Improving Efficiency of Dynamic Analysis with Dynamic Dependence Summaries

Vijay Krishna Palepu Guoqing Xu James A. Jones
University of California, Irvine

***Abstract*—Modern applications make heavy use of third-party libraries and components, which poses new challenges for efficient dynamic analysis. To perform such analyses, transitive dependent components at all layers of the call stack must be monitored and analyzed, and as such may be prohibitively expensive for systems with large libraries and components. As an approach to address such expenses, we record, summarize, and reuse dynamic dataflows between inputs and outputs of components, based on dynamic control and data traces. These summarized dataflows are computed at a fine-grained instruction level; the result of which, we call "dynamic dependence summaries." Although static summaries have been proposed, to the best of our knowledge, this work presents the first technique for dynamic dependence summaries. The benefits to efficiency of such summarization may be afforded with losses of accuracy. As such, we evaluate the degree of accuracy loss and the degree of efficiency gain when using dynamic dependence summaries of library methods. On five large programs from the DaCapo benchmark (for which no existing whole-program dynamic dependence analyses have been shown to scale) and 21 versions of NANOXML, the summarized dependence analysis provided 90% accuracy and a speed-up of 100% (*i.e.,* $\times 2$), on average, when compared to traditional exhaustive dynamic dependence analysis.**

I. INTRODUCTION

As the needs of society are increasingly accomplished with software systems, and those software systems become more complex and interrelated, software developers are, to an increasing extent, building components of software that interact with and build upon existing software components. Rather than writing all needed functionality from the low-level operating system to the high-level client interfaces, developers regularly use features that were developed by others, provided by components such as APIs, libraries, middleware, and infrastructures. Today's reality is a scenario that was predicted by McIlroy [11] in the 1960s.

Numerous researchers have identified some of the challenges that can be faced when depending upon and assembling existing components, often referred to as "Components, off-the-shelf" or *COTS*. One of the common challenges to reusing third-party components is that analysis tasks become increasingly expensive as the extent and depth of component reuse increases (*e.g.,* layer upon layer of transitive component reuse). To properly analyze the program, the effects of the infrastructure must be understood. As such, complete analysis should also analyze all transitively underlying components to determine how they affect the program under test.

Orso *et al.* [14] discussed some of the challenges of performing analysis in the presence of external components and proposed abstract representations, *i.e., metadata*, to provide information about component functionality. Later, Orso *et al.* [13] extended these ideas for component metadata by specifying a concrete metadata scheme to enable regression-test selection in the presence of components. Although Orso's solution for regression-test selection provides a powerful solution for that specific task, such challenges extend to many other (more heavyweight) dynamic software-analysis tasks. For example, when performing a dynamic dependence analysis, the analysis needs to trace data flows through any encountered libraries and components during the whole execution. Frequent profiling of methods in these libraries and components contributes extensively to the already-large run-time costs, making the analysis prohibitively expensive for large-scale, long-running applications. For example, our experiments show that recording a whole dependence trace for DaCapo *antlr* with even small workloads requires more than 20 hours. The analysis time can be reduced to less than 2 hours if the library methods (in *rt.jar*) are not instrumented.

An important technique to reduce the analysis costs and provide such metadata is to *summarize* the behaviors of these components. Once generated, summaries are then *applied* during future executions to improve analysis efficiency. In fact, summary-based analysis has been extensively studied in the static analysis community, and various techniques (such as [4], [17], [18], [20], [26], [28]) have been developed to summarize procedural effects to achieve both modularity and efficiency. However, static summaries provide conservative overapproximations when describing heap data effects. For example, such summaries are often imprecise in modeling heap locations and distinguishing among array elements, making it particularly difficult for a dynamic analysis to use.

In this paper, we propose to compute summaries *dynamically* to improve the efficiency of dynamic analyses and to provide precise dependence metadata. Summaries generated over representative runs of the selected library methods are abstracted and applied in the analysis of a program that invokes these methods, leading to substantially reduced analysis running time and trace size. In this paper, we use dynamic dependence analysis as an example, and show how to compute dynamic dependence summary information to enable developers to characterize and capture external effects of reused components, for modern object-oriented languages. As dependence analysis provides a basis for a variety of dynamic techniques, such as program slicing [29], bloat analysis [25], tainting-based information flow analysis [12], [27], and potential parallelism detection [6], our technique provides

a mechanism to provide increases in efficiency (in terms of time and space) with the potential tradeoff of accuracy loss. This work provides the first technique for producing dynamic summaries and evaluates the tradeoff between speed and accuracy.

To evaluate, we implemented our summary-based dynamic dependence analysis and performed two experiments. The first experiment evaluates the efficiency and cost gains by reusing external component analysis on large programs to which no existing dynamic dependence analyses have been shown to scale. The second experiment examines the analysis accuracy of using the summarized metadata versus performing exhaustive analysis through all external components. Our experiments show an accuracy of 90% and a speed up 100% (*i.e.,* $\times 2$), on average, when compared with traditional, exhaustive dynamic dependence analysis.

The main contributions of this work are:

1) Definition of the technique for producing summarized analysis results for software components, which can enable improved efficiency of subsequent analysis tasks.
2) Designation of the process by which developers can capture and encode component analysis results, in the form of dynamic dependence summaries, and then reuse those summaries for future analysis tasks.
3) Evaluations of the impact of utilizing such summary metadata for dynamic analyses, in terms of both effectiveness and efficiency.

II. Motivation and Challenges

Motivating Example. We now present a motivating example program that will be used throughout the paper. Figure 1(a) shows the simple program containing the implementation of a data structure `IntList`, and a client that creates and uses two `IntList` objects (line 21 and 22). Lines 26–29 show a loop, each iteration of which adds an element into the `IntList` object created in line 22. Lines 33 and 34 retrieve elements from the two `IntList` objects, respectively.

Figure 1(b) shows the dynamic data dependence graph of the program. Each node in the dependence graph is represented by the line number of a source code statement, annotated with an integer i representing the i^{th} execution of the statement. Each edge represents a data dependence relationship between two statement executions. A dashed-line box excapsulates nodes and edges in a single method execution. In this paper, we address the summarization of the data dependence relationships. Summaries for control dependence can be similarly computed, and will be described in the future work.

A typical object-oriented application makes heavy use of libraries and frameworks, such as class `IntList` in our example. Library methods (such as `add` and `get`) are frequently executed; tracking and profiling all their executions can be extremely expensive. For instance, the loop in line 26 may iterate for a great number of times, leading to a prohibitively large execution trace and dependence graph.

A natural idea to reduce the profiling cost would be to summarize the *effects* of such library methods and apply the summaries when they are invoked. Summaries can be computed via static analysis, as proposed by Horwitz, Reps, and Binkley [7], by summarizing all possible statically computed dependence relationships between the inputs and outputs of a method. When a library method is invoked during a dynamic dependence analysis, the statically computed summaries can be applied to find the appropriate dependence relationships, while avoiding the dependence profiling of the library method.

However, the overly-conservative modeling in a static analysis can lead to rather imprecise summary information (*e.g.,* [9], [31]). For example, a typical static analysis unifies all possible effects of the invocation of a polymorphic method resulting in (spurious) dependence relationships between the site of the method invocation and the instructions in all possible implements that realize the polymorphic method in question. Dynamic analyses based on such spurious relationships may not produce useful information, because polymorphism is a widely used feature of object-oriented programming.

We propose to compute summaries using dynamically observed information. We envision that such dynamic summaries may be computed in the following two ways. (1) The dynamic information may be generated and recorded in a separate training phase in the from of execution traces, using a test suite; the dependence information and subsequently the summaries are computed from the execution traces and stored to a disk file for use in a future dynamic dependence analysis. (2) Alternatively, the dependence information and thus, the summaries may be computed entirely *online*, *i.e.,* during execution, and then be used later in the same execution. For instance, for the program in Figure 1, we can use the first execution of `add` (or `get`) to compute its dependence summary and apply it to reduce the profiling cost of its second execution. While the first approach to compute summaries is used in this work, it can be easily modified to adopt the second approach. We now present the challenges against computing dynamic-based summaries.

Challenge 1: Defining Dependence Summaries in Java. Horwitz *et al.* [7] pioneered the work of summary-based dependence analysis—a summary edge of a procedure connects an input parameter i with an output parameter o, abstracting away intermediate dependence relationships inside the procedure. This summary edge indicates that the value in input i may contribute (directly or transitively) to the computation of the value in output o. While we use a similar idea in this work, the handling of modern object-oriented languages like Java imposes many new research challenges that none of the existing techniques have yet addressed. For example, the notions of the input and output of a Java method can be much broader than those discussed in [7], because the method can access not only its parameters (*i.e.,* passed into the method from its caller), but also (potentially) all objects reachable from them. In this paper, *for a given method, the inputs are considered as a set of (heap or stack) locations that existed before the call and that can be read in the call, and the outputs are considered as a set of locations that can be written in the call and that will still exist after the call.*

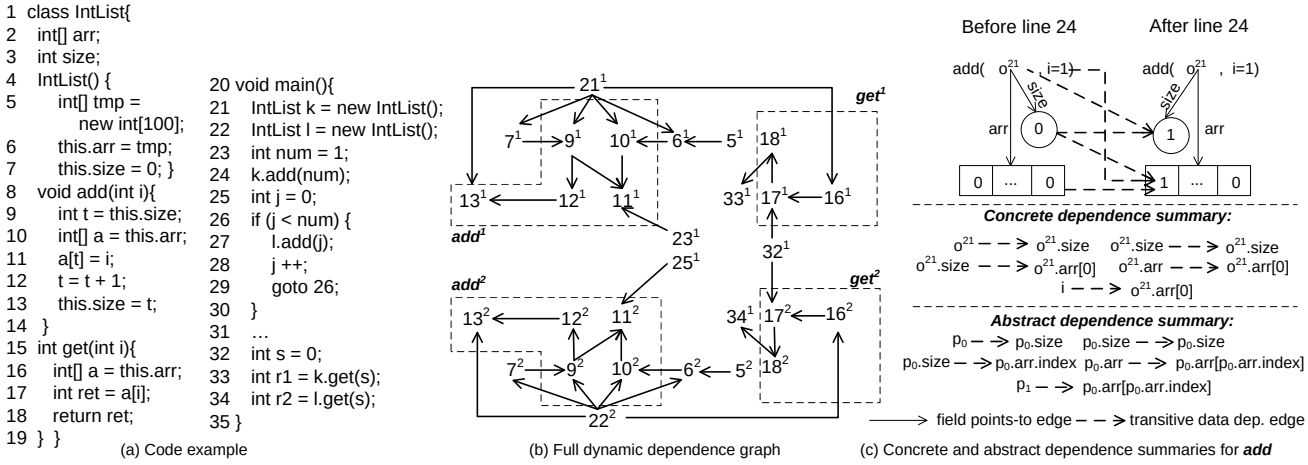

Fig. 1: Example program, dependence graph, and summaries.

Example. To illustrate, consider the top part of Figure 1(c) that shows transitive data dependence relationships between a set of input memory locations before the call at line 24 and a set of output memory locations after the call. o^{21} denotes the run-time `IntList` object created in line 21. Each box represents a reference-typed field and each circle represents a primitive-typed field. From the example, we can see that the value contained in location $o^{21}.arr[0]$ (after the call) depends, transitively, on the values in three other (heap or stack) locations before the call: $o^{21}.size$, $o^{21}.arr$, and the parameter i. The five transitive dependence edges (shown in the middle part of Figure 1(c)) form the concrete dynamic summary for the method `add`. Note that each such location in the summary is named using a combination of a parameter (root) object (*e.g.,* o^{21}) and an *access path* that specifies how the location is reached from the parameter (*e.g.,* arr). A data location unreachable from a parameter (or a global variable) is either inaccessible in the method, or does not escape the method and thus will have no impact beyond the scope of the method. Of course, for a different execution of the method, a different set of transitive dependence edges may be generated. In this work, summary edges computed from different executions in the training phase are combined to serve as the eventual summary for the method.

Challenge 2: Abstracting Concrete Summaries. A concrete dynamic summary contains information specific to the execution of the method where the summary is computed. For example, the use of the concrete object o^{21} in the concrete summary (the middle part of Figure 1(c)) prevents us from applying and reusing the summary in the future execution of `add`. For the summary of a specific invocation of a given method, the concrete information should be replaced by abstract information, such that, the reproduction of the concrete information may be possible for all invocations of the method in question, from the abstract information. In other words, the abstraction of concrete summaries allows application of the summaries for all invocations of the method, instead of specific invocations. To do this, we propose to use symbolic names for parameters. For example, the concrete object o^{21} is replaced by a symbolic name p_0, and the concrete stack variable i is replaced by a symbolic name p_1. A corresponding set of *abstract summary edges* is shown at the bottom of Figure 1(c).

Challenge 3: Accounting for Varying Method Behavior. An important observation on which our technique is based is that while different executions of a library method may handle different incoming data from different clients, their behaviors in these executions often do not differ significantly. Such behavioral consistency can be attributed to the relative simplicity of many library methods, and that all behaviors (in our case, data dependencies) may be realized in a small number of executions.

However, a method invocation may take objects of different types as parameters. Although these types may have the same supertype, their fields can differ significantly, or they might result in the execution of entirely different method implementations due to polymorphism, and thus abstract summaries generated for one method execution may not be directly applicable in a different execution.

When an abstract summary is generated, we additionally record the type information of each parameter with the symbolic name representing the parameter. Before a summary edge is concretized (during the dependence analysis), we first check whether the recorded type in the edge matches the type of its corresponding actual parameter in the current execution, and only apply those summaries that match.

Challenge 4: Precise Handling of Array Accesses. Precise handling of array accesses can be critically important in the dependence analysis of software. Particularly, when an abstract summary involving an array access is applied at a method call, we wish to understand precisely which array element is used or defined inside the method execution. Without such information, spurious dependence relationships may be generated — any data retrieved from an array would depend on any data added into the array. Precise handling of array accesses

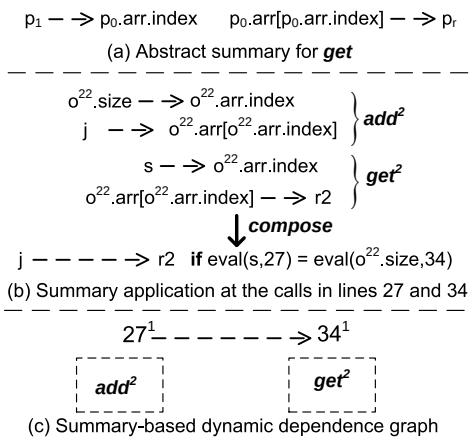

Fig. 2: Summary application at call sites in lines 27 & 34.

is challenging because the index used to access the array is often not an output of the method, and thus, no summary will contain its dependence information. To solve the problem, we create a special symbolic name for each array index. If the accessed array is an input or output of the method, the index used to access the array is considered as a (special) output, and thus the transitive dependence relationships leading to the computation of the index would be included in the summary. If the index is a constant value (*i.e.,* its computation does not depend on any method input), this constant value is recorded in the summary. In our example, index t used in line 11 of Figure 1(a) is abstracted by a symbolic name $p_0.arr.index$, which is dependent on the symbolic location $p_0.size$. When the summary is applied, we concretize $p_0.size$ to obtain its run-time value (before a call to `add`), which will be used to understand which array element is accessed during the call.

Example. Figure 2 shows the application of the summaries for `add` and `get` at their second call sites (*i.e.,* lines 27 and 34 in Figure 1(a)), with a focus on how we recover the (transitive) dependence relationship between the int value retrieved from the `IntList` object (in line 34) and the one added into the object (in line 27). Symbol p_r in Figure 2(a) denotes the symbolic name for the return variable in `get`. Figure 2(b) shows the summary application process—all symbolic names are replaced with their actual (run-time) locations. By composing the concretized summaries for `add` and `get`, we see that there exists a transitive data dependence relationship between r_2 and j if the array indices involved in the summaries of `add` and `get` are the same.

By tracking dependence for array indices, we see that the array index in `add` is computed from $o^{22}.size$ and the array index in `get` is computed from s, and hence, if a certain condition holds between the values of $o^{22}.size$ and s, r_2 will depend on j. For most library classes (such as data structures in `java.util`), different methods in the same data structure often use the same algorithm to compute array indices from the input(s). For instance, `add` and `get` in our example directly use inputs as indices, while `put` and `get` in the Java `HashMap` use the hashcodes of their input (key) objects to compute array indices. Based on this observation, we assume this condition is "equals": if the values of the inputs on which the array indices depend in the two summaries are equal, the array elements accessed in the two methods are considered to be the same. In our example (Figure 2(c)), therefore, a transitive dependence relationship is added between the statement that defines r_2 (*i.e.,* 34^1) and the statement that defines j (*i.e.,* 25^1), because the values of both $o^{22}.size$ (before line 27) and s (before line 34) are 1. While "equals" is sufficient for precise tracking of most methods in standard Java libraries, this condition can be easily redefined by the user to handle other (user-defined) data structures. Note that one situation that cannot be appropriately handled is that a range of array locations are accessed in the method. This case of accessing multiple array locations is further discussed in Section VIII.

III. Approach and Concepts

This section formally defines concrete and abstract dependence summaries, and in this context, presents our core technique that computes and uses summaries to improve the efficiency of dynamic data dependence analysis.

Concrete Summaries. Let us first define the inputs and outputs of a Java method. Note that our implementation treats static fields as additional parameters to a Java method.

> **Definition 1: *Heap Graph (HG).*** *We use $c : m(a_0, a_1, \ldots, a_n)$ to denote a call site c. If m is an instance method, the receiver object is a_0. For each actual argument a_i, we define a location graph g_i, in which each node is a heap location (i.e., a reference-typed or primitive-typed field) in an object reachable from the root object referenced by a_i; each edge connects a parent heap location (that references an object) and a child heap location (that is a field in the object).*

Note that the only way to access a node in g_i during the execution of m is to perform a sequence of field dereferences on the root object. In Figure 1(c), we show two example HGs reachable from parameter `this` of method `add` before and after the execution of line 24. Note that the A set of all heap graphs for the call site c is referred to as c's accessible HG set (AHGS), represented as $\mathcal{G}_c$. Additionally, we use $\mathcal{G}_c^{pre}$ and $\mathcal{G}_c^{post}$ to denote, respectively, c's AHGS immediately before and after the execution of c.

> **Definition 2: *Input and Output Sets.*** *For a call site of the form $c : m(a_0, a_1, \ldots, a_n)$:*
>
> *$\mathcal{I}_c = \mathcal{G}_c^{pre} \cup \mathcal{P}_c$, where, $\mathcal{I}_c$ is the input set, $\mathcal{P}_c$ is a set of stack locations representing parameters passed into c.*
>
> *$\mathcal{O}_c = \mathcal{G}_c^{post} \cup \mathcal{L}_c \cup \{r\} \cup \{g_r\}$, where, $\mathcal{O}_c$ is the output set, r is the stack location containing the value to be returned, g_r is the HG defined by r (if r is of reference type), and $\mathcal{L}_c$ is a set of integer indices used in the accesses of the arrays referenced by heap locations in $\mathcal{G}_c^{pre}$, $\mathcal{G}_c^{post}$ or g_r.*

Note that $\mathcal{P}_c$ is not part of the output set because Java is a call-by-value language where the values of the actual parameters cannot be changed by the method. The indices used to access the arrays in the heap graphs are included in the output set. Tracking these indices is necessary for the precise handling of array accesses, as described in Section II. Based on the definition of the input and output set, the concrete summary for a method is defined as follows.

Definition 3: ***Concrete Dependence Summary.*** *For a call site of the form $c : m(a_0, a_1, \ldots, a_n)$, the concrete dependence summary $\mathcal{S}_c$ for m with respect to c is a Cartesian set $\mathcal{I}_c \times \mathcal{O}_c$. Each element of the set is a summary edge that goes from a stack or heap location in the input set to a stack or heap location in the output set, representing a transitive data dependence relationship between values in these two locations during the execution of c.*

Abstract Summaries. As described in Section II (Challenge 2), concrete summaries contain execution-specific information and cannot be reused. Abstraction needs to be performed to replace concrete information with some type of abstract information, so that the abstracted summaries are applicable to all other executions. The abstraction process has two steps. In the first step, we replace each concrete HG node with a combination of the root object and the access path through which this node can be reached. The result of this step is a set of access-path-based concrete summary edges, as shown in the middle part of Figure 1 (c). In the second step we, replace each concrete parameter object or variable with a symbolic name, resulting in the final abstract summary that can be applied in other executions of the method (as shown in the bottom part of Figure 1 (c)). Note that in our implementation, these two steps are combined in one single summary generation phase. They are discussed separately in the paper for the clarity of presentation.

Definition 4: ***Abstract Dependence Summary.*** *For a call site of the form $c : m(a_0, a_1, \ldots, a_n)$, p_i denotes the symbolic name for parameter a_i. For an access-path-based concrete dependence summary of the form $\bigcup\{o_i.f_0.f_1 \ldots f_* \dashrightarrow o_j.g_0.g_1 \ldots g_*\}$ (where o_i and o_j are the objects pointed to by parameters a_i and a_j, and f_k and g_l are field names), its corresponding abstract dependence summary is of the form $\bigcup\{p_i.f_0.f_1 \ldots f_* \dashrightarrow p_j.g_0.g_1 \ldots g_*\}$.*

For concrete summary edges involving array accesses, e.g., $o_i.f_0.f_1 \ldots f_*[l] \dashrightarrow o_j.g_0.g_1 \ldots g_*$ and $o_k.h_0.h_1 \ldots h_* \dashrightarrow l$, their corresponding abstract summary edges are $p_i.f_0.f_1 \ldots f_*[p_i.f_0.f_1 \ldots f_*.index] \dashrightarrow p_j.g_0.g_1 \ldots g_*$ and $p_k.h_0.h_1 \ldots h_* \dashrightarrow p_i.f_0.f_1 \ldots f_*.index$, where $p_i.f_0.f_1 \ldots f_*.index$ is the symbolic name for index l.

Note that an HG node may have multiple access paths from the root object. For each such access path used in the method, we will generate a summary edge for it. Eventually, abstract summaries computed for all call sites that invoke method m during the training phase are combined and used as m's summary for the future dependence analysis.

Summary-Based Dependence Analysis. A summary-based dependence analysis profiles the execution of all methods except those that have abstract dependence summaries. Before presenting the summary-based analysis, we first define regular dynamic dependence analysis.

Definition 5: ***Dynamic Data Dependence Graph.*** *A dynamic data dependence graph $(\mathcal{V}, \mathcal{E})$ has node set $\mathcal{V} \subseteq \mathcal{D} \times \mathcal{N}$, where each node is a static statement ($\in \mathcal{D}$) annotated with an integer i ($\in \mathcal{N}$), representing the i-th execution of this statement. An edge from a^j to b^k ($a, b \in \mathcal{D}$ and $j, k \in \mathcal{N}$) shows that the j-th execution of statement a writes a (heap or stack) location that is then used by the k-th execution of b, without an intervening write to that location. If an instruction accesses a heap location through $v.f$, the reference value in stack location v is also considered to be used.*

An example dynamic dependence graph is shown in Figure 1 (b). Based on the definitions of abstract summary and regular data dependence graph, we give the definition of summary-based data dependence graph.

Definition 6: ***Summary-Based Data Dependence Graph.*** *A summary-based dynamic data dependence graph $(\mathcal{V}, \mathcal{E} \cup \mathcal{T})$ is a regular dynamic data dependence graph augmented with an additional set of transitive dependence edges $\mathcal{T}$. There exists a transitive edge from node a^j to b^k, if a^j writes a memory location l_1 that belongs to the input set of a method call and b^k reads a memory location l_2 that belongs to the output set of the same call, and there exists a relationship $p_i.f_0.f_1 \ldots f_* \dashrightarrow p_j.g_0.g_1 \ldots g_*$ in the abstract summary of the invoked method, such that the heap location $o_i.f_0.f_1 \ldots f_*$ (before the call) is the same location as l_1 and the location $o_j.g_0.g_1 \ldots g_*$ (after the call) is the same location as l_2.*

A transitive dependence edge can also be added from a^j to b^k if there exist two pairs of relationships in the abstract summary ($p_j.g_0.g_1 \ldots g_* \dashrightarrow p_i.f_0.f_1 \ldots f_*[p_i.f_0.f_1 \ldots f_*.index]$, $p_k.h_0.h_1 \ldots h_* \dashrightarrow p_i.f_0.f_1 \ldots f_*index$) and ($p_r.v_0.v_1 \ldots v_*[p_r.v_0.v_1 \ldots v_*.index] \dashrightarrow p_m.q_0.q_1 \ldots q_*$, $p_t.u_0.u_1 \ldots u_* \dashrightarrow p_r.v_0.v_1 \ldots v_*.index$), such that $o_m.q_0.q_1 \ldots q_*$ is the same location as l_2, $o_j.g_0.g_1 \ldots g_*$ is the same location as l_1, $o_i.f_0.f_1 \ldots f_*$ and $o_r.v_0.v_1 \ldots v_*$ refer to the same array object, and values in $o_k.h_0.h_1 \ldots h_*$ and $o_t.u_0.u_1 \ldots u_*$ (used to compute indices) are the same.

As discussed earlier in Section II, two (abstract) array slots of the form $p_i.f_0.f_1.\ldots.f_*[p_i.f_0.f_1 \ldots f_*.index]$ and $p_r.v_0.v_1 \ldots v_*[p_r.\ v_0.v_1 \ldots v_*.index]$ are considered to be the same location in an execution, if (1) the two concrete array objects in locations $o_i.f_0.f_1 \ldots f_*$ and $o_r.v_0.v_1 \ldots v_*$ are the same, and (2) the values of the inputs that the two indices depend on (*i.e.,* $o_k.h_0.h_1.\ldots.h_*$ and $o_t.u_0.u_1.\ldots.u_*$ in the definition) are the same. A transitive edge is *not* added

if one or both of the indices depend on multiple inputs, because it is unclear how the indices are computed from these inputs and how to compare their values. Note that this treatment can potentially introduce both unsoundness and imprecision. However, as discussed in Section VIII, it is both precise and lossless for many library methods (especially in the Collections framework).

IV. Analysis Implementation

This section presents the details of our implementation of the summary-based dependence analysis. We perform load-time bytecode instrumentation for non-JDK classes using ASM [2], a Java bytecode manipulation framework. In contrast, we instrument the classes in the standard Java library (JDK) prior to execution—several of these classes cannot be instrumented during load time due to the technical requirements of the JVM to load them prior to the instrumenter. The goal of the instrumentation is to produce a detailed trace that records the execution of each instruction in the program and the heap/stack location it accesses. We instrument to record the execution of every instruction in the program and the heap/stack locations that each accesses. With the trace, we construct the dynamic data dependence graph, which then enables the computation of dynamic slices. We assign a unique ID to each run-time object, which is used to identify the object and its fields in the execution trace.

The computation and use of the dynamic dependence summaries are performed in the following phases.

Phase I. Summary Generation in the Training Execution. We produce the abstract summaries for the methods being summarized (these can be user-specified in a configuration file, by method, class or package) by analyzing the execution traces. For each method m, we find all instances of m's execution, and for each instance in the trace, we use a worklist-based algorithm to compute an abstract summary according to the approach and definitions in Section III. The result of the summary generation is a mapping of inputs (formal method parameters or accessible public fields) to outputs (formal method parameters or accessible public fields) that they influenced, expressed as abstract summary edges.

Phase II. Summary Application in Dependence Analysis. To apply the summaries to dynamic dependence analyses, the client would choose to instrument her test case (*i.e.,* execution) with knowledge of the previously gathered summaries. In the client's execution, the instrumented execution would check each method call to determine if a summary for it exists. If the summary is not provided for a method call, the execution and instrumentation proceeds with regular dependence profiling. However, if a summary does exist for the method: (1) the abstract summary edges are obtained, (2) the actual inputs and outputs are matched to the LHS and RHS symbolic names of the summaries, and then (3) the concretized summary edges are recorded to the trace. To illustrate, consider the example of applying summaries at the second call of `get` (line 34 in Figure 1 (a)). Abstract summary edges are concretized into $o^{22}.arr[o^{22}.arr.index] \dashrightarrow r_2$ and $s \dashrightarrow o^{22}.arr.index$ before being recorded into the trace.

Phase III. Dependence Graph Computation and Use. A dynamic dependence graph is computed by recovering dependence relationships from the trace, as described in Definition 5. When (concretized) summary edges are encountered, the location matching approach described in Definition 6 is used to recover the missing relationships. Using this dynamic dependence graph, the client can perform dynamic analyses, such as dynamic slicing.

V. Evaluation

To evaluate how the reuse of dynamic dependence summaries affects analysis, we implemented our technique and performed two experiments. The first experiment is designed to assess the extent of cost savings that can be realized with dynamic dependence summaries with respect to exhaustive instrumentation. The second experiment is designed to determine how the use of dynamic dependence summaries affects the accuracy of a dynamic dependence analysis. As an instrument of evaluation, we used dynamic slicing to find faults in test-case failures with both exhaustive and summarized dynamic dependence graphs. We designed these two experiments to help inform future researchers and software developers of the tradeoffs involved in choosing either the reuse of dynamic dependence summaries or exhaustive analysis of all external components.

The goal of these experiments is to answer the following research questions.

RQ1:	How does the reuse of dynamic dependence summaries affect the costs of dynamic analysis?
RQ2:	How does the reuse of dynamic dependence summaries affect the accuracy of dynamic analysis?

Experimental Subjects. We performed all experiments on a quad-core Intel i5-2430M 2.40GHz machine, running 32-bit Ubuntu (version 12.04). We used the OpenJDK 6 JVM with a maximal heap size of 1GB. We summarized all methods in the standard JDK (*i.e.,* `rt.jar`).

We performed our performance evaluation using five large programs—ANTLR ($>$ 35KLOCs), BLOAT ($>$ 41KLOCS), FOP ($>$ 102KLOCs), JYTHON ($>$ 245KLOCs), and PMD ($>$ 60 KLOCs)—from the DaCapo Benchmark Suite [1], which is provided with inputs to facilitate performance benchmarking. We performed our accuracy evaluation using the NANOXML program ($>$ 7KLOCs) and its test cases and bugs, which are provided by the Subject-artifact Infrastructure Repository [5]. We utilize 21 versions of NANOXML: one version for training the summaries, and 20 subsequent, faulty versions for evaluation. We compute and reuse dynamic dependence summaries for all external components, outside of the NANOXML code, which includes large portions of the Java Standard Library.

Independent Variable. We use a single independent variable for both Experiment 1 and Experiment 2: the dynamic slicing technique used for performing the analysis. We performed our dynamic dependence analysis in two ways:

Treatment 1: Exhaustive Analysis. All components that are executed, whether core or external, are instrumented and analyzed.

Treatment 2: Summary-based Analysis. Only the program under test is instrumented, and summaries from all external components are reused.

Experiment 1: Analysis Efficiency. In Experiment 1, we investigate the impact of reused dependence summaries on the costs involved in performing dynamic dependency analysis.

Experiment 1 Dependent Variables. We assess the costs of the dynamic slicing techniques in two ways:

Metric 1: Execution-Trace Size (S). Size of the execution trace required for each analysis.

Metric 2: Analysis Running Time (T). Time to run the dependence analysis.

Experiment 1 Results. Five large benchmarks from the DaCapo benchmark set (2006 release) were chosen for this experiment.[1] Each benchmark was run for one iteration with the large workload. *The exhaustive analysis could not scale to any of these benchmarks*—running the analysis with trace generation enabled could not finish in 24 hours for even the smallest program (*i.e.,* FOP) in this set. Disk I/O and frequent OS context switches obviously became the bottleneck that caused a significant slowdown. In order to enable the comparison, we slightly modified the exhaustive analysis and ran it twice, once to generate (small) traces only for summary computation and once to collect performance statistics without generating any trace. In the first run, traces were generated for 10 random executions of each library method and used to compute summaries. This did not take too long because the logs for the majority of the method invocations were not dumped to disk. As discussed in Section II, library methods often exhibit simple behaviors and 10 executions may be sufficient to summarize their dependence effects. In fact, this handling describes an intended scenario for summary usage—summaries are generated in a small number of executions of the selected methods and applied for a large number of invocations of these methods in a (future) analysis.

We collected the performance statistics in the second run where trace generation was completely disabled (while the instrumented code was still executed). Although we did not produce a trace in this run, we measured the size of the trace by maintaining a global size counter, which is increased by the size of a log message every time the message needs to be written. To enable a fair comparison, the performance statistics for the summary-based analysis were also collected in the same manner. It is important to note that the summary-based analysis was scalable even when the trace was generated (it finished in 3 hours for all the benchmarks).

These results show that performing full instrumentation that includes both application code and external-component code incurs an average $112\times$ runtime overhead. The summary-based approach successfully reduced this overhead to $43\times$, which provides a greater than $2\times$ speed up over exhaustive. In a real-world situation where all traces need to be generated, the overhead reduction can be much larger than this number. Summary-based analysis provides trace sizes that are, on average, 44% smaller than the exhaustive trace sizes.

[1]The DaCapo results are limited to the subjects that were scalable to a 24-hour computational requirement for the exhaustive analysis. Subsequent iterations of this work will include results that relax this requirement.

TABLE I: Experiment 1 Results. T_O is original running time of the program, $T_{E|S}$ and $S_{E|S}$ are running times of dependence analysis and trace size for each technique.

		Exhaustive		Summary	
Subject	T_O *(sec)*	T_E (sec)	S_E (# instr. $\times 10^9$)	T_S (sec)	S_S (# instr. $\times 10^9$)
ANTLR	10.4	451.3	4.2	248.5	2.6
BLOAT	28.3	9061.5	214.8	2520.7	79.5
FOP	2.7	68.6	0.7	35.5	0.3
JYTHON	68.6	2873.9	51.9	1922.8	38.8
PMD	11.3	1448.5	28.4	718.7	16.1

These cost savings of over twice as efficient were achieved by summarizing only one common library of each of the five DaCapo programs: the standard JDK library. These programs each use several other large external libraries that we did not summarize. When summarizing all libraries, the savings enabled would be much greater, as exhibited in Experiment 2.

Experiment 2: Analysis Accuracy. Motivated by the savings found from Experiment 1, we next investigated the extent to which these savings are realized at the expense of accuracy for client analyses. In Experiment 2, we investigate the impact of reused dependence summaries on the accuracy of dynamic analysis, specifically dynamic slicing for debugging. As discussed in Section VIII, summary-based dynamic dependence analysis can be both unsound and imprecise. However, the degree to which this affects the results of analysis, in practice, is yet to be known. As such, this experiment is designed to answer this question, at least for our experimental data.

Using the training version of NANOXML, and the 20 subsequent faulty versions, we investigate this issue by performing backward dynamic slicing to determine accuracy of the slice. We injected faults at the points in the program at the beginning of the execution — during the reading of the data files. We identified our slicing criterion, for each of the 20 faulty versions by observing the output stream and identifying the first point that the output from the faulty version differs from the output of the correct version. That is, we define the slicing criterion as the output instruction (and its outputted data) being executed at the moment that the test-case output first violates the test oracle (*e.g.,* the first character difference). We then slice based upon that instruction, the variable that was used to hold the externally observed incorrect data, and the specific execution instance of that instruction (remember, our traces include all executions instances of each instruction). As such, the faults that we slice are "deep" — the fault execution occurs at the beginning of the trace, the propagation of the fault's state infection spans the entire execution, and the slicing criterion is placed at the output manifestation at the end of the execution.

Experiment 2 Dependent Variables. We present our results with the following five metrics:

Metric 1: Found Bugs. The ratio of the dynamic slices that include the fault. This metric determines the degree of unsoundness that the summary-based dynamic analysis brings.

TABLE II: Experiment 2 Results

Metric	Exhaustive	Summary
Found Bugs (ratio)	20/20	18/20
Mean Size of Trace (# of instruction instances)	392946.65	143150.7
Mean Size of the Slice (# of instructions)	239.95	213.45
Mean Runtime Overhead (ratio)	3838.39	41.69
Mean Slicing Time (milliseconds)	2608	1899

Metric 2: Size of Trace. The size of the trace file, in terms of the number of instruction instances, that is needed to perform the slicing.

Metric 3: Size of the Slice. The size of the resulting slice, as a set of source-code instructions.

Metric 4: Runtime Overhead. The slowdown of the original non-instrumented program, in terms of a multiplicative factor of the original time. The overhead is computed as $\frac{(\textit{instrumented time})-(\textit{non-instrumented time})}{(\textit{non-instrumented time})}$.

Metric 5: Slicing Time. The time to required to compute the dynamic slicing.

Experiment 2 Results. We present our results for Experiment 2 in Table II and Figures 3–6. Table II shows the aggregated data across all 20 bugs, whereas Figures 3–6 break down results per bug. Figure 3 presents the size of the traces for each bug, for each technique, in terms of the number of instruction instances composing the execution trace. Figure 4 presents the size of the resulting slices for each bug, for each technique, in terms of the unique source-code instructions in the slice. Figure 5 presents the multiplicative factor of the overhead of the necessary instrumentation over the non-instrumented version. Finally, Figure 6 presents the time required to compute the dynamic slice. In each of these sub-figures, lines are drawn to help the reader with tracking the results for each technique — they do not denote a continuous range between bugs.

On average, the costs for exhaustive slicing are substantially more than the costs for summarized slicing. Particularly, the runtime overhead costs of exhaustive slicing were dramatically greater than those of summarized slicing — and these are costs that would be incurred for *every* execution to be sliced.

When examining the bugs that were found, we found that 18 out of the 20 bugs were localized in the dynamic slice that we computed with the summary-based technique. Such a localization rate is attributed to faults propogating through multiple dependency chains, and as such, the ability to slice back to the fault depends on at least one such chain persisting. The exhaustive slicing technique found 100% of the bugs, as expected. The summary-based technique missed 2 out of the 20 bugs, and this was due to a missing dependency from the training of a summary.

VI. Analysis and Discussion

In this section we present a discussion of both studies to draw more general conclusions. In addition, we discuss each research question that motivated our evaluation, in order. In every measure of performance costs, we found that as a whole, the use of summarized dynamic data substantially reduced

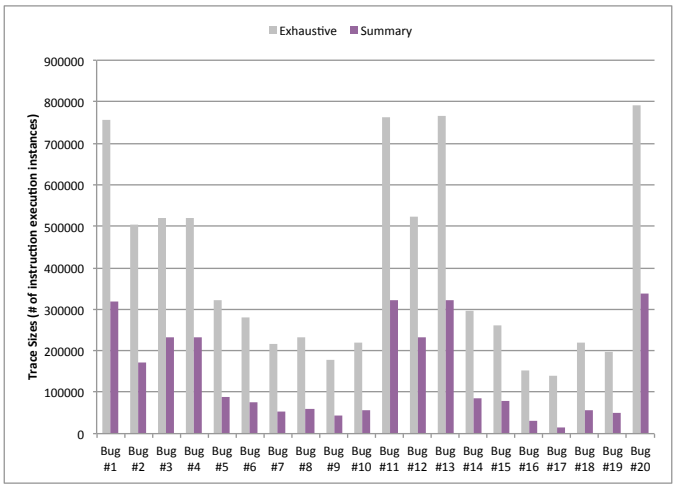

Fig. 3: Trace Sizes.

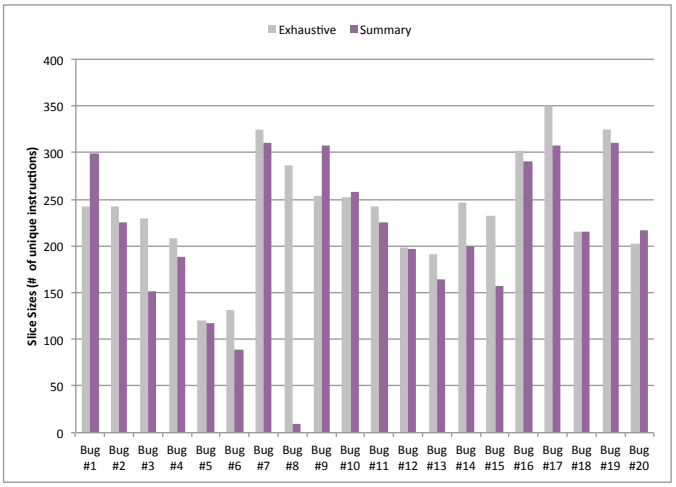

Fig. 4: Slice Sizes.

costs. When examining the results from Experiment 1, we find that the exhaustive technique often exceeded our thresholds for costs, whereas the summary-based technique was efficient.

As such, **to RQ1, we assess:** The reuse of dynamic summaries caused the costs involved in performing dynamic dependency analysis to be significantly reduced.

In terms of analysis accuracy, we evaluated a debugging task using dynamic slicing. When examining the results from Experiment 2, we find that although we are utilizing summarized results from past executions to approximate the dependencies of external components, the summarized slicing technique produced accurate results: 90% of the bugs were found with the summary-based technique.

As such, **to RQ2, we assess:** The reuse of dynamic summaries caused a small loss of accuracy as a consequence of the gains in performance.

Taken as a whole, our results suggest that the reuse of dynamic summaries can provide a way to make dynamic de-

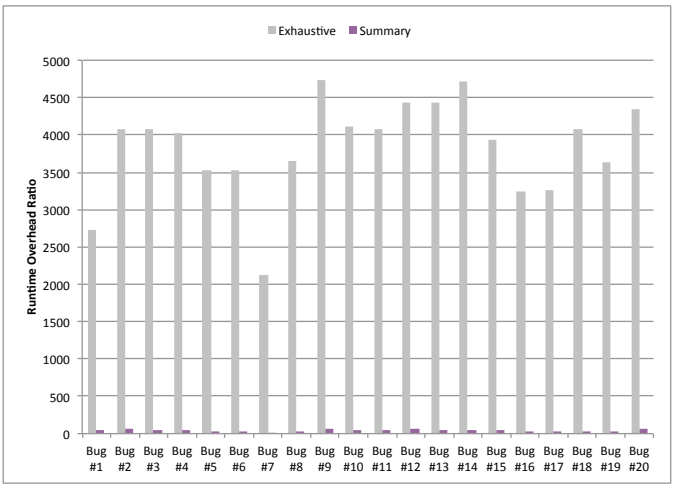

Fig. 5: Runtime Overhead.

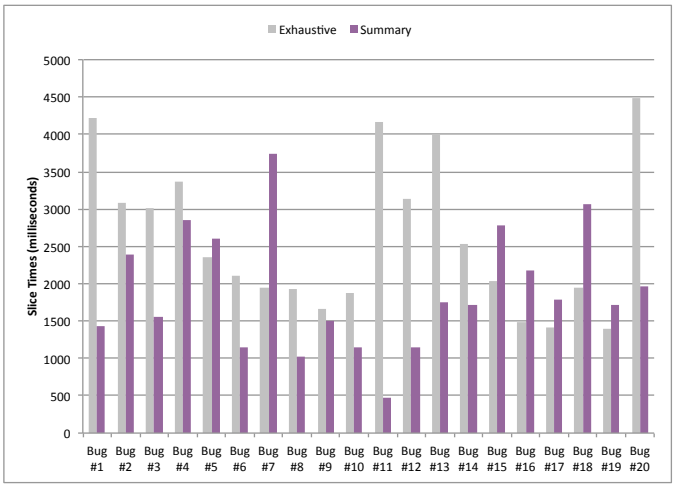

Fig. 6: Slicing Time.

pendency analysis more feasible for real-world use, with modest losses in accuracy. In software-development projects that accept such small inaccuracies (*e.g.,* in non-critical systems), or for analyses that are more heuristic in nature (*e.g.,* bloat analysis), our results suggest that the tradeoffs favor reuse of dynamic summary information for external components.

Threats to Validity. Threats to external validity arise when the results of the experiment are unable to be generalized to other situations. In this experiment, we evaluated the impact of using dynamic dependency summaries on two client programs, with their set of dependent external components, and thus we are unable to definitively state that our findings will hold for programs in general. However, we are confident that these results are indicative of the impacts of such summarization. The external components in this study are the Java Standard Library, which is a dependency that many other programs will also have and thus the effects on efficiency and effectiveness of dynamic summarization of that common library is likely to hold for those programs. Moreover, the significant gains exhibited in our results gives strong evidence that our approach, at least, shows promise for use in practice.

Threats to construct validity arise when the metrics used for evaluation do not accurately capture the concepts that they are meant to evaluate. Our experiments measured the costs involved in performing tracing and dynamic slicing in terms of computational time and data storage. Although our results give an indication of the degree of such costs, our implementation can be greatly optimized in both regards. Our tracing information is verbose and our implementation is not optimized. However, this limitation does not affect the overall result, as this same implementation was used for both treatment techniques; *i.e.,* the direction and magnitude of the difference between the results should not significantly change when these factors are optimized. Also, our experiments measured the accuracy of the slicing techniques in finding faults. Although these metrics give an indication of the accuracy of our results, they do not give a sense for how these will affect either developer time or client analyses that build upon such results. Further studies will need to be conducted to assess the impacts on such clients of these analyses.

VII. Related Work

While there exists a body of work related to the proposed technique, discussion in this section focuses on techniques that are most closely related to ours.

Summary-Based Program Analysis. Procedural summaries have been computed and used widely in the static analysis community to achieve modularity and efficiency. Summary functions for interprocedural analysis date back to the functional approach in the work by Sharir and Pnueli [21], with refinements for Interprocedural, finite, distributive, subset (IFDS) problems from Reps *et al.* [19] and for interprocedural distributive environment (IDE) dataflow problems from Sagiv *et al.* [16]. Yorsh *et al.* [28] use conditional micro-transformers to represent and manipulate dataflow functions. Rountev *et al.* [17], [18] propose a graph representation of dataflow summary functions, that generate and use summaries, to reduce the cost of whole-program IDE problems. Xu *et al.* [26] propose a summary-based analysis that computes access-path-based summaries to speed up the context-free-language (CFL)-reachability-based points-to analysis. Dillig *et al.* [4] propose a summary-based heap analysis targeted for program verification that performs strong updates to heap locations at call sites. Ranganath and Hatcliff [15] statically analyze the reachability of heap objects from a single thread, for slicing concurrent Java programs. Salcianu and Rinard [20] propose a regular-expression based, purity and side effect analysis for Java, to characterize the externally visible heap locations that a given method mutates. This is similar in spirit to the heap location-based dependence summarization as proposed in our work. Inspired by such static analyses, the work presented in this paper, is the first technique to compute and use dynamic dependence summaries. Dynamic dependence summaries, can potentially better inform dependence information more precisely than static dependence summaries, by leveraging information collected at runtime.

Dynamic Slicing. Since first being proposed by Korel and Laski [9], dynamic slicing has inspired a large body of work on efficiently computing slices and on applications to a variety of software engineering tasks. Early work from Kamkar *et al.* [8] introduces the theory behind summary-based slicing for a stack-only language. Our approach attempts to be more general; in that it handles all features of a modern object-oriented language. A general description of slicing technology and challenges can be found in Tip's survey [22] and Krinke's thesis [10]. The work by Zhang *et al.* [30]–[34] considerably improved the state of the art in dynamic slicing. This work includes, for example, a set of cost-effective dynamic slicing algorithms [31], [33], a slice-pruning analysis that computes confidence values for instructions to select those that are most related to errors [30], a technique that performs on-line compression of the execution trace [32], and an event-based approach that reduces the cost by focusing on *events* instead of individual instruction instances [34]. Apart from Zhang's work, Wang and Roychoudhury [23] develop optimizations to compress bytecode execution traces for sequential Java programs resulting in space efficiency. Moreover, they develop a slicing algorithm, applicable directly on the compressed bytecode traces. Wang and Roychoudhury's work on Hierarchical Dynamic Slicing [24] aims at guiding the programmer through large and complex dependence chains in a dynamic slice, with debugging as the primary application. Deng and Jones [3] propose a dynamic dependency graph that encodes frequency of inferred traversal in order to prioritize heavily trafficked flows. Xu *et al.* [25] propose an abstraction-based approach, to scale a class of dynamic analyses that need the backward traversal of execution traces. This approach employs user-provided analysis-specific information to define equivalence classes over instruction instances, so that dynamic slicing can be performed over bounded abstract domains instead of concrete instruction instances, leading to space and time reduction. Our technique achieves efficiency from a different angle—we use summaries generated for library classes to speed up general dynamic data dependence analysis, and thus, all dynamic analyses that need dependence information may benefit from this technique.

VIII. Limitations and Future Work

Unsoundness and Imprecision. It is important to note that the summary-based dynamic dependence analysis can introduce both unsoundness and imprecision. On one hand, the quality of an abstract summary relies on the coverage of the tests used to train the summary. Thus, a summary may miss certain dependence relationships due to the lack of test cases. On the other hand, the abstract summary aggregates information from multiple executions of the method. Thus, the application of the summary for a specific invocation may generate additional *spurious* dependence relationships that would not have been added in a regular dependence analysis.

To reduce these negative effects, it is important to find methods that are suitable for summarization. Our experience shows that API methods in large libraries may be good sources of summarizable methods because they often do not mutate client objects, their behaviors are often relatively simple, and even similar under different clients. In our evaluation (Section V), the entire Java Standard library is summarized to speed up the dynamic slicing of a real-world application with limited loss (10%) in accuracy to find bugs via slicing.

Future work will develop ways to assess suitability of summarization for methods. Also, adequacy criteria will be developed to inform when the training phase has sufficiently exercised the behaviors in order to summarize them.

Abstracting Multiple Array Accesses. Although our technique improves upon existing work in distinguishing array index accesses, this improvement is limited to method calls that access only a single index. We found that most of our studied methods accessed a single index. Even for methods that access multiple array locations (such as `addAll` in many of the `java.util.*` methods), they are often implemented by invoking methods that access a single array location (such as `add`) multiple times. Hence, our technique can precisely handle array accesses for most of the common and frequently-used data structure methods.

Future work will provide yet further improvement to array summarization by developing ways to summarize access to multiple array indices. We envision summary notation containing logic that abstracts array access ranges.

Additionally in future work, we will evaluate on a larger set of applications; and extend it to address scalability issues for other dynamic analyses (*e.g.,* bloat- and change-impact analysis). Further, we will investigate how dynamic summaries compare with static summaries for various expensive dynamic analyses in terms of performance and effectiveness.

IX. Conclusions

This paper presents a novel summary-based dynamic analysis approach to speed up a class of dynamic analysis techniques for modern applications that use large object-oriented libraries and components. During training, summaries are produced for library methods, which are later used for dependence analysis for improved efficiency. To compute the summary for a library method, we first extract concrete dependence summary edges from its execution trace, and then abstract them with symbolic data. This symbolic data is then used during subsequent analysis by replacing symbolic names with the new concrete data. Our experimental results on real-world software found that applying these summaries in the dependence analysis can significantly save costs despite causing only a modest loss of accuracy.

X. Acknowledgements

This material is based upon work supported by the National Science Foundation under award CCF-1116943. The second author was supported in part by NSF under grant CNS-1321179.

REFERENCES

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[2] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, November 2002.

[3] F. Deng and J. A. Jones. Weighted system dependence graph. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 380–389. IEEE, 2012.

[4] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, pages 567–577, 2011.

[5] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10:405–435, 2005.

[6] J. Holewinski, R. Ramamurthi, M. Ravishankan, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *PLDI*, pages 371–382, 2012.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *TOPLAS*, 12(1):26–60, 1990.

[8] M. Kamkar, N. Shahmehri, and P. Fritzson. Interprocedural dynamic slicing. In *Programming Language Implementation and Logic Programming*, volume 631, pages 370–384. 1992.

[9] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[10] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, University of Passau, 2003.

[11] D. McIlroy. Mass-produced software components. In *Proceedings of the NATO Conference on Software Engineering*, pages 88–98, 1968.

[12] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[13] A. Orso, H. Do, G. Rothermel, M. J. Harrold, and D. S. Rosenblum. Using component metadata to regression test component-based software: Research articles. *Journal of Software Testing, Verification and Reliability*, 17(2):61–94, June 2007.

[14] A. Orso, M. J. Harrold, and D. S. Rosenblum. Component metadata for software engineering tasks. In *International Workshop on Engineering Distributed Objects*, pages 129–144, 2001.

[15] V. P. Ranganath and J. Hatcliff. Pruning interference and ready dependence for slicing concurrent java programs. In *Compiler Construction*, pages 39–56. Springer, 2004.

[16] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[17] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *International Conference on Compiler Construction*, LNCS 3923, pages 2–16, 2006.

[18] A. Rountev, M. Sharp, and G. Xu. IDE dataflow analysis in the presence of large object-oriented libraries. In *CC*, LNCS 4959, pages 53–68, 2008.

[19] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1-2):131–170, 1996.

[20] A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[21] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[22] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[23] T. Wang and A. Roychoudhury. Using compressed bytecode traces for slicing Java programs. In *ICSE*, pages 512–521, 2004.

[24] T. Wang and A. Roychoudhury. Hierarchical dynamic slicing. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 228–238, 2007.

[25] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

[26] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, pages 98–122, 2009.

[27] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, pages 121–136, 2006.

[28] G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *POPL*, pages 221–234, 2008.

[29] X. Zhang. *Fault Localization via Precise Dynamic Slicing*. PhD thesis, University of Arizona, 2006.

[30] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *PLDI*, pages 169–180, 2006.

[31] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *PLDI*, pages 94–106, 2004.

[32] X. Zhang and R. Gupta. Whole execution traces. In *MICRO*, pages 105–116, 2004.

[33] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE*, pages 319–329, 2003.

[34] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *FSE*, pages 81–91, 2006.