Exploring the Impact of Context Sensitivity on Blended Analysis

Marc Fisher II
Virginia Tech
Blacksburg, VA, USA
fisherii@cs.vt.edu

Bruno Dufour
University of Montreal
Montreal, QC, Canada
dufour@iro.umontreal.ca

Shrutarshi Basu
Lafayette College
Easton, PA, USA
basus@lafayette.edu

Barbara G. Ryder
Virginia Tech
Blacksburg, VA, USA
ryder@cs.vt.edu

***Abstract*—**
This paper explores the use of context sensitivity both intra- and interprocedurally in a blended (static/dynamic) program analysis for performance diagnosis of framework-intensive Web-based applications. Empirical experiments with an existing blended analysis algorithm [9] compare combinations of (i) use of a context-insensitive call graph with a context-sensitive calling context tree, and (ii) use (or not) of context-sensitive code pruning within methods. These experiments demonstrate achievable gains in scalability and performance in terms of several metrics designed for blended escape analysis, and report results in terms of object instances created, to allow more realistic conclusions from the data than were possible previously.

I. Introduction

Recently, many researchers have been investigating different combinations of static and dynamic program analyses, to create more effective approaches to testing, performance diagnosis, and program understanding [3], [9], [11], [23]. There are many choices of how to couple these analysis paradigms in order to achieve scalabilty with acceptable precision on programs which cannot currently be analyzed statically. For example, the mutability analysis in [3] is a loosely coupled combination of paired static and dynamic analyses on the same data with feedback between them provided by using a UNIX pipe model of interaction between analysis application passes [3]. In contrast, blended analysis [8], [9] features strong coupling between a dynamic trace-gathering pass that provides an interprocedural calling structure representation, and a subsequent static analysis that uses the representation.

There are algorithm design choices that affect the practicality and/or precision of the resulting static/dynamic analysis [22]. For example, a lightweight dynamic analysis may overly restrict the amount of runtime information available about the specific execution, while ensuring the practical time demands of the technique. Alternatively, a higher-overhead dynamic analysis may enable specialization of the static technique using better runtime information. For example, in blended analysis knowledge of object instance creations and executed method calls can be used to prune code intraprocedurally that is known to be unexecuted on a particular run.

This paper explores the effects of two design choices — calling structure representation and (intraprocedural) code pruning — in a tightly-coupled static/dynamic analysis (i.e., blended analysis [9]). Specifically, we are interested in how algorithm practicality and precision are affected by (i) the use of a context-sensitive calling structure, a calling context tree [1], in comparison to a context-insensitive call graph, (ii) the use (or not) of control-flow-graph code pruning enabled by dynamic information, and (iii) the effect of combinations of choices from (i) and (ii). Note: we can consider control-flow-graph pruning as an application of context sensitivity, as the code is being pruned using knowledge about a particular invocation of the method. The comparison is accomplished through empirical investigation of these choices using four realistic, framework-based Web applications.

In our previous work [8], [9], we explored the use of blended escape analysis with control-flow-graph pruning. Experiments with the same benchmarks presented here measured the effectiveness of the pruning in terms of space and time savings as well as some aggregate precision effects. Many of the metrics included in this study (e.g., size of temporary data structures, number of types of constituent objects and their capture depths) were used in this previous work. This paper differs from our previous work both in focus (i.e., exploring the effects of context sensitivity in blended analysis) and content, findings from empirical comparisons of combinations of context sensitivity (i.e., (i) and (ii) above) on blended escape analysis. Note that the metrics here present findings in terms of numbers of object instances rather than as numbers of static objects (i.e., allocation sites) as in our previous work. Reporting captured object instances is more practical in order to prioritize areas of object churn that should be examined. In addition, this paper contains detailed interpretations of the observed differences between the algorithm designs.

The contributions of this paper are:

- a study comparing the effects of using context sensitivity intra- and interprocedurally, on the precision and scalability of a blended escape analysis. We believe the study results can be generalized to inform static/dynamic analysis design, particularly for using context sensitivity derived from dynamic information.
- interpretations of findings obtained using a set of metrics for measuring the precision and scalability of a blended (escape) analysis, that illustrate (i) scalability gains possible and (ii) the complexity of assessing analysis results, particularly in terms of sources of observed imprecision.

II. Background

Framework-intensive applications present new challenges for program analysis. To address some of these challenges, we proposed *blended analysis* [8], a new technique that combines static and dynamic analyses and is summarized here.

A. Blended Analysis

Blended analysis is a new analysis paradigm that aims to achieve precision comparable to that of a fully dynamic analysis, but at a practical runtime cost. This is accomplished by narrowing the focus of a static analysis to a set of executions of interest. Blended analysis first collects a lightweight profile of the application for the executions being considered. A dynamic calling structure (e.g., call graph) is then generated from the profile and used to trigger a static analysis for the region(s) of interest. This ensures that only executed methods are analyzed. A blended analysis therefore potentially analyzes a much smaller portion of the code than a whole-program static analysis, and thus generally uses less resources as well.

Because blended analysis only considers code that was executed at runtime, its results are safe only for the specific runs that were observed. In many cases, however, this limited safety is more useful than safety over all possible executions offered by most traditional static analyses. When trying to diagnose a performance problem, for example, it is common for some inputs to lead to poor performance while others make the system behave normally. Analysis results that focus on the problematic inputs thus are better for the task at hand, generally more concise and easier to understand.

B. Call Graphs and Calling Context Trees

Interprocedural static analysis typically explores a program by following calls between methods, either forward (i.e., from caller to callee) or backward (i.e., from callee to caller). This requires knowledge about the possible target methods at each call site. The vast majority of static analyses obtain this information by building a *call graph* (CG) from the analyzed code. A call graph is a directed graph in which nodes represent methods in a program, and edges represent calls between methods. In its most basic form, a call graph comprises a single node for each method in the program. For more complex call graphs, multiple nodes can correspond to the same method. Such nodes are said to represent that method in different *contexts*. Examples of common contexts include the caller of a method [13] or its associated receiver object [18]. Because of its importance to static analysis, call graph building is a heavily studied topic. Many call graph building algorithms have been defined, each with particular tradeoffs between cost and precision (e.g., [5], [14], [15], [27]).

In blended analysis, the call graph is computed from a finite set of concrete executions. Conceptually, a full dynamic trace can be viewed as a *call tree*, a call graph representation that uses a distinct node for each dynamic invocation of a method. Call trees are generally very large, even for relatively short program runs. More concise representations of the calls can easily be obtained by aggregating nodes in the call tree. For example, a basic (context-insensitive) call graph can be derived from a call tree by merging all nodes that represent invocations of the same method. While this approach minimizes the size of the resulting call graphs, it discards a lot of valuable context information. More precise aggregation schemes are possible. In this paper, we focus on *Calling Context Trees (CCTs)* [1], a popular way to represent dynamic calls. CCTs offer a much more precise representation of the calls in a program than a basic call graph, but without the prohibitively high cost of full call trees. The key idea behind CCTs is to differentiate between calls of a method based on the full invocation stack that resulted in the call. While techniques exist to collect CCTs directly at runtime (e.g., [31]), it is easy to generate them from an existing call tree by aggregating nodes that share the same method sequence from the beginning of the trace. Note that although they are called trees, CCTs contain cycles in the presence of recursion.

C. Escape Analysis

Initial experiments with blended analysis focused on object churn, a common performance problem in framework-intensive applications caused by the excessive creation of temporary object instances [9]. Temporary objects are costly, not only because of memory allocation and garbage collection costs, but mainly due to the amount of work performed to initialize them. In some extreme cases, object churn has been known to dominate execution time. Our first objective therefore was to automatically identify program regions that are responsible for significant object churn, a process which is typically accomplished manually. Identifying temporary objects instances through analysis requires an approximation of object lifetime. For this purpose, we used an existing escape analysis algorithm by Choi *et. al.* [4].

Escape analysis computes bounds on the dynamic scope of static objects. It was first proposed to enable compiler optimizations such as stack allocation of object instances, which reduces heap fragmentation and garbage collection overhead by allocating objects on the run-time stack rather than the heap, and synchronization removal, a technique that avoids costly synchronization operations when code can be shown to be thread-safe. The former requires information about objects that escape a particular method invocation; the latter necessitates knowing which objects escape their allocating thread. Escape analysis is similar to reference (i.e., points-to) analysis in that it computes the set of static objects that may be pointed to by each reference in the analyzed program region(s). Escape analysis additionally associates an *escape state* with each static object. The escape state of a static object at a method m indicates if the object is only reachable during executions of m (*captured*), or if it escapes m through a global reference (*globally escaping*) or through parameters and return values (*argument escaping*). During the analysis, a given static object can have different escape states in different methods along a call path in the program; however, all objects eventually either globally escape or become captured. We refer to the final escape state of a static object as its *disposition*. Objects

whose disposition is *captured* are of particular interest for identifying temporaries, as they can be viewed as temporary to the program region rooted in the method that captures them.

D. Blended Escape Analysis

Our blended escape analysis consists of three phases. In the first phase we use an existing profiling tool (see Section III-A2) to collect execution traces and to build a representation of the calling structure of the execution. The profiler collects a call tree and information about the object allocation for a particular execution of the system. We then aggregate the information in the call tree to construct a CG or CCT with additional information about the dynamic instances allocated within each context.

The second phase of our analysis is a static escape analysis. The output of this analysis is a connection graph for each node of the calling structure (CG or CCT). A connection graph encodes the points-to information between the variables, static objects, and fields visible from within the context to which it corresponds. The escape status of each object is also included in the connection graph.

The third phase maps the dynamic instance information from a CCT to the static objects used in the escape analysis. For each calling context in the CCT, the profiler we used identifies the number of instances of each type that were allocated, although it does not record their actual allocation sites. Therefore, we cannot precisely match the allocated instances to the static objects, and in many cases we map an allocated instance to multiple static objects to ensure analysis safety. Additionally, the profiler is unable to accurately identify the type of arrays with arity greater than 1 or with non-primitive base types. These limitations add imprecision to the dynamic instance to static object mapping.[1]

After mapping the dynamic instances to the static objects, reduced connection graphs are built for each context in the CG or CCT. Reduced connection graphs are simple object-to-object graphs that include only those static objects that were mapped to dynamic instances.

III. Study

The purpose of this study is to determine the impact of different algorithm design choices with respect to context sensitivity on the performance and precision of blended escape analysis.

A. Experiment Design

1) Benchmarks: We used version 6.0.1 of the Trade benchmark running on WebSphere 6.0.0.1 and DB2 8.2.0.[2] The way in which the Trade benchmark interfaces with the WebSphere middleware can be configured through parameters. We experimented with four configurations of Trade by varying two of its parameters: the run-time mode and the access mode. The run-time mode parameter controls how the benchmark accesses its backing database: the Direct configuration uses the Java Database Connectivity (JDBC) low-level API, while in the EJB configuration database operations are performed via Enterprise Java Beans (EJBs).[3] The access mode parameter was set to either Standard or WebServices. The latter setting causes the benchmark to use the WebSphere implementation of web services (e.g., SOAP) to access transaction results. All other parameters retained their default values. Each of the three benchmarks was warmed up with 5000 steps of the built-in scenario before tracing a single transaction that retrieves a user's portfolio information from a back-end database into Java objects. Our analysis was applied to the portion of the transaction that retrieves nine holdings from a database. The warm-up phase is necessary to allow all necessary classes to be loaded and caches to be populated. Tracing the benchmark in a steady state is more representative of the behavior of real Web applications.

Because the Trade application consists of a relatively small user code that interacts with a large amount of framework and library code, the four configurations of the same application have very different properties and behavior in practice. Therefore, we use these four configurations as different benchmarks, as have other researchers [26].

2) Experimental Setup: Our blended escape analysis is built using the WALA analysis framework.[4] The IBM research prototype Jinsight [6] was used to obtain execution traces including method calls and object creations. To obtain complete call graphs and calling context trees from the trace, all experiments were performed with an IBM JVM version 1.4.2 with the JIT disabled in order prevent method inlining at runtime. Note that different JIT implementations may provide more fine-grained control over the specific optimizations performed by the JIT, and may allow inlining to be disabled without requiring the JIT to be turned off completely.[5]

The escape analysis was run on Apple Macintosh Pros, each powered by 2 Quad Core Intel Xeon processors running at 2.8GHz and 8GB of RAM. Each machine ran the 64-bit Linux kernel version 2.6.27. All analyses were performed using a 64-bit Sun JVM version 1.6.0 with a maximum heap size of 7000MB.[6]

3) Algorithm Design Choices: There are two different dimensions along which the blended escape analysis can be varied: interprocedural calling structure and intraprocedural control flow. As mentioned in Section II-B, we examined two different interprocedural calling structures: call graphs (CGs) and calling context trees (CCTs) built by our extension to Jinsight.

For intraprocedural control flow we consider two different options, unpruned and pruned control flow. Unpruned control flow for a method uses the actual bytecode of that method.

[1] We are working on a new profiling tool that will address these limitations.

[2] Trade, WebSphere and DB2 are available to academic researchers through the IBM Academic Initiative.

[3] Trade 6 uses the EJB 2 framework.

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

[5] Instrumentation-based profiling techniques generate accurate call graphs even in the presence of inlining.

[6] Except for `direct-ws` with pruning enabled and calling context trees where the maximum heap size was 7500MB.

TABLE I
CALLING STRUCTURE STATS

	CG		CCT	
	Nodes	Edges	Nodes	Edges
direct-std	710	1,116	1,473	1,473
direct-ws	3,308	6,361	18,267	18,363
ejb-std	1,978	3,454	8,089	8,113
ejb-ws	4,480	8,597	25,012	25,135

TABLE II
RUNNING TIMES

	CG		CCT	
	unpruned	pruned	unpruned	pruned
direct-std	4.8s	5.4s	7.8s	4.4s
direct-ws	40m 54.4s	3m 59.4s	2m 3.3s	*1m 2.9s
ejb-std	27.8s	13.4s	40.5s	18.1s
ejb-ws	N/A	4h 21m 8.5s	N/A	15m 50.0s

Recall that to prune control flow, we use the information collected by Jinsight about method calls and object allocations to remove paths from the control flow graphs that we can prove did not execute on the run of interest. When the interprocedural calling structure is a CCT, we prune the control flow of each different context independently.

B. Scalability Findings

The first question we need to answer is whether or not our analysis will scale to the potentially larger CCT calling structure. When we previously analyzed these benchmarks using CGs with and without pruning [7], we were able to analyze all with pruning enabled, and all but one with pruning disabled. However, as shown in Table I, the CCTs for these benchmarks are significantly larger than the CGs, with between 2 and 5.6 times as many nodes and 1.3 and 2.9 times as many edges.

Table II presents the running times for each of the runs. Our analysis was able to complete using CCTs in all the cases that completed using CGs. Furthermore, on all runs that took longer than a minute to complete when using a CG, the corresponding CCT run was faster, sometimes significantly faster.

As mentioned earlier, the pruned CCT run for direct-ws used a larger maximum heap size setting than the other runs. In general, the ability of the algorithm to complete is sensitive to the order in which nodes in the calling structure are processed. This is due to the fact that our escape analysis is not a strictly monotonic dataflow analysis. Due to the use of a hash-based data structure for storing and iterating through calling structure nodes, an easy method to permute the order in which the nodes are processed is to change the maximum heap size, thereby changing the allocation address for some of the objects. In the case of direct-ws when the heap size was set to 7000MB, the analysis would get stuck in an infinite cycle. By changing it to 7500MB, we were able to perturb the algorithm sufficiently to allow it to complete on this benchmark. Because of this change in heap size, the timings for the pruned CCT run of direct-ws are not directly comparable to the other timings in the table, but are consistent with them.

We looked more deeply at the structure of the CCTs and CGs that were being used and developed several hypotheses about the reasons for the improved performance. We knew from observation that the analysis spends significant amounts of time iterating around strongly connected components (SCCs) in the calling structure. Since CCTs are more tree-like than CGs,[7] it seemed likely that there would be fewer non-trivial SCCs[8] in the CCTs than in the CGs. However, this turned out not to be true, and, in fact, due to duplication of SCCs observed in the CG on different calling paths in the CCTs, there were more non-trivial SCCs in the CCTs than in the CGs.

Our second hypothesis was that the savings were achieved through a decrease in the in-degree of nodes in the calling structure. Since our analysis propagates information backwards through the calling structure, that information needs to be replicated along each incoming edge of a node. On direct-ws we found that the CCT had only one node with in-degree greater than 1 (out of 1473 nodes total), while the CG had 156 nodes with in-degree greater than 1 (out 710 nodes total), including nodes with in-degree as high as 38. We found similar results on the other three benchmarks. These results support our hypothesis that lower in-degree is a likely reason for the improved performance of the CCTs.

In addition to looking at the running times, we also looked at the pruning differences between the CGs and CCTs. Although our pruning algorithm removes basic blocks from the control flow graphs for methods, we also include information on the statements removed to give some indication of the size of the pruned basic blocks. Table III shows the total number of statements and basic blocks, the number of statements and basic blocks remaining after pruning, and the percent removed by pruning for each of our benchmarks with both CGs and CCTs. Since the CCT may contain several nodes (i.e., contexts) for each method node in the CG, with more specific information on these nodes about the allocated objects and called methods, we can guarantee that the control flow graph for a CCT node will have at least as many basic blocks pruned as the corresponding CG node. This led us to expect that the percentage of pruned statements and basic blocks would increase when using a CCT; however, our data contradict that expectation. In fact, we found that the percentage of pruned basic blocks (with respect to the total number of basic blocks in the calling structure) decreased when using the CCT. This is likely because the methods that are replicated most frequently within the CCT are small methods (e.g., *getter* and *setter* methods are likely to be called often from a variety of contexts) where pruning is generally less effective. The increase in the number of instances of these methods then would account for the decrease in the effectiveness of pruning.

[7] As mentioned in Section II-B, in spite of being called trees, CCTs are not in fact trees and can contain cycles.

[8] An SCC with more than one node

TABLE III
PRUNING STATISTICS

		CG			CCT		
		Total	Remaining	% Savings	Total	Remaining	% Savings
direct-std	basic blocks	16,450	9,118	44.6%	25,170	15,710	37.6%
	statements	17,864	9,268	48.1%	26,803	15,640	41.6%
direct-ws	basic blocks	65,692	39,453	39.9%	238,114	163,932	31.2%
	statements	69,220	38,651	44.2%	234,200	150,593	35.7%
ejb-std	basic blocks	43,992	23,456	46.7%	118,362	69,599	41.2%
	statements	47,388	23,138	51.2%	124,122	67,207	45.9%
ejb-ws	basic blocks	93,285	53,666	42.5%	350,178	229,798	34.4%
	statements	98,666	52,338	47.0%	352,649	215,347	38.9%

TABLE IV
DISPOSITION IMPROVEMENT

(a) unpruned CG vs. pruned CG

unpruned	pruned	direct-std	direct-ws	ejb-std
unchanged		186	5258	1743
captured	escaped	-	14	1
indeterminate	escaped	-	97	-
escaped	captured	-	153	2
escaped	indeterminate	-	-	5

(b) unpruned CCT vs. pruned CCT

unpruned	pruned	direct-std	direct-ws	ejb-std
unchanged		186	5418	1749
captured	escaped	-	25	2
indeterminate	escaped	-	51	-
escaped	captured	-	28	-

(c) unpruned CG vs. unpruned CCT

CG	CCT	direct-std	direct-ws	ejb-std
unchanged		185	4791	1590
indeterminate	captured	-	46	-
indeterminate	escaped	-	7	-
escaped	captured	1	676	66
escaped	indeterminate	-	2	95

(d) pruned CG vs. pruned CCT

CG	CCT	direct-std	direct-ws	ejb-std	ejb-ws
unchanged		185	4919	1598	6351
captured	escaped	-	4	-	4
indeterminate	escaped	-	7	-	7
escaped	captured	1	590	63	722
escaped	indeterminate	-	2	90	4

C. Precision Findings

1) Disposition: The simplest measure of precision improvement is a comparison of the disposition of the dynamic instances. Although our static analysis calculates a unique disposition of captured or escaping for each static object, due to imprecision in mapping those static objects to their corresponding dynamic instances, it is possible that sometimes a dynamic instance can map to two different static objects, one that is escaping and one that is captured. Therefore, the disposition for a given instance, (i.e., the final escape state of that instance), may be: captured, escaping, or indeterminate.

Table IV shows the change in disposition of instances on our benchmarks for all 4 combinations of CG versus CCT with pruned or unpruned code. This allows us to to examine the effects both of pruning and of CGs versus CCTs independently. Table IV(a) reaffirms our prior results that pruning has limited effect on the precision of the analysis when using call graphs. [9]

[9]We observe an anomaly in the results from Table IV(a) and Table IV(b) where some dynamic instances fall in the captured → escaped category. This data asserts that some dynamic instances were found to be captured by the analysis without pruning on CG, but as escaped by the analysis with pruning. This loss of precision should not occur; on inspection it is explained by an infrequent 'corner case' bug we found in the mapping of parameters from callees to callers when some calls are pruned. (We are working on fixing this bug for the final paper.)

When we compare unpruned versus pruned control flow graphs for CCTs in Table IV(b), we obtain similar results to the comparison on CGs. However, when we begin to compare the CCTs to CGs we see bigger differences than observable from pruning alone. When comparing the use of CCTs to CGs with pruning disabled (Table IV(c)), we find that on `direct-ws`, 676 instances ($> 12\%$ of all of the dynamic instances) that were classified as escaped with the CG were classified as captured with the CCT. Similarly, looking at use of the pruned CCTs versus CGs, a large number of instances for both `direct-ws` and `ejb-ws` that were classified as escaped with a CG are classified as captured with a CCT. (Table IV(d)). By identifying additional static objects (and thus their instances) as captured, our analysis can help developers identify additional areas of object churn that can be examined. Therefore, intra- and interprocedural context sensitivity here leads to improvements in analysis results having ramifications in practice.

2) Capture depth: A second metric we examined when comparing precision between different algorithm design choices is the change in the capture depth of instances. The *capture depth* of an instance is the distance between the node in the calling structure where the instance was allocated and the node where the instance was captured. Instances with a large capture depth are more difficult for developers to identify

TABLE V
INTERESTING METHODS

	Captured Count				Percentile			
	CG		CCT		CG		CCT	
Method Name	unpruned	pruned	unpruned	pruned	unpruned	pruned	unpruned	pruned
direct-ws								
SAXParser.parse	0	0	216	216	0	0	97	98
P2DConverter.flush	91	0	316	180	95	0	98	97
ChannelTargetImpl.getCFEndPoint	8	103	104	103	64	95	94	94
SOAPElement.addTextNode	0	45	0	45	0	88	0	86
HTTPSender.invoke	7	7	8	20	63	63	63	81
ClusterServiceImpl.matchEndPoints	20	13	20	13	83	79	82	76
ejb-std								
GeneratedMethodAccessor38.invoke	8	14	122	122	50	65	92	92
GeneratedMethodAccessor32.invoke	7	12	102	102	45	63	90	90
ejb-ws								
SAXParser.parse	N/A	0	N/A	216	N/A	0	N/A	98
P2DConverter.flush	N/A	0	N/A	180	N/A	0	N/A	97
ChannelTargetImpl.getCFEndPoint	N/A	7	N/A	103	N/A	60	N/A	95
OneRowResultCollectionImpl.processOneRowToCacheEntry	N/A	9	N/A	27	N/A	64	N/A	82

as object churn because they involve tracing manually through long call chains in the code. Our previous work showed that the capture depth within these programs can be quite large, in some cases as many as 14 calls. By using a more precise algorithm to compute the capturing locations for instances, we expect to provide better information to developers for identifying object churn.

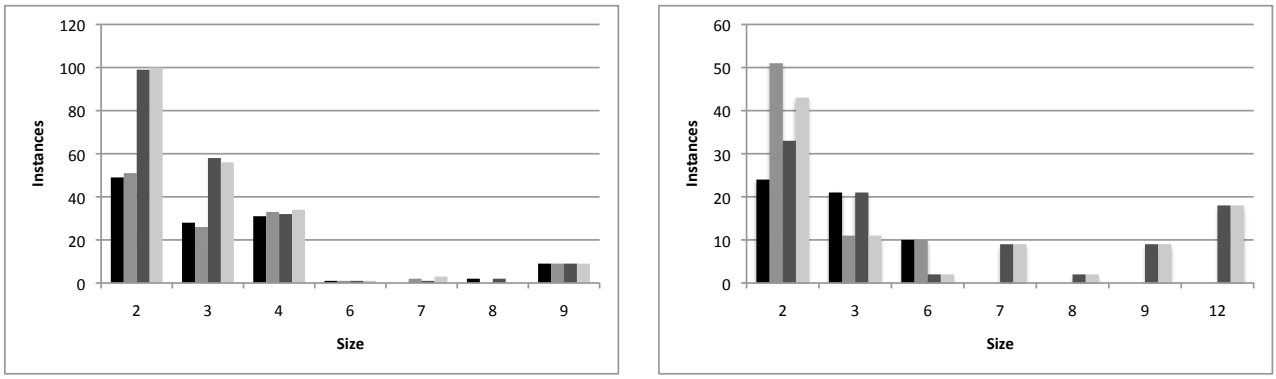
However, our results show that the capture depth of the vast majority of dynamic instances did not change either when transitioning from unpruned control flow to pruned control flow or when transitioning from CGs to CCTs. Specifically, we saw no change in the capture depth for any dynamic instances on `direct-std` or `ejb-std`. On `direct-ws` the captured depth of 47 dynamic instances (of 5,522 total dynamic instances) changed when transitioning from the unpruned CCT algorithm to the pruned CCT algorithm. On the other comparisons of `direct-ws` and `ejb-ws` the capture depth of only one to three dynamic instances changed. These results show that the algorithm choice had nearly no effect on the capture depth.

3) Concentration: Since the goal of this analysis is to identify locations within the application where object churn can be removed, we also wanted to look at how algorithm design choices differed in identifying the methods where large numbers of temporaries are being captured. Previously, we focused on a metric called *concentration* to identify the percentage of captured instances that were captured by the top x% of capturing methods. This work showed that the top 20% of the capturing methods accounted for the majority of the captured instances. When applied to the CCT data, we found similar results; however, the set of methods that appeared in the top 20% shifted.

Table V shows the methods that had interesting shifts with respect to (i) the number of captured instances and (ii) the method rank within the list of capturing methods. The *captured count* columns show the number of instances identified as captured by the method, when using the specified design choices. The *percentile* columns show what percentage of all of the capturing methods captured *fewer* instances than the method in the table. For example, on `direct-ws`, the pruned CCT run identified 216 instances as being captured in `SAXParser.parse` and found that `SAXParser.parse` captured more dynamic instances than 98% of the identified capturing methods for that run. We identified a method as *interesting* for purposes of this table if it captured different numbers of dynamic instances for different design choices and if the percentile crossed the 80%, 90% or 95% boundary for some pair of choices.

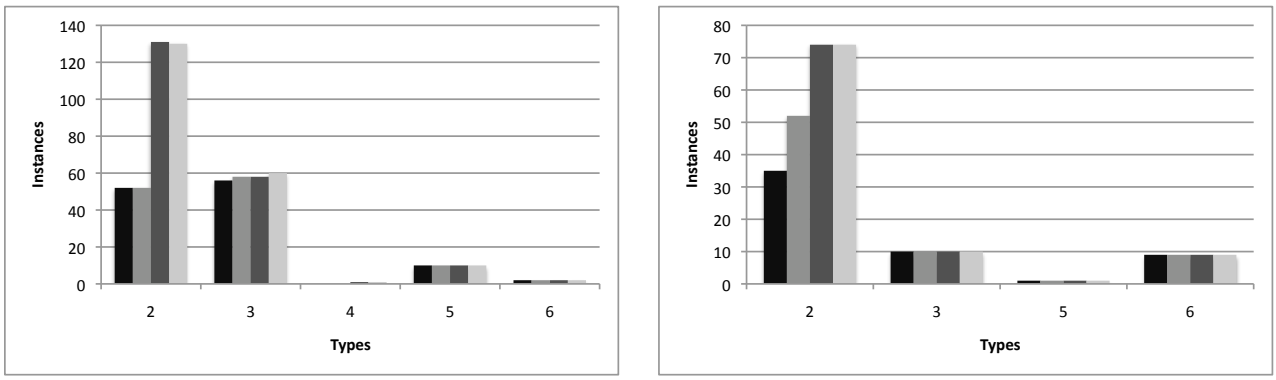
As can be seen in Table V, changes to both the inter- and intraprocedural context sensitivity have significant effects on the relative ranking of capturing methods for three of the four benchmarks (on `direct-std` there were no methods that met our criteria). A more precise algorithm can both indicate new methods that the user should consider (e.g., `SAXParser.parse` on both the `ws`runs) as well as those that should be avoided because they may be less fruitful to consider (e.g., `ClusterServiceImple.matchEndPoints` on `direct-ws`).

`P2DConverter.flush` on `direct-ws` is an interesting case. Adding intraprocedural pruning allows the algorithm to identify fewer captured instances, but adding interprocedural context sensitivity results in the identification of more captured instances. The reduction in captured instances from pruning is caused primarily by 91 `StringBuffer` instances. The `flush` method includes 6 different `StringBuffer` allocation sites. Due to our mapping of dynamic instances to static objects, the 91 `StringBuffer` instances must be mapped to each of these allocation sites. Of these 6 allocation sites, the instances allocated at 5 of them are shown to be captured by the escape analysis. However, after pruning, all 5 of the captured sites are removed (the removed allocations all relate to debugging output that is disabled), leaving only the escaping allocation site. This results in some instances no longer matching captured static objects, decreasing the number of captured instances, and therefore, the rank of the `flush` method in the list of capturing methods.



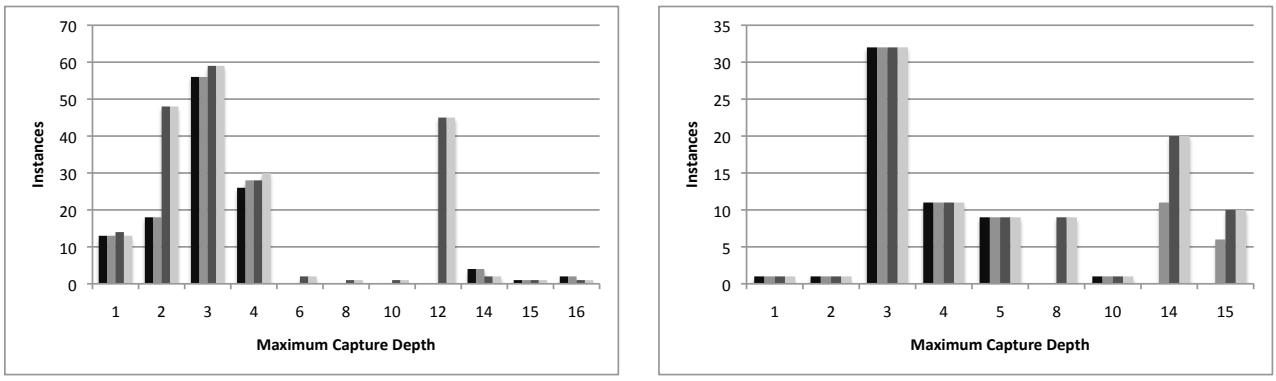
(a) `direct-ws` (b) `ejb-std`

Fig. 1. Size of Data Structures



(a) `direct-ws` (b) `ejb-std`

Fig. 2. Number of Types in Data Structures



(a) `direct-ws` (b) `ejb-std`

Fig. 3. Capture Depths of Data Structures

The increase in captured instances when moving from a CG to a CCT is primarily due to the reclassification of a data structure rooted at a `SOAPBuilder` instance from escaping to captured. `SOAPBuilder` objects are allocated in a method, `SOAPBuilder.onStartChild`, that is called directly from `flush` and transitively though a call to `DEventProcessor.onSimpleChild` from `flush`. The `SOAPBuilder` objects that are returned from the direct call eventually are able to escape from `flush` (via an assignment to a field), while the objects that are returned transitively through `onSimpleChild` do not escape from the `flush` method. When using a CG, the analysis is not able to distinguish between the objects from the separate calls to `onStartChild` and therefore must indicate that all of the`SOAPBuilder` objects that are returned to `flush` could potentially escape from `flush`, while the CCT allows these two kinds of `SOAPBuilder` objects to be disambiguated.

4) Data structures: We hypothesized that when trying to fix object churn problems, it might be useful to focus on areas where large data structures are captured. We define a captured data structure as a captured root object and all captured objects that are reachable from that root. Note that a root can be a single captured object or a strongly connected component of captured objects with no incoming reference edges in the reduced connection graph. For a data structure with a single static object root, we use the number of dynamic instances that mapped to that static object as the number of instances of that data structure. If a data structure has an SCC consisting of multiple static objects as its root, we identify the static object in that SCC with the fewest dynamic instances mapped to it and use that as the number of instances of that data structure. We only include data structures with at least two static objects in our investigations. We only talk about data structures in `direct-ws` and `ejb-std` here, because the data structures in `direct-std` were relatively uninteresting and those in `ejb-ws` were similar to the results below.

Figure 1 shows how many instances of each size of data structure were captured. Since we cannot know the actual number of instances within each data structure at runtime, we use the number of static objects within the captured data structure as an approximation of the size of the data structure. On `direct-ws` we see that pruning has minimal effect on the size of the captured data structures, but that using a CCT significantly increased the number of captured small data structures without significantly changing the number of larger captured data structures. This indicates that the additional data instances identified as captured when using a CCT were not part of already captured data structures.

On `ejb-std` we see that the use of pruning significantly impacts the number of captured data structures of size 2 and 3, increasing the number of size 2 data structures and decreasing the number of size 3 data structures. This is most likely due to the pruning of allocation sites that were identified as captured in the unpruned analysis. Additionally on `ejb-std` we see significantly more large captured data structures and fewer small captured data structures when moving from a CG to CCT. This suggests that unlike the case with `direct-ws`, the additional captured objects were part of existing data structures, thereby increasing the size of many data structures.

We also looked at the number of types within the data structures. Figure 2 shows the number of data structure instances that had a particular number of types. For `direct-ws` the distribution of number of types within the data structure is similar to the distribution of the sizes of the data structures. However, on `ejb-std`, we see that much of the variation we saw in the size of data structures was removed when we just consider the number of types within the data structures. In particular there is almost no difference between unpruned and pruned for the CCT runs, and the differences between unpruned and pruned in the CG runs were significantly reduced. This supports our hypothesis that the variance in size was a result of pruning of allocation sites within the control flow graphs. Additionally, the difference in number of types between the CG and CCT runs mostly disappears. This indicates that the additional static objects that were included in the data structures were of the same types as objects already in the data structures.

For each data structure we compute a maximum capture depth as the largest distance within the calling structure between an allocation site in the data structure and the capturing method for that data structure. Figure 3 shows the number of data structures with each maximum capture depth. In general the use of pruning had only small effects on the capture depth of data structures. The use of a CCT tends increase the number of data structures captured at several depths. When coupled with the small changes in capture depths found for instances, these increases probably mostly correspond to instances that were identified as captured with CCTs but as escaping with CGs.

IV. Related Work

A number of previous studies have investigated the impact of context sensitivity on the results of static analyses. Liang *et. al.* [16] performed a set of empirical studies to evaluate the effect of context sensitivity on the precision of Andersen's points-to analysis. They compared call string and object sensitive variants of the analysis to data obtained by dynamic analysis, and show that context sensitivity can lead to significant precision improvements. Lhoták and Hendren [15] performed a similar study of the impact of context sensitivity using a BDD-based implementation. They analyzed a large number of benchmarks to determine the impact of context sensitivity on the characteristics of the points-to sets as well as on the precision of client analyses such as devirtualisation of calls and cast removal. Their results show that context sensitivity almost always achieves better precision than a context-insensitive analysis (and is never worse), with object sensitivity surpassing other types of context sensitivity in all cases. Liang *et. al.* [17] performed a complementary study of the impact of various object abstractions on the precision of analyses for concurrency. Their object abstractions use call stack, object recency, and heap connectivity information.

Similarly to previous studies, their results show that context sensitivity is instrumental in achieving high precision. They observe that the effectiveness of a given heap representation is client-dependent, and correlates heavily with properties of the analysis under consideration. While we share the goal of evaluating the impact of context sensitivity with these previous studies, our work differs on two main points: (i) we use dynamic call information rather than statically derived call graphs and (ii) we focus our study on specific blended analyses, in particular blended escape analysis.

More generally, our work aims to effectively characterize the behavior of framework-intensive applications. Several previous analyses have also used dynamic analysis to understand the behavior of these large systems, for example to diagnose performance problems or to understand the data structures used. Ammons *et. al.* [2] used execution profiles to find performance bottlenecks by identifying expensive call sequences. Srinivas *et. al.* [26] designed a dynamic analysis technique that identifies method invocations that account for a specified cumulative percentage of execution cost in large, commercial Java applications. Mitchell *et. al.* have designed tools to identify memory leaks in long-running applications [19] and to identify key data structures in a heap snapshot [20]. Zhao *et. al.* [30] have studied the impact of excessive object allocations on the scalability of a number of Java benchmarks to a high number of CPU cores. Specifically, they were concerned with the bus write traffic caused by the garbage collected memory management strategy. Recent work by Shankar *et. al.* [24] addresses the identification and removal of temporary objects in the context of a production just-in-time (JIT) compiler. They use sampling profiles of object lifetimes to identify program regions, *churn scopes*, that encapsulate the lifetime of many objects. Xu *et. al.* [29] have designed a new runtime technique that identifies bloat in framework-intensive applications caused by excessive copying of data between objects. Performance defects were found and fixed in a number of benchmarks with great success, with the resulting runtime improvements reaching up to 30% for some benchmarks. Tripp *et. al.* [28] have devised a static taint analysis algorithm for Java that specifically targets web applications. Their analysis uses a variety of models that are tailored to the applications under study, e.g., for reflective calls, Enterprise Java Beans (EJBs), Java Server Pages (JSPs), and other popular frameworks.

In an early paper, Ernst [10] discussed various ways in which both static and dynamic techniques can complement each other. Our work uses a combination of static and dynamic analysis techniques where the dynamic information is used to augment a static analysis. Other uses of dynamic analysis to improve a static analysis have been used to solve a wide variety of well-known problems, including program slicing (e.g., [12]), change-impact analysis (e.g., [21]) and symbolic analysis (e.g., [11]). Artzi *et. al.* [3] described various pipelined combinations of static and dynamic mutability analyses for Java method parameters, and showed that it can exceed the accuracy of a more complex static analysis. In contrast, our approach is more tightly coupled than the pipelined approach, in which results from analysis are only used as input to the next one. Sinha *et. al.* [25] described a technique for fault localization and repair that uses stack traces from faulty concrete executions to guide a static dataflow analysis. The dynamic information here serves a similar purpose as with our use of blended analysis, but it defines the program region to be analyzed more loosely since the faulty statements do not necessarily belong to methods that appear in the provided stack trace.

V. Conclusion

Blended analysis is a powerful technique for scaling existing static analyses to large programs such as framework-based Web applications, for problems solvable on a set of interesting executions. We have used blended escape analysis for performance diagnosis to identify areas of object churn.

This paper has presented the first detailed study of the impact of various algorithm design decisions on the scalability and precision of a blended analysis. We have explored the use of context sensitivity with respect to calling structure representation (i.e., CGs vs. CCTs) and control-flow pruning within a method. Control-flow pruning was enabled using knowledge of unexecuted calls and object allocations on a particular run.

The most surprising finding of our study was that the use of CCTs on complex Web applications is both possible and practical, although CCTs clearly are larger than CGs in number of nodes and statements. The execution times reported on CCTs were less than 15 minutes, even for our most realistic (and complicated) benchmark, `ejb-ws`. The significant analysis speedups demonstrated on the CCTs occur because paths in a CCT are less ambiguous than those in a CG, resulting in faster propagation of a smaller, more accurate dataflow solution. Important timing savings were realized by using pruning on CGs (shown in our previous work) and on CCTs, although there was little change in precision due to pruning in either case. Thus, precision gains mostly were due to the context-sensitive calling structure representation. For example, significant differences in the number of static objects in captured data structures were observed. By focusing our metrics more on the dynamic instances rather than the static objects used in the analysis, we also were able to identify improvements to precision more important to a developer. These improvements can direct a developer more effectively to the key capturing methods for temporary objects, thus aiding in focusing attention on the right parts of the program to examine.

Therefore, this study demonstrates the significant impact of context sensitivity on both algorithm scalability and precision, both of which affect the practicality of blended analysis on real programs. Moreover, the non-intuitive result that a larger, context-sensitive calling structure representation is scalable, and results in a faster, more precise analysis is significant in highlighting a key difference between whole-program static analysis and blended analysis.

REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997, pp. 85–96.

[2] G. Ammons, J.-D. Choi, M. Gupta, and N. Swamy, "Finding and removing performance bottlenecks in large systems," in *Proceedings of the European Conference on Object-Oriented Programming*, 2004.

[3] S. Artzi, M. D. Ernst, D. Glasser, and A. Kiezun, "Combined static and dynamic mutability analysis," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 104–113.

[4] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff, "Stack allocation and synchronization optimizations for Java using escape analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 6, pp. 876–910, 2003.

[5] D. David Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 6, pp. 685–746, 2001.

[6] W. DePauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the execution of Java programs," in *Software Visualization*, ser. LNCS, 2002, vol. 2269, pp. 151–162.

[7] B. Dufour, "Practical analysis of framework-based applications," Ph.D., Rutgers, The State University of New Jersey, December 2009.

[8] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analalysis*, 2007, pp. 118–128.

[9] ——, "A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2008.

[10] M. Ernst, "Static and dynamic analysis: Synergy and duality," in *Proceedings of the International Workshop on Dynamic Analysis*, 2003.

[11] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[12] A. Groce and R. Joshi, "Exploiting traces in program analysis," in *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2006.

[13] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 1997, pp. 108–124.

[14] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," in *Proceedings of the International Conference on Compiler Construction*, ser. LNCS, vol. 2622, April 2003, pp. 153–169.

[15] ——, "Context-sensitive points-to analysis: is it worth it?" in *Proceedings of the International Conference on Compiler Construction*, ser. LNCS, vol. 3923, March 2006, pp. 47–64.

[16] D. Liang, M. Pennings, and M. J. Harrold, "Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs," in *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, 2005, pp. 6–12.

[17] P. Liang, O. Tripp, M. Naik, and M. Sagiv, "A dynamic evaluation of the precision of static heap abstractions." [Online]. Available: http://berkeley.intel-research.net/mnaik/pubs/

[18] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 1, pp. 1–41, 2005.

[19] N. Mitchell and G. Sevitsky, "LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications," in *Proceedings of the European Conference on Object-Oriented Programming*, 2003.

[20] N. Mitchell, G. Sevitsky, and H. Srinivasan, "Modeling runtime behavior in framework-based applications," in *Proceedings of the European Conference on Object-Oriented Programming*, 2006.

[21] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2003.

[22] B. G. Ryder, "Dimensions of precision in reference analysis of object-oriented programming languages," in *Proceedings of the International Conference on Compiler Construction*, ser. LNCS, vol. 2622, April 2003, pp. 126–137.

[23] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, 2005.

[24] A. Shankar, M. Arnold, and R. Bodik, "JOLT: Lightweight dynamic analysis and removal of object churn," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2008.

[25] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for Java runtime exceptions," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analalysis*, 2009, pp. 153–164.

[26] K. Srinivas and H. Srinivasan, "Summarizing application performance from a components perspective," in *Proceedings of the International Symposium on the Foundations of Software Engineering*, September 2005, pp. 136–145.

[27] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2000, pp. 281–293.

[28] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: effective taint analysis of web applications," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 87–97.

[29] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, "Go with the flow: profiling copies to find runtime bloat," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[30] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao, "Allocation wall: a limiting factor of Java applications on emerging multi-core platforms," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2009, pp. 361–376.

[31] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi, "Accurate, efficient, and adaptive calling context profiling," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 263–271.