494. ACM, 2016.

[15] K. Muslu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative Analysis of Integrated Development Environment Recommendations. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 669–682. ACM, 2012.

[16] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 907–918. ACM, 2014.

[17] M. Plath and M. Ryan. Feature Integration Using a Feature Construct. *Science of Computer Programming (SCP)*, 41(1):53–84, 2001.

[18] W. N. Sumner, T. Bao, X. Zhang, and S. Prabhakar. Coalescing Executions for Fast Uncertainty Analysis. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 581–590. ACM, 2011.

[19] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):6:1–6:45, 2014.

[20] J. Tucek, W. Xiong, and Y. Zhou. Efficient Online Validation with Delta Execution. *ACM SIGARCH Computer Architecture News*, 37(1):193–204, 2009.

[21] E. Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, 2013.

[22] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *Proc. Int'l Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, pages 213–226. ACM, 2014.