Shared Execution for Efficiently Testing Product Lines

Chang Hwan Peter Kim
Dept. of Computer Science
University of Texas at Austin, USA
chpkim@cs.utexas.edu

Sarfraz Khurshid
Dept. of Electrical and Computer Engineering
University of Texas at Austin, USA
khurshid@ece.utexas.edu

Don Batory
Dept. of Computer Science
University of Texas at Austin, USA
batory@cs.utexas.edu

***Abstract*—A *software product line (SPL)* is a family of related programs, each of which is uniquely defined by a combination of features. Testing an SPL requires running each of its programs, which may be computationally expensive as the number of programs in an SPL is potentially exponential in the number of features. It is also wasteful since instructions common to many programs must be repeatedly executed, rather than just once. To reduce this waste, we propose the idea of *shared execution*, which runs instructions just once for a set of programs until a variable read yields multiple values, causing execution to branch for each value until a common execution point that allows shared execution to resume. Experiments show that shared execution can be faster than conventionally running each program from start to finish, despite its overhead.**

***Keywords*-product lines; feature oriented programming; testing; shared execution; dynamic analysis.**

I. Introduction

A *Software Product Line (SPL)* is a family of programs in which each program is defined by a unique combination of features. Developing a set of programs with commonalities and variabilities in a systematic way can significantly reduce both the time and cost of software development. However, the fact that an SPL can represent many programs, a number potentially exponential in the number of features, makes testing SPLs more expensive than testing ordinary programs. Just 10 optional features can produce over a thousand (2^{10}) distinct programs, meaning that a test case now has to be run not once, but over a thousand times.[1]

The combinatorial problem can be tamed when some features are irrelevant to the property being tested. For example, if we can detect, before executing a test case, that the code for 8 of the 10 optional features is not even reachable, then the test case need only be run on 4 rather than over a thousand programs. Previously, we developed static analyses to tame combinatorics this way [14][15].

Unfortunately, combinatorics cannot be tamed when most of the features are relevant to the test case and they interact, meaning the test case must be run on each combination. Also, a test case may require every combination to be run *by design* because it expects each program to produce unique behavior. Thus the previously developed static analyses, and static analyses in general, are ineffective in this setting, which brings us to this paper.

Although combinatorics cannot be tamed in this setting, we can still do better than just running the test case against each program. Because programs in an SPL are syntactically similar (i.e. share common code) by design, they are likely to have semantic commonality as well. Namely, it is likely that many bytecode instructions executed across the programs will be identical. In this paper, we present the idea of *shared execution*, which essentially "product lines" execution, executing common instructions once, rather than multiple times, to eliminate redundancy and reduce execution time. Shared execution runs an instruction common to multiple program executions just once by using a single call stack and memory that keeps track of each program's data. This paper makes the following contributions:

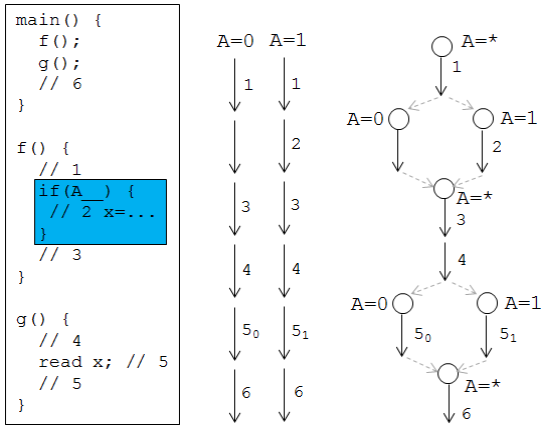
- **Technique.** We define shared execution as a bytecode level algorithm that can be implemented on top of any virtual machine (VM).
- **Implementation.** We implement shared execution on top of *Java PathFinder (JPF)* [24], a model checker for Java that can also function as an easy-to-extend, off-the-shelf VM. We use only the VM portion of JPF.
- **Evaluation.** We show, using non-trivial subjects used in prior publications of other research groups, that shared execution, despite its overhead, can run a product line test case up to 50% faster than the conventional way of running the test case for each configuration from start to finish.

II. Shared Execution: Basic Technique

An SPL is a family of programs defined by feature combinations and corresponding code. A *feature model* defines the legal feature combinations, which form a subset of 2^N combinations possible with N *optional features*. [2]

[1] A *test case*, with each variable set to a concrete value, represents a single path through a test. For our purposes, a *test* is an arbitrary program.

[2] Henceforth, an optional feature will simply be referred to as a *feature* since the paper focuses on optional features. Arbitrary constraints, such as XOR, may be applied on optional features.



(a) Sample SPL (b) Conventional execution (c) Shared execution

Figure 1. Shared Execution on Sample SPL

One feature combination or *configuration* corresponds to one program. A feature maps to code scattered throughout the program such that the corresponding code is syntactically present if and only if the feature is present. There are different ways to represent a feature model and to map features to code. For this paper, a feature model is represented as a context-sensitive grammar [1] and features are mapped to code by using *feature variables*, which are simply Java Boolean variables (so an SPL is an ordinary program with if-conditions [16]).[3]

Figure 1(a) shows the skeleton of a typical SPL that has six fragments of code labeled as comments, whose numbers indicate the order in which the fragments are executed. In our example, the feature `A` is mapped to code by feature variable `A__` (code executed when the feature is `true` is colored for visual aid). Note that a program with a feature included can behave differently from a program without that feature. For example, `x` could have two different values in executions of fragment 5 and subsequently two different program behaviors, depending on whether `A` is selected or not. More generally, in a product line with N features, fragment 5 could yield up to 2^N different values of `x` and trigger up to as many different program behaviors.

Conventionally, different program behaviors are produced by first determining the configurations allowed by the feature model, where a configuration is an *n*-digit boolean $f_1..f_n$ that represents feature assignments (*true* or 1 representing a feature's presence and *false* or 0 its absence). Then for each configuration, the corresponding program is run from start to finish. Figure 1(b) shows this process for the example. Note the similarities between the execution traces: the only differences are that the A=1 configuration executes fragment 2 while A=0 does not and that fragment 5 behaves differently due to the configurations having different values of `x`. Note that the majority of computations are repeated across both traces. Although a computation can only be repeated once in this example, it can be repeated in general up to $2^N - 1$ due to 2^N traces.

Instead of running each configuration separately, our idea is to execute all the configurations together, executing a computation just once for the configuration(s) for which it is shared. Figure 1(c) shows the resulting trace, which is essentially a superimposition of all the traces produced in conventional execution in Figure 1(b). Note how fragments 1, 3, 4 and 6 are executed once, rather than twice. Fragments in branches, such as the unnumbered fragment and fragment 2, are not shared by all the configurations and each branch is executed one after another before shared execution resumes.

We now present the basic ideas behind our approach.

A. Bookkeeping

Since each variable can have as many different values as there are configurations, memory M must be able to map a variable v *and* a configuration c to a value o. Namely, $M : V \times C \rightarrow O$. A *variable* refers to any data storage that can be accessed by a programmer-defined symbol, i.e. fields (including array elements) and local variables. Conceptually, memory can be thought of as an array of length up to 2^N with one array element holding the memory of one product line configuration. Section IV will present a more efficient representation.

We define *shared execution* as executing instructions for a set of configurations using a common call stack, which stores information including active method calls, program counter of the current method call, and instruction operands to represent a point in program execution or *execution point*. Note that since we are treating local variables as part of memory, a stack frame of the call stack only includes the stack operands and associated attributes (e.g. which operands are references). Before (or after) each instruction execution, there exists a *state* `S` with configuration set $S_{configs}$ and the call stack $S_{callstack}$ in use. At the beginning of shared execution, $S_{configs}$ has all configurations of the product line and $S_{callstack}$ just has the main function's stack frame whose PC is set to the first instruction.

B. Splitting

Since the idea of shared execution is to use one call stack for multiple configurations, instructions execute as they would for conventional programs except that loads and stores must now access memory from a configuration set. Assigning value o to a variable v in state S simply means that $\forall c \in S_{configs}, M(v, c) = o$. Loading is a bit trickier. If $M(v, c)$ returns the same o for every c in set $S_{configs}$, shared execution continues with the read value o. But if

[3]Our technique is not dependent on any feature model representation. In fact, because a feature model is simply a set of feature combinations for our technique, we do not discuss feature models in depth.

$M(v, c)$ returns multiple values, shared execution cannot continue since we can only push one unique value on the call stack. Shared execution now splits: there will be as many call stacks as there are unique values. Namely, the current state S's children states, $S_{children}$, are created such that:

- The union of every child state's configuration set equals the parent configuration set, $S_{configs}$,
- Each configuration in a child state's configuration set holds the same value of v,
- Each child state's call stack is set to a clone of $S_{callstack}$.

We say that S is *split* into children states with respect to v. Each child state, which is ready to have a unique value of v pushed onto its call stack, is set as the current state and executed from where the splitting occurred. As execution proceeds, if a load of another variable y yields multiple values, the executing state will be split in a similar way.

In Figure 1(c), right before `A__` is read, S has $S_{configs}$ equal to $A = *$ (the wildcard, which represents all possible values, is used to represent multiple configurations concisely for presentation purposes). So $A = *$ represents $\{A = 0, A = 1\}$). When the read occurs, S is split (dashed lines outwards) into $S_{A\_\_=0}$ and $S_{A\_\_=1}$ since `A__` has different values: 0 in A=0 and 1 in A=1.[4] The first child state (circle labeled A=0) runs and then execution backtracks to load the call stack of the second child state (circle labeled A=1) for it to run.

C. Merging

We could execute each child state until the end of the program, but doing so, we would miss out on opportunities to share execution after splitting. For optimal shared execution, we should wait until all children states come to a common execution point, i.e. where their call stacks are equivalent, and then resume shared execution with the same call stack.

There are two issues to finding a common execution point. First, the children states could have considerably different paths of execution. Second, finding a common execution point close to the splitting point allows sharing to resume earlier, but it may require more checks, each of which comes with the cost of comparing potentially up to 2^N call stacks, where N is the number of features.

1) Conservative Merging: A reasonable compromise that addresses both issues, which we call *conservative merging*, is to wait until each child state's execution reaches the end of the method (just before a return statement) of where splitting occurred since each child is guaranteed to reach this execution point.[5]

[4]Thus, a feature variable is treated like an ordinary variable in shared execution.

[5]A child state's execution may not reach the end of the method due to abnormal program execution (e.g. exception or system exit), in which case the child state can be simply executed until the end of the program and shared execution resumes with the remaining children. Our implementation currently does not handle abnormal program execution.

Although a return value is in practice written and read off of the call stack, for uniformity in explanation, we treat a return value as a variable (written and read off of memory) throughout the paper. This means that even with different return values, the children states would have the identical call stack at the function return, allowing shared execution to resume. Then splitting would occur when the return value is read *after* the function returns.

For example, in Figure 1(a), when the state executing fragment 1 splits, A=0 and A=1 children states would execute until the end of fragment 3 and then merge, allowing fragment 4 to be shared. Then fragment 5 would cause the state to split again and merge at the end of fragment 5, allowing fragment 6 to be shared. But note that the proposed solution is not as optimal as Figure 1(c) in that the former does not allow fragment 3, or any instruction executed between the splitting point and the end of the method, to be shared.

2) Predictive Merging: *Predictive merging* improves on conservative merging by using what we call a *merge point*. When splitting occurs, we determine an optimistic merge point, i.e. an execution point *before* the end of the method that each child state is likely, but not guaranteed, to reach. We then execute each child state until it reaches this optimistic merge point or the pessimistic merge point, i.e. the end of the method where splitting occurred. If the children have all stopped at the same execution point, i.e. all are at the optimistic or all are at the pessimistic merge point, we resume shared execution with the parent state. Otherwise, we execute each child stopped at the optimistic merge point until the pessimistic, but guaranteed to be common, merge point and then resume shared execution.

In Section IV-B, we discuss in detail how optimistic merge points are determined. For now, it suffices to know that when splitting is due to a read of a boolean variable that is followed by an if-statement, such as `if(A__)`, the optimistic merge point is determined to be the end of the if-statement since programs are typically written such that both `true` and `false` branches will end up at this point. If the `true` branch executes a control-flow breaking instruction such as a `return`, shared execution will resume at the pessimistic merge point.

As an illustration, in Figure 1(a), when the state executing fragment 1 splits, the optimistic merge point is set to the end of the `if(A__)` block. The children states would execute until the *beginning* of fragment 3 and then merge, allowing fragment 3 and 4 to be shared as Figure 1(c) shows. But if there is a `return` within the if-condition, merging would not be possible the first time around since `A=1` state would end up at the end of the method while `A=0` ends up at the beginning of fragment 3. Then state `A=0` would be executed from the beginning of fragment 3 until the end of the method before merging with state `A=1`.

```
1  class SharedExecution extends VMListener {
2    State s = new State(legalConfigs, VM.getCallstack());
3    SPLMemory m = new SPLMemory(legalConfigs);
4
5    void beforeInstruction(Instruction insn) {
6      if(s.isAtMergePoint())
7        tryMerge();
8      else if(isLoad(insn)) {
9        s.children = split(s, getVariable());
10       if(s.children != null) {
11         s.mergePoints = getMergePoints();
12         loadState(s.children.get(0));
13       }
14       else
15         VM.load(getVariable(), m.values
16                 (getVariable(), s.configs).first());
17     }
18     else if(isStore(insn))
19       m.set(getVariable(), s.configs, VM.getTopValue());
20   }
21
22   void tryMerge() {
23     if(s.isLastRemainingChild()) {
24       if(atSameExecPoint(s.parent.children) {
25         s.parent.callstack = s.parent.children.
26                              get(0).callstack;
27         loadState(s.parent);
28       }
29       else {
30         s.parent.setMergePoint(RETURN_MERGE_POINT);
31         loadState(s.parent.nextRemainingChild());
32       }
33     }
34     else
35       loadState(s.parent.nextRemainingChild());
36   }
37
38   void loadState(State t) {
39     s = t;
40     VM.changeCallstack(s.callStack);
41   }
42 }
```

Figure 2. Shared Execution Algorithm

D. Putting Ideas Together

The ideas of bookkeeping, splitting, and merging can be summarized in an algorithm that intercepts every bytecode instruction, as shown in Figure 2. As shared execution works with call stacks, the algorithm must be written in terms of bytecode instructions.

1) Initialization: The legal configurations are determined by applying an off-the-shelf SAT solver against the feature model. S, which represents the state before the currently executing instruction, is initialized so that $S_{configs}$ is set to these configurations and $S_{callstack}$ is identical to the VM call stack (line 2). Memory M requires legal configurations (line 3) to explicitly map feature variables to values across configurations since feature variables are special Boolean variables that can only be read from but not written to. For example, for feature variable `A___`, configuration `A=1` maps to the value 1 and `A=0` to the value 0.

2) Loads and Stores: If a load returns multiple values for a variable, the state is split into children such that each child has configurations that map to the same variable value and a clone of the executing call stack. The parent state keeps track of both optimistic and pessimistic merge points where all of its children will stop their execution (line 11). Then the first child state is set as the current state and the executing (VM's) call stack is changed to the current state's call stack (lines 12, 39 and 40). On the other hand, if the load returns one value of the variable, the value is simply pushed on the stack as it would be in conventional execution (line 15). Note that the load of a feature variable, whose values across configurations have been explicitly stored during initialization, will be what first triggers splitting.

A store just sets the variable value for each configuration of the current state (lines 18-19).

3) Merge: Following a split, we check if the current state is at a merge point, i.e. its call stack is at a return instruction (or an earlier instruction for an optimistic merge point) and the stack's depth is equal to that of where splitting occurred (line 6).[6] If the state is at a merge point, we backtrack execution to the next child state (line 35), allow it to execute until a merge point, and repeat the process until the last child comes to a merge point (line 23). Then, if the children's call stacks are not identical, children are executed up to the pessimistic merge point, i.e. the end of the method (lines 24, 30, 31). At this point, the parent state's call stack takes a child's call stack and is set as the current state (lines 26 and 27), completing the merge.[7]

III. Example

We demonstrate shared execution on the example SPL in Figure 3(a) (line numbers in this section refer to this figure). Features `A___` and `B___` are simply boolean variables that are assigned concrete values for a particular program. An SPL test case is simply an execution of the main method with all variables, except the feature variables, assigned concrete values. The test case must be run on all 4 combinations of the feature variables.

A. Splitting and Merging

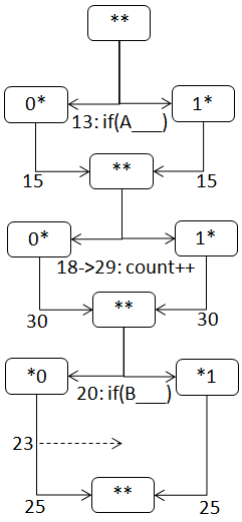
Figure 3(b) shows how states split and merge throughout execution. Splitting first occurs in line 13 because `A_` has multiple values, i.e. 1 (`true`) for the configurations `AB`=1* and 0 (`false`) for the configurations `AB`=0*. Therefore, as the top of Figure 3(b) shows, the state with configurations `AB`=** is split into two children. The optimistic merge point is set to the end of the if-statement (line 15). The 0* state does nothing and ends up on line 15. Then the 1* state appends 2 and ends up on line 15. The two states merge because their call stacks are identical on line 15. Figure 4 shows the memory snapshot at line 17. Note that line 16 computation is shared by all configurations.

[6]Note that we use the stack depth because comparing method signatures alone will not suffice due to recursion.

[7]Note that the merge attempt is the first step in `beforeInstruction()` because processing a load instruction before merging can end up splitting the current state without taking its siblings into account, which can reduce the configuration set corresponding to a unique value of a variable, which in turn reduces sharing.

```
1  class Example {
2    static boolean A___;
3    static boolean B___;
4
5    static char[] array =
6                new char[5];
7    static int count = 0;
8    static int x = 0;
9
10   static void main(){
11     append('1');
12
13     if(A___)
14       append('2');
15
16     x++;
17
18     append('3');
19
20     if(B___) {
21       return;
22     }
23
24     x++;
25   }
26
27   static void append
28                 (char c){
29     array[count++] = c;
30   }
31 }
```

(a) Product Line Test



(b) Split Merge Diagram

Figure 3. Example Product Line

Config (A_B_)	String	X
00	"1"	1
01	"1"	1
10	"12"	1
11	"12"	1

Memory snapshot at line 17

Config (A_B_)	String	X
00	"13"	2
01	"13"	1
10	"123"	2
11	"123"	1

Memory snapshot at line 25

Figure 4. Memory snapshots

Appending 3 causes a split because line 18 leads to line 29, where reading `count` for increment yields multiple values (1 for configurations 0* and 2 for configurations 1*). The split states merge in line 30. Then, when line 20 causes a split due to reading `B_`, optimistic merge point is set to line 23, the end of the if condition. Although the state *0 hits this merge point, the state *1 returns and misses it, causing merge at the conservative merge point (line 25). Figure 4 shows the memory snapshot at the end of the test case (line 25). Note that although each configuration produces a distinct row or test case output, each computation was shared by at least 2 configurations.

IV. Shared Execution: Optimizations

For shared execution to be practical, we need an efficient memory representation and an optimistic merging strategy. Also, garbage collection needs to be modified to understand product lines. In this section, we discuss these issues.

A. Memory

As mentioned in Section II-A, to access a variable value, memory must now be addressed by both the variable *and* a configuration. The easiest way to structure memory would be to allocate a value slot for each configuration and variable pair. However, this would be wasteful for the following reasons.

1) Ownership: A variable can only exist in the configurations that created it. Suppose that a method `bar()` is called in a state with configurations where $A = true$, as shown in Figure 5. The local variables allocated for that method call, such as `b`, cannot exist in configurations where the feature is absent and therefore do not even need to have storage for these configurations. Similarly, an object and its fields cannot exist in a configuration that is not a part of the state that created the object. Thus, it would be impossible to access object 2 and its fields in a configuration with $A = false$, while object 1 and its fields can be accessed in any configuration (since the object is owned by all configurations). We say that a variable's *owner* is the configuration set that created it either through a method call (for local variables) or through an object creation (for instance fields). For a variable, only as many value slots as there are owner configurations are allocated.

```
1  class Program {
2    Obj x = new Obj(); // 1
3
4    void foo(){
5      if(A) {
6        bar();
7        x = new Obj(); // 2
8      }
9      m(x);
10     ...
11   }
12
13   voix bar(){
14     int b = 5;
15     ...
16   }
17 }
```

Figure 5. Memory Example

2) Small number of unique values: A variable owned by M configurations could have M unique values, but chances are that there will be far fewer number of values. The reason is a variable is owned by configurations with the creating feature present. For the variable to have as many values as there are owner configurations, the creating feature must interact with *all* the other features, which is possible but less likely than interacting with just some of the other features. Moreover, there will be variables used only by the creating feature, i.e. by the owner configurations, to achieve computations relevant only to it.

3) Dual Memory: We exploit the notion of ownership and small number of unique values in the following way. Since a variable having one value across its owner configurations is no different than a variable in conventional execution,

we partition memory into two storages: 1) *conventional storage*, i.e. memory created by VM that maps a variable to a value ($M_1 : V \rightarrow O$) and 2) *multivalued storage*, i.e. memory created by our technique that maps a variable *and* a configuration to a value ($M_2 : V \times C \rightarrow O$) and is used to store variables with multiple values. This representation, called *dual memory*, exposes how loads and stores work in Figure 2:

- **Loads.** If a variable exists in multivalued storage, a load is performed for each configuration of the current state and splitting occurs if there is more than one unique value. If the variable doesn't exist in multivalued storage, then the variable is loaded from the conventional storage.
- **Stores.** A write could occur with the current state's configurations $S_{configs}$ spanning either 1) owner configurations or 2) a subset of the owner configurations. In scenario 1), the variable will have the same value across owner configurations after the write. Therefore, the write is performed just once for the conventional storage and the variable is removed from the multivalued storage since it no longer has multiple values. In scenario 2), $S_{configs}$ will take on the new value in multivalued storage. If the variable exists in the conventional storage, its value is transferred to multivalued storage (to the complement of $S_{configs}$ with respect to the owner configurations).

For example, in line 16 of Figure 3, `x` is read from and written to conventional storage, but in line 24, `x` is transferred to multivalued storage. In line 11, `count` and `array[1]` are in conventional storage, but by line 15, they are in multivalued storage. `array`, `array[0]`, `array[3]` and `array[4]` remain in conventional storage throughout the test case.

B. Optimistic Merging

As mentioned in Section II-C2, when splitting occurs, we want to merge before the end of the method to potentially share the remaining instructions of the method. Although we could execute each child one instruction at a time and check if their call stacks are equivalent, this would perform checks too often. Thus, we use merge points, which are candidate execution points for merging, to minimize the number of call stack comparisons and maximize the likelihood of a comparison succeeding. Note that because children will merge within the same function where splitting occurred (modulo abnormal program execution discussed earlier), only the top frames of the children's call stacks need to be checked for equivalence, meaning the cost of comparison boils down largely to the number of children, i.e. the number of unique values of the variable whose load caused the split.

If the variable load causing the split is followed by a conditional instruction, the merge point is set to the target instruction since both `true` and `false` branches are likely to end up there. If the variable load is not followed by a conditional instruction, an instruction following a store instruction or a method invocation is treated as a merge point. This is because a variable is likely to be loaded for computing a value to write to another variable or for invoking a method with the value as one of its arguments. Also, if the children all come to the same instruction following the same store instruction or the same method invocation, call stack comparison is likely to succeed since the different values have already been popped off the call stack. So these merge points are ideal places to compare call stacks. If the comparison fails, the children are executed from where they are until the end of the method is reached (i.e. the conservative merge point), where the call stacks will be equivalent.

For example, in Figure 5, loading `x` as an argument for method invocation `m()` will cause a split since `x` will point to object 1 in configurations without `A` and to object 2 in configurations with `A`. The end of the method invocation, i.e. right before line 10, will be set as the optimistic merge point.

C. Garbage Collection

As shared execution requires more memory than running each configuration separately, garbage collection is especially important for optimal performance. Unfortunately, we cannot use the VM's garbage collection as-is since it is not aware of product lines. For example, it will collect objects that are still alive for some configurations. Another reason why we need modify the conventional *garbage collector (GC)* is to clean up our own multivalued storage.

In Figure 5, `foo()` is executed up to line 9 once with $A = true$ and once with $A = false$. At this point, as far as the conventional GC can tell, `x` can point to either object 1 or object 2 but not both, meaning one of the objects will be garbage collected, which is clearly erroneous since `m()` must be invoked with each object as its argument.

Shared execution GC is a simple modification of conventional GC such that objects are marked starting from the call stacks of leaf states of the hierarchy of states, rather than just from the executing call stack. Note that only the leaf states have call stacks and the non-leaf states exist for splitting/merging purposes. Also, the marking phase is changed such that if a variable has multiple values across its owner configurations, multiple references will be marked as being alive. Finally, when an object is garbage collected, shared execution GC removes variables corresponding to its fields from the multivalued memory. Note that nothing needs to be done with the object itself as an object is not a variable that can have different values across configurations.

V. Evaluation

Shared execution can be implemented on top of any virtual machine, such as Jikes RVM [11] or Java PathFinder (JPF) [24]. Although JPF is typically used as a model checker rather than as a VM, we chose it as our platform because of its extensibility and our familiarity with it. As a VM, JPF is considerably slower than an ordinary VM, but as we are running both shared execution and non-shared execution using JPF, using it should not affect our results. We evaluated shared execution on 3 subjects: Graph Product Line (GPL) (originally appeared in [20]), JTopas [10], and XStream [19], that have also been used for evaluating testing techniques by other groups (GPL by [2], JTopas by [3] and XStream by [8][25]). Our shared execution implementation, subjects, and results can be downloaded from [13].

A. Graph Product Line (GPL)

GPL encodes programs that implement different graph algorithms. The product line, with 1713 LOC, 14 features and 146 configurations, was developed in our research group, but long before this paper was written [20]. Note that there are only 146 configurations despite 14 features due to constraints. The features represent various graph algorithms and structures (e.g. directed/undirected and weighted/unweighted). Due to lack of tests for this product line, we generated graphs and ran the main method on each graph against all configurations. Although many algorithms are independent, some features do interact to produce different outcomes. For example, `DIRECTED` will change the outcome of `CYCLEDETECTION` and combinations of `BFS` (Breadth First Search) and `DIRECTED` may change the outcome of `NUMBERING` (how nodes are numbered).

Table I shows the results for GPL. As the headers show, three types of graphs were generated: 1) sequence of neural networks (each with one input node and one output node), 2) trees with a fixed degree, and 3) random graphs with a fixed number of nodes and an average degree generated using an off-the-shelf random graph generator [9]. *No. of test case instrcts* only includes the bytecode instructions executed by the test case, and does not include instructions from the shared execution implementation. The table shows that shared execution executes about 1/20th of the test case and saves between 24% and 53% of execution time over the conventional approach of running the test case against each configuration from start to finish. Note that shared execution saves time for the larger network and tree. Time saving stays consistent across the random graphs, whose numbers of nodes and edges are nearly the same, suggesting that the results are probably representative of other graphs with similar numbers of nodes and edges.

Table I
GPL Results

	Conventional	Shared Execution	Factor/Saving (%)
Network 1: 45 nodes, 80 edges			
No. of test case instrcts	76702636	4024943	19:1
Duration (sec.)	25	19	24%
Network 2: 221 nodes, 400 edges			
No. of test case instrcts	2951629460	146355025	20:1
Duration (sec.)	503	277	45%
Tree 1: 85 nodes, 84 edges			
No. of test case instrcts	234426320	11907505	20:1
Duration (sec.)	51	36	29%
Tree 2: 341 nodes, 340 edges			
No. of test case instrcts	9318420952	448314944	21:1
Duration (sec.)	1619	756	53%
Random 1: 101 nodes, 374 edges			
No. of test case instrcts	450951320	23119790	20:1
Duration (sec.)	101	66	35%
Random 2: 101 nodes, 381 edges			
No. of test case instrcts	431427468	22113491	20:1
Duration (sec.)	95	61	36%
Random 3: 101 nodes, 372 edges			
No. of test case instrcts	449992982	23077201	19:1
Duration (sec.)	98	65	34%
Random 4: 101 nodes, 362 edges			
No. of test case instrcts	429500846	22067178	19:1
Duration (sec.)	94	61	35%
Random 5: 101 nodes, 336 edges			
No. of test case instrcts	431307370	22318788	19:1
Duration (sec.)	93	62	33%

B. JTopas

JTopas [10] is an open source Java program for parsing text that has 2031 lines of code. We converted this conventional program into an SPL simply by modifying Boolean configuration flags into Boolean feature variables that our shared execution tool can recognize: `LINECOMMENTS`, `BLOCKCOMMENTS`, `COUNTLINES`, `IMAGEPARTS`, and `TOKENPOSONLY`. If `COUNTLINES` is `true`, each token will have line and column information. `IMAGEPARTS` gives each token's string more structure, such as breaking it into lines. `TOKENPOSONLY` represents a token by its position in the original text, rather than string. `LINECOMMENTS` and `BLOCKCOMMENTS`, which return a single token representing a line comment or a block comment respectively if the feature is on and skips the corresponding characters if the feature is off, change the result of tokenizing an input embedded with comments significantly. These 5 features yield 32 configurations.

We simplified an existing test called `TestLargeSource`, which tokenizes a Java class containing some methods, to run against the configurations. We created 9 test cases out of this test to not only test inputs of different sizes, but also inputs that are expected to result in different amount of instruction sharing. Since we expect tokenization across configurations to be more different the more comments there are, we used test cases with varying number of comments to see if shared execution results are consistent with this expectation. The Java code input for `Many` test cases is shipped with JTopas. `Some` and `Without` test cases simply take `Many`'s code input and remove some or all comments respectively. The test case number `N` (e.g. `Many N`) means the code input is tokenized `N` times. Table II shows that instruction sharing, and consequently time saving, indeed

Table II
JTOPAS RESULTS

	Conventional	Shared Execution	Factor/Saving (%)
Many comments 1			
No. of test case instrcts	38195022	14988848	2.5:1
Duration (sec.)	16	25	-56%
Many comments 2			
No. of test case instrcts	75706878	30919876	2.4: 1
Duration (sec.)	27	49	-81%
Many comments 3			
No. of test case instrcts	113319726	46912432	2.4:1
Duration (sec.)	39	60	-53%
Some comments 1			
No. of test case instrcts	34283622	3967675	8.6:1
Duration (sec.)	15	14	6.7%
Some comments 2			
No. of test case instrcts	67826606	12661565	5.4:1
Duration (sec.)	26	28	-7.7%
Some comments 3			
No. of test case instrcts	101424102	21416331	4.7:1
Duration (sec.)	33	39	-18%
No comments 1			
No. of test case instrcts	33245790	2421775	14:1
Duration (sec.)	14	14	0%
No comments 2			
No. of test case instrcts	65735326	4824448	14:1
Duration (sec.)	24	19	20%
No comments 3			
No. of test case instrcts	98281742	7234081	14:1
Duration (sec.)	34	26	24%

Table III
XSTREAM RESULTS

	Conventional	Shared Execution	Factor/Saving (%)
0 Common, 10 Variable			
No. of test case instrcts	99814860	16308440	6.1:1
Duration (sec.)	32	27	15%
0 Common, 20 Variable			
No. of test case instrcts	174168126	31258531	5.6:1
Duration (sec.)	49	43	12%
0 Common, 30 Variable			
No. of test case instrcts	248489238	46172217	5.4:1
Duration (sec.)	67	59	11%
6 Common, 4 Variable			
No. of test case instrcts	95104366	11849715	8:1
Duration (sec.)	36	22	39%
12 Common, 8 Variable			
No. of test case instrcts	163223088	22415718	7.3:1
Duration (sec.)	46	35	24%
18 Common, 12 Variable			
No. of test case instrcts	231362556	33049336	7.0:1
Duration (sec.)	63	48	24%
10 Common, 0 Variable			
No. of test case instrcts	89531110	8351763	11:1
Duration (sec.)	28	19	32%
20 Common, 0 Variable			
No. of test case instrcts	153652858	15970598	9.6:1
Duration (sec.)	43	30	30%
30 Common, 0 Variable			
No. of test case instrcts	217951258	23623858	9.2:1
Duration (sec.)	57	37	35%

does increase the fewer comments there are. A minus saving indicates that the amount of reduction in executed code was not large enough to offset shared execution's overhead.

C. XStream

XStream [19] is an open source program for serializing objects to XML and back again that has 14,480 LOC. Like we did with JTopas, we converted this conventional program into an SPL by simply converting the following boolean configuration flags into feature variables. `TREE-STRUCTURE` inlines references such that the produced XML is a hierarchy, not a graph. `CLASSALIAS` and `FIELDALIAS` allow class and field names to be aliased. `OMITFIELD` omits specified fields when producing XML. `IMPLICITARRAY` omits specified container objects to reduce XML clutter. `ATTRIBUTE` places specified fields in the tag of the owner object for readability. `BOOLEANCONVERTER` allows a boolean field to be represented with custom string representation for 'true' and 'false'. With these 7 features, XStream SPL encodes 128 configurations.

Like with JTopas, we developed 3 sets of 3 test cases, with different sets testing different levels of instruction sharing and cases within a set testing different input sizes. It was easier to write our own classes and objects to serialize than to reuse existing ones. The test is structured as follows: a contiguous block of `Variable` objects are sandwiched between contiguous blocks of `Common` objects in a list that is serialized. XML of each `Variable` object is different for each configuration because each feature influences it. On the other hand, XML of each `Common` object is identical for each configuration, meaning that serialization between configurations should be shared for these objects. As Table III shows, the first 3 test cases have 0 `Common` object and therefore not much instruction sharing (a bit higher than 5:1), but shared execution is still 11% - 15% faster. For the next 3 test cases, 60% of the objects are `Common`, which increases sharing to 7:1 and higher and the speedup to at least 24%. Then with 0 `Variable` objects, sharing increases to around 10:1 and speedup to as high as 35%.

VI. DISCUSSION

A. Testing for Correctness

Shared execution optimizes but is otherwise semantically equivalent to conventional execution. To test that our tool implements shared execution correctly, we check that every shared execution's output is identical to conventional execution's output. For the 3 subjects, we produced a console output for each configuration, each of which was identical to the corresponding output of conventional execution.

B. Native Code

Java VMs call native methods, which are blackbox to the VM. To handle shared execution, native code execution can be changed to understand it or it can be treated as an atomic operation. For our implementation, we chose the latter and we ensure that splitting and merging occurs before and after entering a native method, meaning that the native method never reads a variable with multiple values across configurations. The simplest, safest but also the most expensive way to achieve this would be to split on each configuration of

the current state when a native method is invoked. Instead, we manually analyzed frequently executed native methods to determine under which circumstances we need to split. For example, before entering `System.arraycopy` (native in JPF), our tool checks whether the source array arguments are multivalued and split if they are. Because there are not many native methods, manual analysis was not a significant issue.

C. Hybrid Approaches

Hybrid approaches would exploit shared execution but also allow conventional execution to minimize overhead. For example, instructions could be shared only up to the first variable load that causes a split (i.e. a feature variable load), at which point each configuration being tested would be run to completion. This would almost guarantee a time saving, although it may not be much if splitting occurs early in the test. Another possibility is to switch to conventional execution after a tester specified limit, such as time, memory size or number of instructions executed. A more elaborate possibility is to split the configuration set to test at the very beginning of the test into N configuration sets and run shared execution on each configuration set. The split would be done in a way to maximize shared execution's effectiveness for each configuration set and could be performed manually using domain knowledge or automatically using static analysis.

D. Other Benefits of Sharing Execution

Although the main idea of this paper was to exploit shared execution to save execution time, we noticed other benefits. For example, shared execution reduces the size of the execution trace for the entire product line significantly, which can make it easier to store and analyze. Also, it can be used to analyze behavioral properties related to product lines. For instance, suppose that a tester knows that a block of code must be shared by all configurations. Shared execution can be used to determine whether it is or not.

VII. Related Work

A. Testing Conventional Programs

Clustered test execution [22][12] combines test cases with common initial segments into a hierarchical structure such that tests are executed together until they differ, at which point execution splits, much like shared execution. Unlike shared execution, these techniques run until completion rather than merging and thus is not able to share instructions after splitting. Also, these techniques require comparing test cases to find commonality, whereas the commonality in SPLs exists already, provided naturally by the way a product line is structured.

`Rozzle` [17] is a JavaScript multiexecution VM for exposing environment-specific malware that, like our work, explores multiple execution paths within a single execution. However, our purpose is to optimize a given set of executions by exploiting similarity between them, while their purpose is to find bugs. Our technique preserves the given set of executions, while their technique uses a form of symbolic execution that allows infeasible and unsound executions.

B. Testing Product Lines

Sampling relies on domain knowledge to select combinations of features to test [7][6][21]. It is practical but may miss problematic interactions, which our work does not. Model checking product lines [5] [4], which builds on standard model checking techniques, is different from shared execution in that they are not able to share instructions after splitting.

In [14], we statically determined features that are irrelevant to a test (e.g. unreachable or does impact outcome) to reduce combinatorics. In [15], we inserted monitors only for feature combinations that can trigger them by constructing path conditions over the features using static analysis. Shared execution, a dynamic analysis, complements these works by providing a practical reduction in a setting where the test case must be run on every feature combination because most of the features are relevant and interact, as far as can be determined statically.

There exists works on eliminating redundaunt test cases for product lines. For example, [26] determines dataflow dependencies that must be tested for each configuration and prevents redundant dependencies from being tested. Also, [23] uses symbolic execution to determine the configurations required to achieve structural coverage and shows that the number of configurations can be far fewer than the number of all possible configurations. Our work is complementary to these works in that ours eliminates bytecode instruction redundancy still remaining between the configurations determined by theirs.

[18] proposes reusing execution traces to reduce product line testing. When running a test case for a given configuration, every use of a module (a programming construct with an interface) is recorded. If another configuration uses the same module in a way that is identical to the recorded trace, then the result is retrieved from the recorded trace without having to recompute it. Shared execution can achieve greater reuse and save more time than reusing execution traces by working at the finer grained, instruction level. Also, the two works are complementary since a technique may incorporate both shared execution and trace reuse.

VIII. Conclusion

Shared execution is a technique for efficiently testing product lines that allows each variable to have as many values as there are configurations but carries out execution using a single call stack. A variable is likely to have a number of values that is considerably smaller than the number of configurations being tested, meaning that many

instructions will be shared across multiple configurations. This notion, coupled with the idea of ownership for low memory overhead, is what makes shared execution generally faster than running each configuration from start to finish. And while shared execution's performance has room for improvement, the benefit of shared execution may not be limited to time saving. Shared execution effectively "product lines" execution, which may allow us to systematically exploit commonalities and variabilities to tackle problems that remain unsolved.

Acknowledgments: we gratefully acknowledge support for this work by NSF grants CCF 0724979 (Science of Design Project), CNS-0958231, CCF-0845628, and AFOSR grant FA9550-09-1-0351.

REFERENCES

[1] D. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[2] I. Cabral, M. B. Cohen, and G. Rothermel. Improving the testing and testability of software product lines. In *Software Product Line Conference (SPLC)*, pages 241–255, 2010.

[3] S. Chandra, E. Torlak, S. Barman, and R. Bodík. Angelic debugging. In *International Conference on Software Engineering (ICSE)*, pages 121–130, 2011.

[4] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *ICSE*, pages 321–330, 2011.

[5] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *International Conference on Software Engineering*. IEEE, 2010.

[6] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and adequacy in software product line testing. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*. ACM, 2006.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 129–139, New York, NY, USA, 2007. ACM.

[8] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 207–218, 2010.

[9] GraphStream. GraphStream: A Dynamic Graph Library. http://graphstream-project.org/.

[10] Java tokenizer and parser tools. JTopas. http://jtopas.sourceforge.net/jtopas/index.html.

[11] Jikes RVM. Jikes research virtual machine. http://jikesrvm.org/.

[12] S. A. Khalek and S. Khurshid. Efficiently running test suites using abstract undo operations. In *International Symposium on Software Reliability Engineering (ISSRE)*, 2011.

[13] C. H. P. Kim. Shared execution for efficiently testing product lines: Evaluation. {http://www.cs.utexas.edu/~chpkim/sharedexecution}.

[14] C. H. P. Kim, D. Batory, and S. Khurshid. Reducing Combinatorics in Product Line Testing. In *Aspect Oriented Software Development (AOSD)*, 2011.

[15] C. H. P. Kim, E. Bodden, D. S. Batory, and S. Khurshid. Reducing configurations to monitor in a software product line. In *Runtime Verification*, pages 285–299, 2010.

[16] C. H. P. Kim, C. Kästner, and D. S. Batory. On the modularity of feature interactions. In Y. Smaragdakis and J. G. Siek, editors, *GPCE*, pages 23–34. ACM, 2008.

[17] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Oakland*, 2012.

[18] J. J. Li, B. Geppert, F. Rößler, and D. M. Weiss. Reuse execution traces to reduce testing of product lines. In *Software Product Lines Testing Workshop (SPLiT 2007) in Software Product Line Conference (SPLC)*.

[19] Library to serialize objects to XML and back again. XStream. http://xstream.codehaus.org/.

[20] R. E. Lopez-herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proc. 2001 Conf. Generative and Component-Based Software Eng*, pages 10–24. Springer, 2001.

[21] J. McGregor. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022, CMU/SEI, Mar. 2001. Available from http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tr022.pdf.

[22] S. C. Narayanan. Clustered test execution using java pathfinder. In *Master's Thesis. Department of Electrical and Computer Engineering. University of Texas at Austin*, 2010.

[23] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *International Conference on Software Engineering*, ICSE '10, pages 445–454, 2010.

[24] RIACS/NASA Ames Research Center. Java PathFinder. http://javapathfinder.sourceforge.net/.

[25] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 69–80, 2009.

[26] V. Stricker, A. Metzger, and K. Pohl. Avoiding redundant testing in application engineering. In *Software Product Line Conference (SPLC)*, pages 226–240, 2010.